PARALLELISM

Laboratory 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

PAR1105

Albert Borrellas Maldonado

Víctor Martínez Murillo

13-11-2019

Fall 2019-2020

1. Task decomposition analysis for the Mandelbrot set computation

In this laboratory session we will explore the tasking model in *OpenMP* for iterative task decompositions. First of all, we will explore the most appropriate ones using *Tareador.*

The *Mandelbrot set*, which is a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognisable two-dimensional fractal shape, is the concept we have based our study on. For each point c in a delimited two-dimensional space, the complex quadratic polynomial recurrence $z_{n+1} = z_n^2 + c$ is iteratively applied in order to determine if it belongs or not to the Mandelbrot set. It is a compact set, since it is closed and contained in the closed disk of radius 2 around the origin

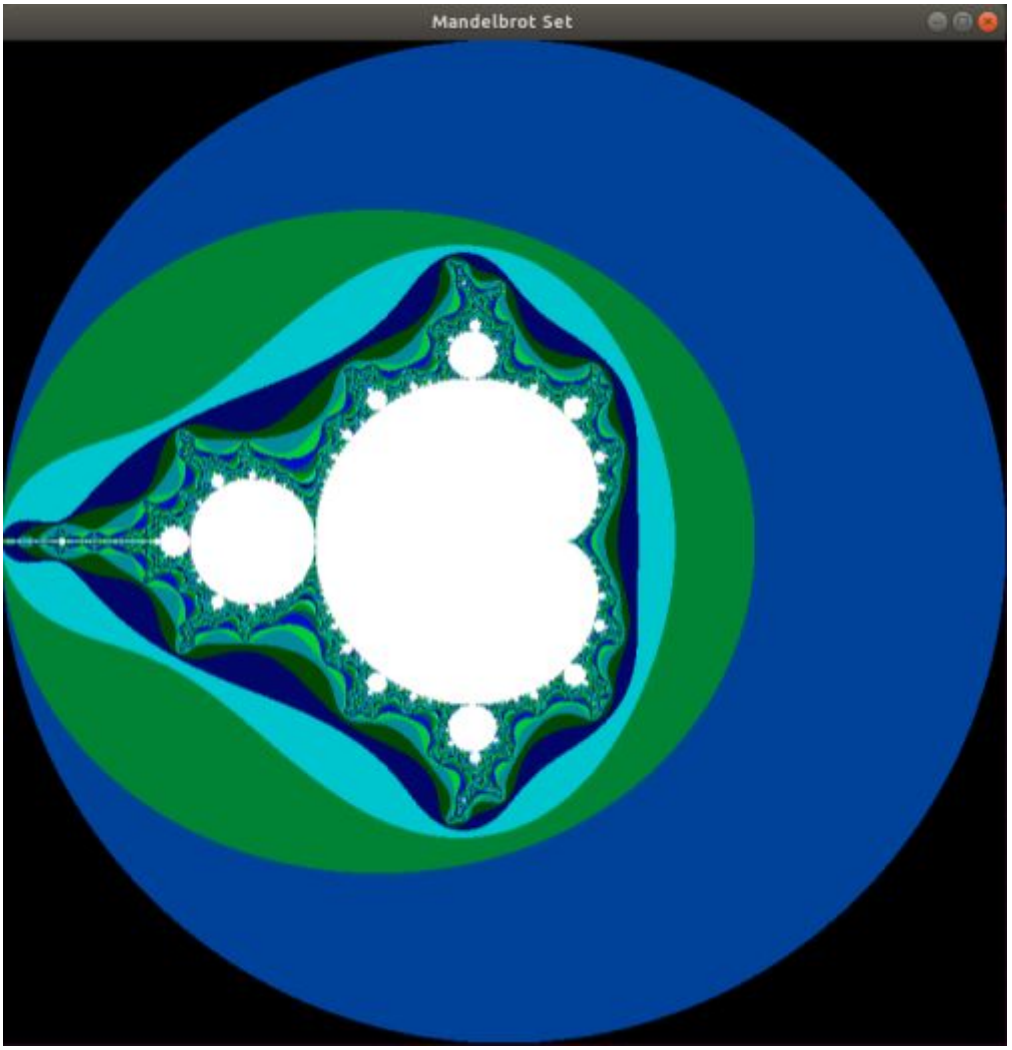


Image 1: The Mandelbrot set generated with the lab code.

In this session we have *mandel-tar.c* with the code of the *Mandelbrot set* and two executables: *mandel* and *mandeld*. The first one shows us in the command line (konsole) its execution time and the second one shows the fractal generated and displayed by the code.

Then our task is to modify this code to individually analyze the potential parallelism when tasks are defined at the two granularity levels: point or row. From image 2 to 4 we have used *Tareador* to obtain the task dependencies graph of the two versions.

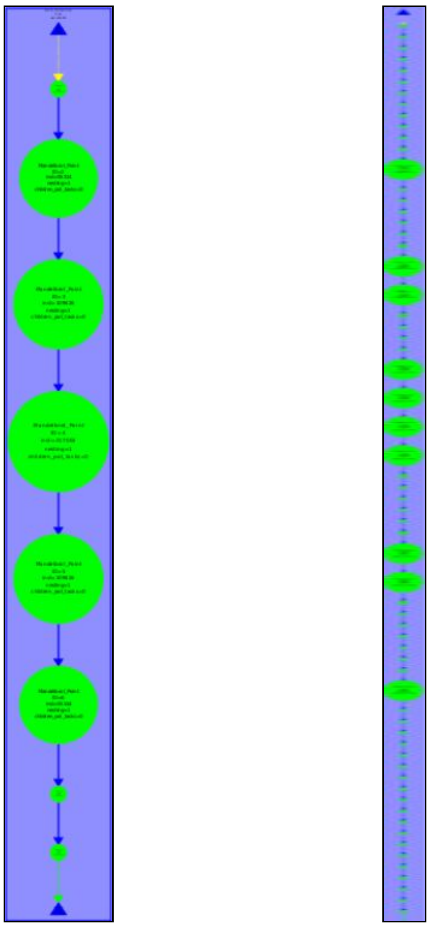


Image 2: *Mandelbrot set* display versions' task dependence graphs: row (left) and point (right).

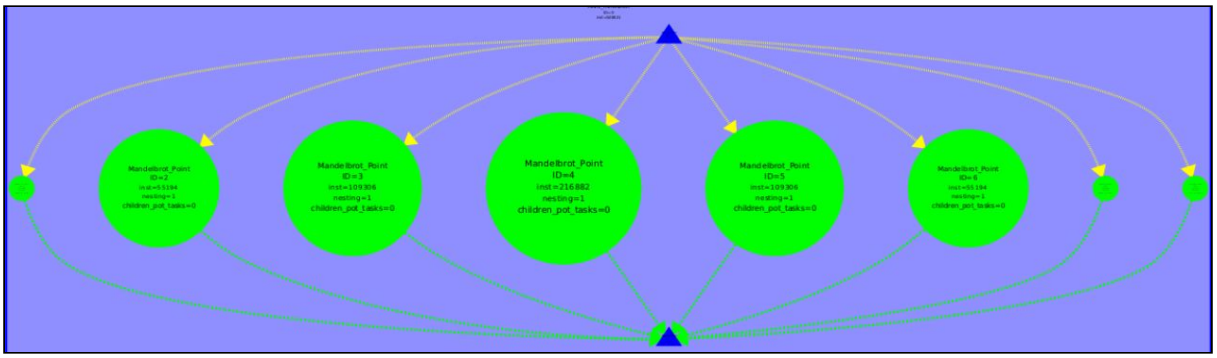


Image 3: Non-display row version task dependence graph.

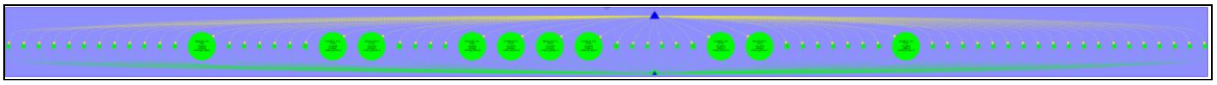


Image 4: Non-display point version task dependence graph.

The most important characteristics shown in the task dependence graphs for both *row* and *point* are that the point version have more tasks than the row version, which is logical as the point version have the *#pragma task* inside the innermost *for*, besides the row version have the *#pragma task* inside the outer *for*. Another characteristic is the balance between tasks which is not good because in both versions there are tasks that are bigger than others, an outstanding difference in the image 4.

There is a severe difference between the graphical and the non-graphical code provided in the lab session: the graphical one is serial even though we introduced the openMP directives to gain parallelism. The essential contrast is the code block inside the sentence “#if _DISPLAY_” programmed to produce the graphical version of the code.

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

Code 1: Focused code block in the *Mandelbrot Set* (*Tareador* version).

XSetForeground is used to set a pixel color in the screen and **XDrawPoint** displays a determined point in the screen. The lost parallel capability is due to the fact that the variable stored in memory where the color is saved is accessed afterwards to draw it successfully (creating a dependency).

In the past lab sessions we have learned OpenMP directives to prevent data race between different threads. In code 1, particularly the display region, is not protected against data races (several threads accessing the same memory space). To protect it, the chosen directive is “#pragma omp critical”. Once it is recompiled and executed, the different threads will access the memory sequentially and hence the data race will be avoided.

As mentioned, the point version creates a new task every iteration in the innermost loop, rather than creating them in the outer one as the row version, that is why there is a remarkable task creation overhead. Moreover, there is an important imbalance between the task size having some task to wait much time for the others to finish their work. Thus being said, the row version seems a better option in this case.

2. Point decomposition in *OpenMP*

In this part we will make different versions of the Mandelbrot set code using OpenMP directives such as *taskwait* or *taskgroup*. We will see the difference between them comparing their time and scalability plots. Before starting with the comparisons, we are going to briefly review some *Paraver* timelines from the original parallel code:

Image 5: OpenMP task execution and instantiation with 8 threads.

The code has been executed with 8 threads. In image 5, each green flag indicates a new task creation. The above timeline shows the task execution, and the below one shows de task instantiation (both are not necessarily parallel in the same time reference).

Image 6: Paraver trace timeline, original parallel code executed with 8 threads.

The more zoom we apply to the timeline, a better task granularity we achieve. In image 6, red is time waiting in the *critical* zone, yellow is a descendant task creation and blue is the proper pixel processing of the Mandelbrot Set. Thread number 5 is the task creator. There is a red dominance meaning threads spend most time waiting to enter the *critical* zone and do the pixel processing. Even though we avoid data races, we give up time execution.

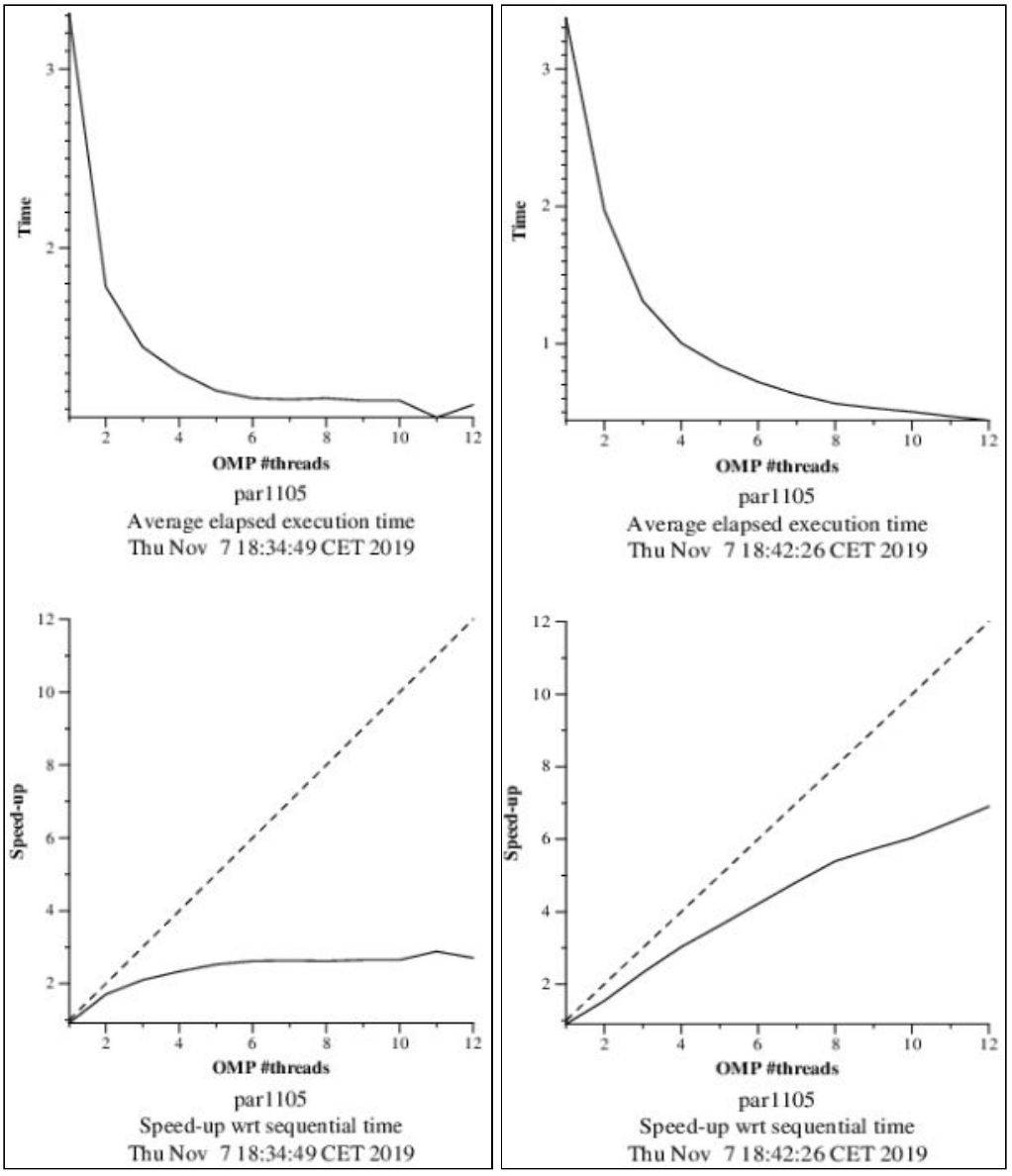
	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	85,504	220,800
THREAD 1.1.2	78,084	32,000
THREAD 1.1.3	78,568	16,800
THREAD 1.1.4	76,892	112,800
THREAD 1.1.5	81,881	38,400
THREAD 1.1.6	82,012	5,600
THREAD 1.1.7	72,293	30,400
THREAD 1.1.8	84,766	183,200
Total	640,000	640,000
Average	80,000	80,000
Maximum	85,504	220,800
Minimum	72,293	5,600
StDev	4,113.61	77,116.02
Avg/Max	0.94	0.36

Image 7: *Paraver* configuration file indicating the OpenMP task functions.

Image 7 shows the number of OpenMP task functions instantiated and executed by the different threads (8 threads in this particular case). There is a surprisingly imbalance in task instantiation/threads, but it is compensated as the number of executions of the task function is well balanced between threads, resulting in a total of 640.000.

Proceeding with the different task strategies and plot comparison, there is an important annotation: each code (representing a different strategy), included in the compressed deliverable, has been tested with the flag “-o” to compare the output of each code with the sequential code (the one we were given) to ensure correctness.

As we can see below in the different plots there is not much difference between the different versions besides the *taskloop* strategy where we can see an incredible upgrade. The benefit of this strategy is that it specifies that the iterations will be executed in parallel using OpenMP tasks



Images 8 and 9. Execution time along the incrementation of threads and scalability plots, using task (left) and taskloop (right) [point].

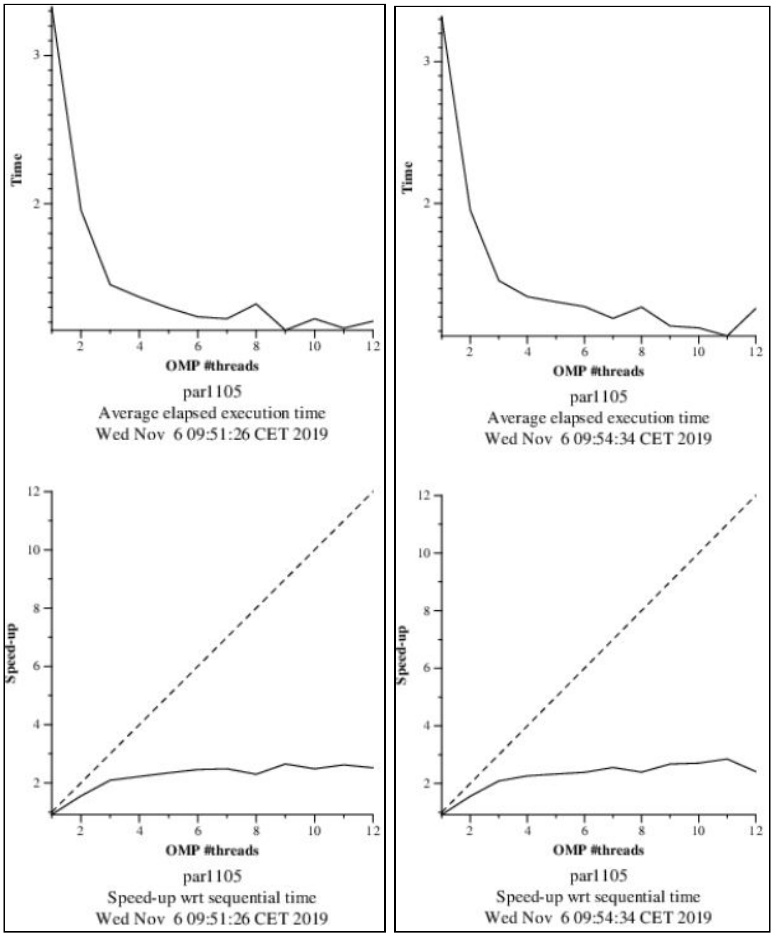
In addition, in this version we used the clause *num_tasks()* with the directive taskloop, so we test which number of tasks attribute fits our expectations (a great balance between the best execution time with less overhead).

Num_tasks()	1	2	5	10	25	50	100	200	400	800
Non-display version	0.39	0.285	0.269	0.238	0.10	0.139	0.15	0.28	0.309	0.239
Display version	0.565	3.29	4.07	3.7	3.28	3.60	2.37	2.35	3.28	3.73

Table 1: Comparison of execution time in both code versions changing the number of tasks [point].

Our conclusion is that 100 tasks is the version with the best time execution and overheads ratio, as we can see above at table 1.

Between task version and taskloop version there are the *taskgroup* version and *taskwait* version. The principal difference between them is that the *taskwait* only waits for the children of the original task (creator) to finish, but *taskgroup* waits for their children and his descendants, so while *taskgroup* wait for his descendants that create new children the taskwait version have probably already finished.



Images 10 and 11: Execution time along the incrementation of threads and scalability plots, using taskwait (left) and taskgroup (right) [point].

As there are not any real dependencies between tasks there is no need for synchronization mechanism (e.g. taskwait, taskgroup).

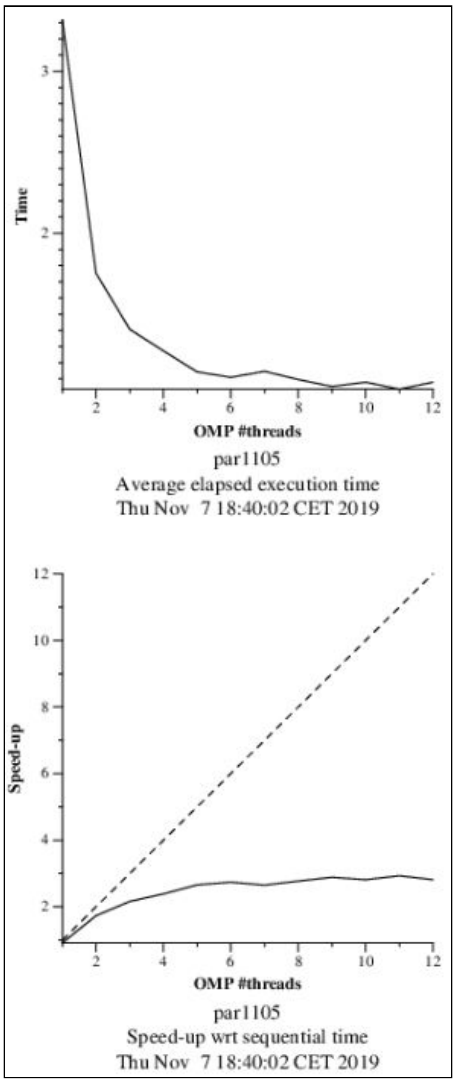
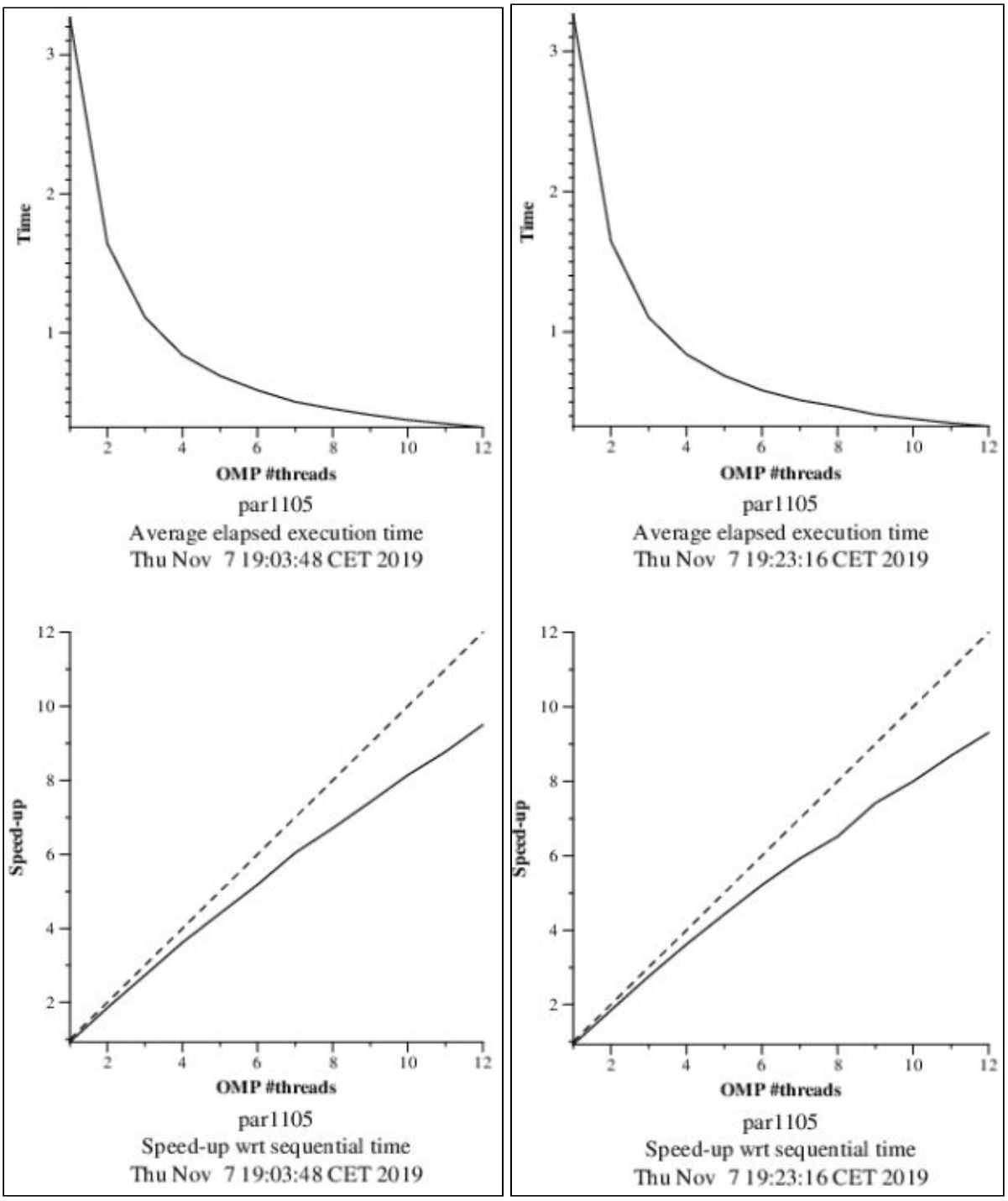


Image 12: Execution time along the incrementation of threads and scalability plots, using no synchronization mechanism [point].

The time seen in image 12 improves compared to the ones with synchro mechanism implemented. However, the scalability plot is pretty similar.

3. Row decomposition in *OpenMP*

As just mentioned, no synchronization is needed. Therefore, having seen little difference between the versions using it, we are going to analyze directly the task against the taskloop directives in this row version.



Images 13 and 14: Execution time along the incrementation of threads and scalability plots, using task (left) and taskloop (right) [row].

First of all, we can see a major improvement in both versions compared to the point versions that we commented before. Now, if we continue analysing the task and taskloop versions (row version) we can see that unlike the point strategy, on this one the task version has a slightly better speed-up than the taskloop, the reason of this is due to the overheads. But besides that, we consider that both versions have a very good scalability that is closer than the point strategy to the ideal one (the discontinuous line in the plot).

In conclusion, when comparing the two seen approaches in the Mandelbrot Set parallel code, we see that the row version is by far a better approach mainly because of the difference of creation overhead and work balance.

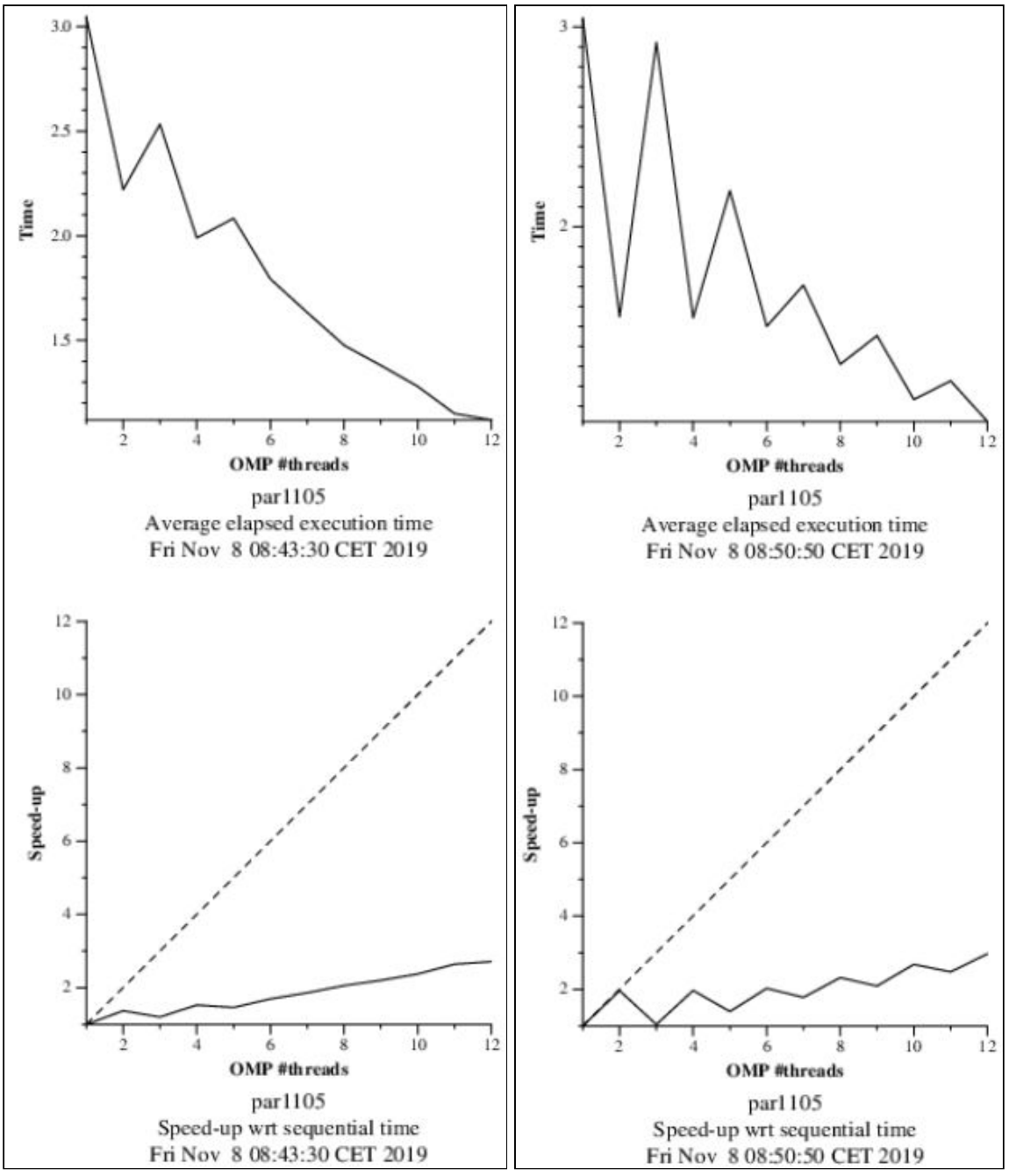
4. For-based parallelization (optional)

Last but not least, as an optional exercise, we change the parallel approach to test de *Mandelbrot Set* with a for-based parallelization strategy. We will use *#pragma omp parallel for schedule(x)* with an x that can be static, dynamic or guided and compare which version produces a better result.

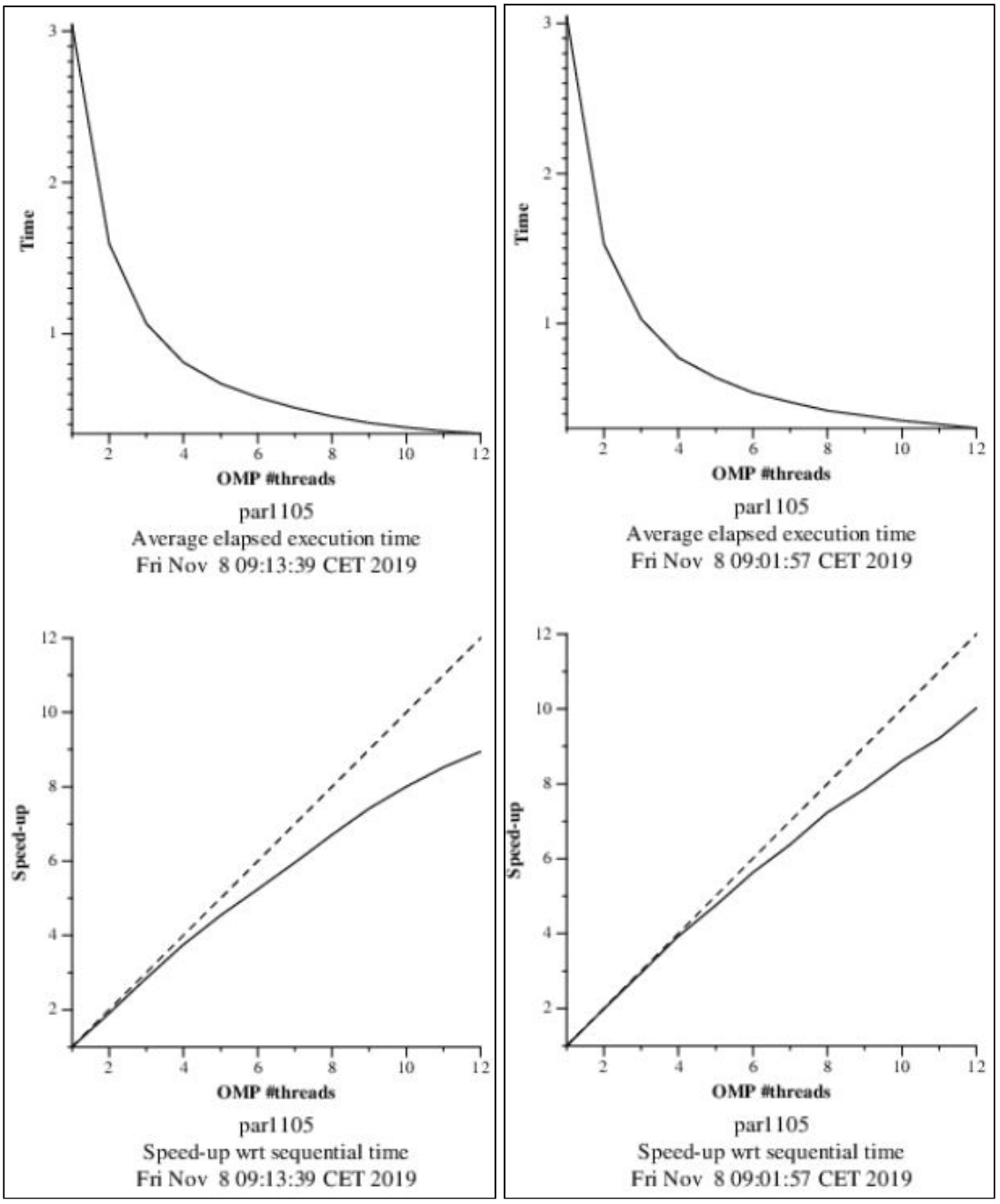
The difference between the aforementioned is the following: static distributes the iteration space in chunks of size N/num_threads (**default**) in compilation time, in dynamic threads dynamically grab chunks of N iterations until all iterations have been executed and finally, in guided, the size of the chunks decreases as the threads grab iterations. In our particular case we tested the schedules dynamic and guided with the **default** value (N = 1). Both distribute the iteration space in execution time.

*Peaks are produced by some processors which begin a turbo mode ordered by the server (boada).

**The difference in the code implementations between row and point versions is the collocation of the directive parallel + the clause *for* and so on.



Images 15 and 16: Execution time along the incrementation of threads and scalability plots, using static strategy [point] (left) and [row](right) .



Images 17 and 18: Execution time along the incrementation of threads and scalability plots, using dynamic strategy [point] (left) and [row](right) .

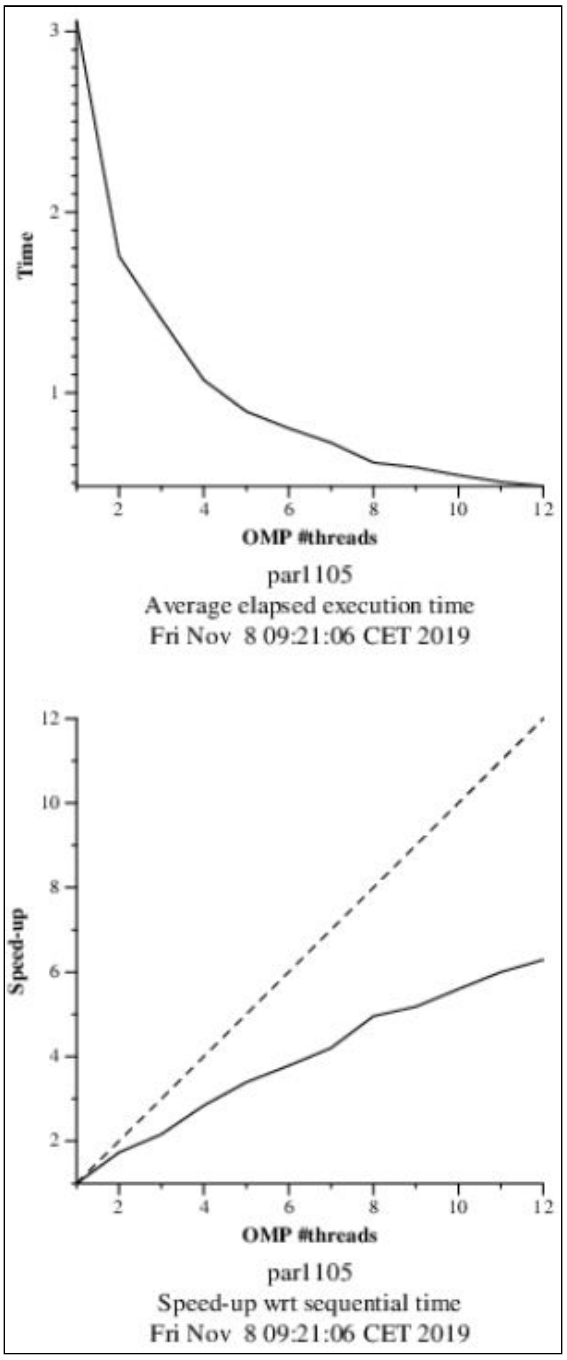


Image 19: Execution time along the incrementation of threads and scalability plots, using guided-row strategy .

In the plots above, at first instance we can see that the *static* strategy it is a fatal idea because the plot shows us some irregularities in the execution time and even in the row version the speed-up it is even worse, probably it has some load imbalance problems.

Then, we have the *dynamic* versions where there is an incredible upgrade compared to the previous version, this is probably the condition that this strategy does not need

to wait for another process to finish executing their work and makes the code faster in execution, and it solves the load imbalance problems.

Between the point and the row versions with the average elapsed execution time we cannot make a distinction but in the scalability plot there is a difference where the row version is better than the point version.

Finally, in the *guided* version we show only the row strategy because it is similar to the dynamic version (using the default value) and we know that in dynamic the row was better so we only wanted to compare it with the previous plot and as we can see clearly, in this case, dynamic is better than guided.

To conclude with, there are some advantages using for-based parallelization against task-based such as the workload distribution, but there are also disadvantages, for example, a less flexible approach and, especially, heavier overheads.