PAR Laboratory Assignment

Lab3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Sergio Araguás Villas - par4107
David Pérez Barroso - par4121
08-11-2022
22-23 Q1

Índice

Session 1: Task decomposition analysis for the Mandelbrot set computation

The Mandelbrot set

The objective of this laboratory session was to explore the tasking model in *OpenMP* to express iterative task decomposition. For that, firstly we've been introduced by a program that is used to execute the computation of the *Mandelbrot set* which generates a distinctive two dimensional fractal shape as the Fig1. below.

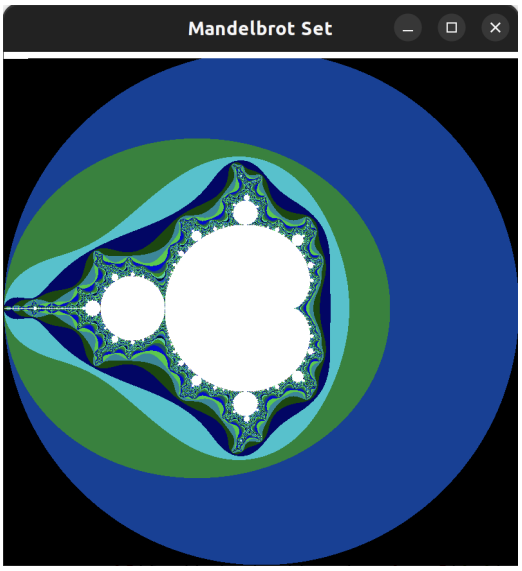


Fig.1. View of Mandelbrot set

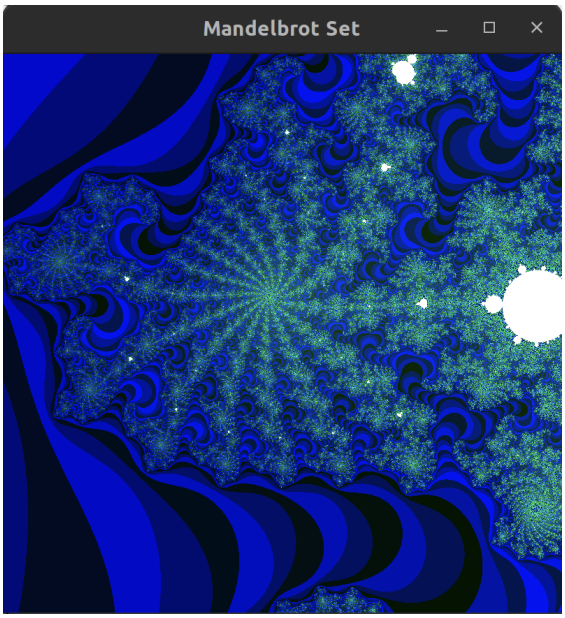


Fig.2. Different view of Mandelbrot set

We executed the program with different options, obtaining different results. With the option **-d** we obtain Fig.1. image, with option **-h** the histogram for the values in the Mandelbrot set was also computed, and with option **-o** the values of the Mandelbrot set were written to the disk.

Using the option **-c** with values -0.737, 0.207; option **-s** with value 0.01 and the option **-i** with value 100000 we obtain a different view of the Mandelbrot set as it can be observed at Fig.2.

Task decomposition analysis with Tareador

In this part we will study the two decomposition strategies that were proposed at the last section (Row and Point), using *Tareador* as a tool to obtain the dependency graphs of the two strategies.
For this, we will analyze and modify the *mandel-tar.c* file in order to execute the strategies commented.

Row strategy

Firstly, we will analyze the parallelism with a row decomposition strategy. In order to do that we will set tasks to be as long as an entire row execution. We can do that by positioning *tareador_start_task* instruction at the beginning of the *row* loop and the *tareador_end_task* instruction at the end of the row loop.
In color orange, at the figure 3, it can be observed the modified part of the *mandal-tar.c* code.

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

    // Calculate points and generate appropriate output
    for (int row = 0; row < height; ++row) {
        #tareador_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");
    }
}
```

Fig.3. Code of mandal-tar.c file with row strategy

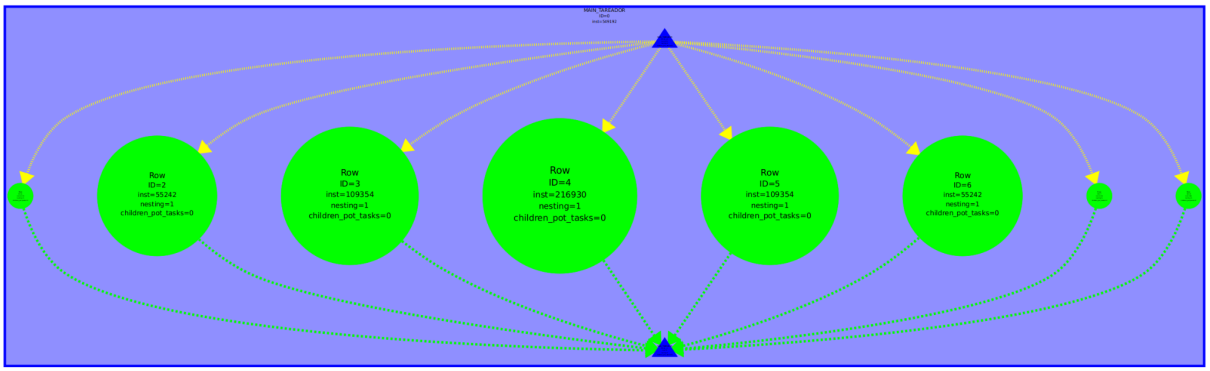


Fig.4. Row task decomposition strategy

Looking at the dependency graph (Fig.4.), one characteristic that we can observe is that there isn't any dependency between the tasks created (green nodes). Another characteristic that we can obtain by observing the graph, is that the work isn't equally distributed which isn´t optimal for parallel execution.

Now we execute *mandel-tar* again, but at this time, with the option -d, that displays the *Mandelbrot* set, and with *Tareador* we obtain the dependence graph that you can see at Fig.5.

As we can see on the graph, the work is once again unequally distributed, we obtain again tasks with heavy different sizes, by the other hand, the tasks now have a dependence with a previous one, creating a sequence of tasks that breaks the parallelism.

But, Why is there a big difference between these executions? Well, we found the part of the code that is causing this problem at the "*if (output2display)*" statement. Inside, we found the XSetForeground and XDrawPoint functions, as can be seen at the figure Fig.6; which set a color of a pixel and display it on the screen.

Fig.5. Row strategy with -d option

```
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.6. if statement option -d

We could use *#pragma omp critical* before the functions *XSetForeground* and *XDrawPoint* for protecting this part of the code, giving that there could be a data race between threads because they access the same variable that stores color multiples times.

The last step is to execute the *mandal-tar* but with the option *-h*, obtaining an histogram and its respectively dependence graph.

If we take a look at the dependency graph, we note the existence of three chains of tasks, due to the line of code *if (output2histogram) histogram[k-1]++*; that executes the code for generating the histogram and makes threads wait because they can't access elements that don't exist in that position.

The modification of variables at this part could lead to dataraces so we could use *#pragma omp atomic* to force access by only one thread.

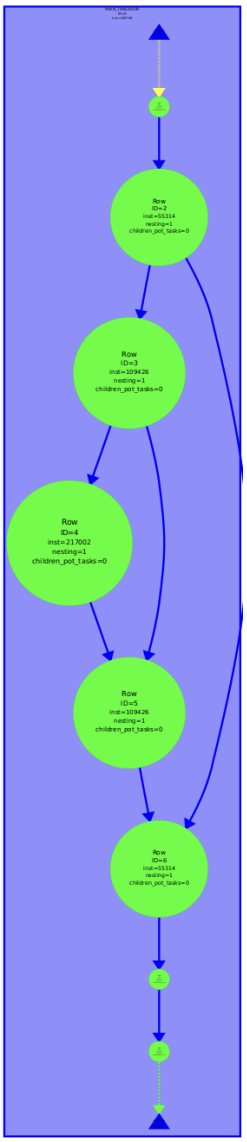


Fig.7. Row strategy with -h option

Point strategy

Now we will do the same analysis but following the point strategy.
For following this strategy we will use the *Tareador* to delimitate the task execution inside the column loop as seen in the following code.

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

    // Calculate points and generate appropriate output
    for (int row = 0; row < height; ++row) {
        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");
        }
    }
}
```

Fig.8. Code of mandal-tar.c file with point strategy

If we take a look at the dependency graph of the code with point strategy (Fig.9.), we note that the number of tasks has increased due to more granularity, but we still see an unbalance in the tasks's workload distribution. However this unbalance is more manageable than the previous due to being more distributed. This leaves us the conclusion that with enough threads this code would execute faster.

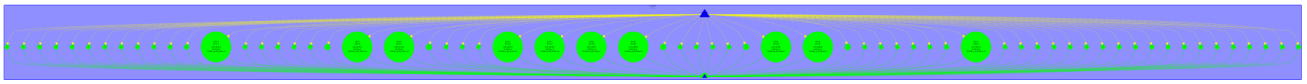
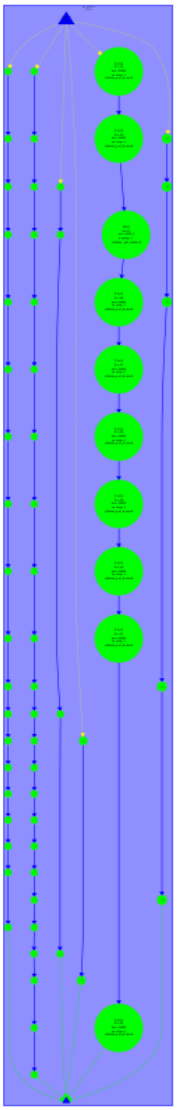


Fig.9. Point decomposition strategy

Executing the same code but with the option **-d**, we obtain a sequential dependence graph (Fig.10.) with more tasks due to the granularity mentioned above and being sequential for the same reasons we explained in the row strategy case, with also being eligible for the same solutions.



Finally executing the code with the option **-h**, we see on the corresponding graph (Fig.11.) different chains of tasks in parallel in contrast to the previous strategy. Nevertheless the tasks with the most workload are executed in sequential order, making the other chains of tasks wait for them to finish, which is something not ideal for parallelization. The serialization again is caused by the same reason we mentioned in the row strategy case.

Having analyzed both strategies, we come to the conclusion that the main difference between them is the number of tasks generated, given that the point strategy targets a more concrete piece of code.

We think that the point strategy is optimal when the workload of the code can be distributed between many tasks so they can be executed in parallel.

The row strategy in contrast, is better as an alternative when the workload can't be divided as it will reduce the overheads created because of the tasks synchronization

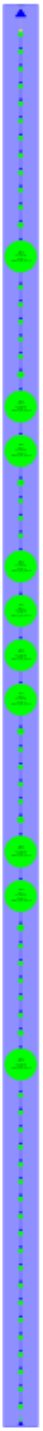


Fig.11. Point strategy with -h option

Fig.10. Point strategy with -d option

Session 2: Implementation and analysis of task decompositions in OpenMP

In this session we explored different options using the OpenMP tasking model to express the iterative task decomposition strategies that we saw in the previous laboratory session. Also, we have analyzed the scalability and behavior of our code.

Point decomposition strategy

Firstly, in this section, we have modified the code of the file *mandel-omp.c*, that has implemented a point strategy. As it can be seen at Fig.12. we add the directives that resolve some of the dependencies of the program with the omp instructions discussed in the last session (*atomic* and *critical*).

```
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.12. Modified code protecting dependencies

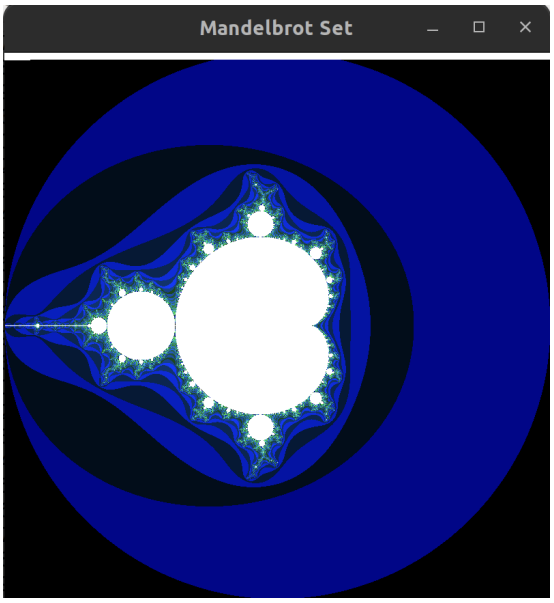


Fig.13. Mandelbrot set obtained from the execution with 1 and 2 threads

Once the code is modified, we execute its binary with 1 thread using OMP_NUM_THREADS=1 *./mandel-omp -d -h -i* 10000. As a result, we obtain the same image (Fig.13.) as in the beginning of the laboratory but with changes in color due to the *atomic* directive

Following that, we compile again the binary but with 2 threads using THREADS=2 *./mandel-omp -d -h -i* 10000 obtaining the same image as before.

We didn't see a visual difference between these two executions, but if we look at their execution time, that thing changes. In the following paragraph we will see the difference between the execution time of a sequential version, the execution time when it is executed with one thread and the execution time when it is executed with 8 threads.

Sequential execution
Total execution time (in seconds): 2.608068

Execution time with one thread
Total execution time (in seconds): 2.966925

Execution time with eight threads
Total execution time (in seconds): 1.494083

As it can be seen in the times, the execution time with eight threads is clearly faster than the sequential execution. But why is the execution time of one thread slightly slower? That is because the overhead time is added to the total execution time causing that extra time.

We can see these conclusions in the graphs of speed-up and time relation with threads from one to twenty (Fig.14. and Fig.15.).

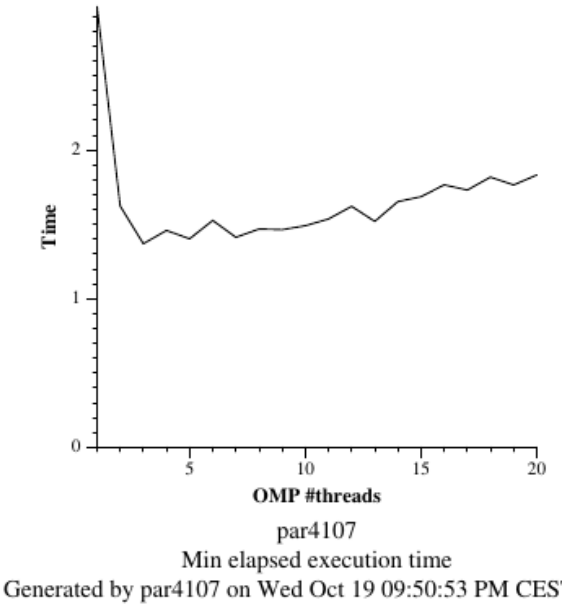


Fig.14. Time/threads graph

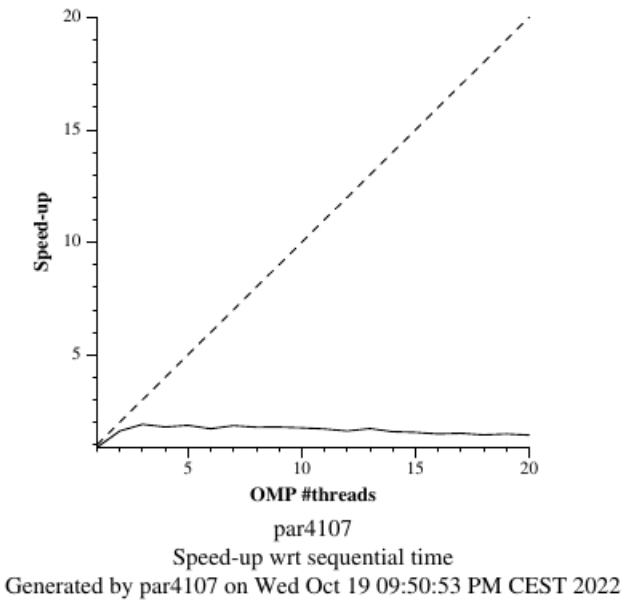


Fig.15. Speed-up/threads graph

We can observe in Fig.14. how time harshly decreases to its lowest value with three threads and then progressively increases due to the overheads, even though time is significantly less than its initial value.

On the other hand, the speedup performs badly with new threads added, leaving us with the conclusion that the scalability is far from optimal.

To find out more about the results obtained, we analyze with *modelfactors* a smaller Mandelbrot set. We can see in Fig.16. that the speed up decreases from 8 to 16 threads confirming its low scalability with new threads added.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.36	0.32	0.33	0.34
Speedup	1.00	1.61	1.82	1.77	1.71
Efficiency	1.00	0.40	0.23	0.15	0.11

Table 1: Analysis done on Sat Oct 22 05:39:38 PM CEST 2022, par4121

Fig.16. Table with program metrics

In the next tables, we can see that the number of tasks executed is independent of the number of threads used and that the average execution time of these explicit tasks seems to be small but the overhead associated with them tends to be high.
That causes the low execution efficiency that can be seen in the second table.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	1	4	8	12	16
Global efficiency	95.28%	38.23%	21.71%	14.09%	10.19%
Parallelization strategy efficiency	95.28%	46.15%	29.23%	20.38%	15.13%
Load balancing	100.00%	90.82%	54.59%	35.06%	25.22%
In execution efficiency	95.28%	50.81%	53.54%	58.13%	60.01%
Scalability for computation tasks	100.00%	82.84%	74.27%	69.15%	67.36%
IPC scalability	100.00%	80.07%	74.22%	71.46%	69.67%
Instruction scalability	100.00%	103.80%	104.40%	104.17%	104.17%
Frequency scalability	100.00%	99.67%	95.86%	92.89%	92.83%

Table 2: Analysis done on Sat Oct 22 05:39:38 PM CEST 2022, par4121

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	0.82	0.84	0.87
LB (time executing explicit tasks)	1.0	0.89	0.89	0.9	0.89
Time per explicit task (average us)	4.91	5.68	5.93	6.06	6.05
Overhead per explicit task (synch %)	0.0	103.74	273.76	483.32	723.72
Overhead per explicit task (sched %)	5.46	30.7	24.48	22.16	21.97
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Sat Oct 22 05:39:38 PM CEST 2022, par4121

Fig.17. Tables of efficiency and explicit tasks in parallel fraction

In the next step, we analyzed how the execution is working, for that we observe the execution with twelve threads using *Paraver* obtaining some conclusions. The following tables show a profile of explicit tasks of the execution:

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	102,400
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
THREAD 1.1.9	-
THREAD 1.1.10	-
THREAD 1.1.11	-
THREAD 1.1.12	-
Total	102,400
Average	102,400
Maximum	102,400
Minimum	102,400

Fig.18. Histogram of explicit task creation

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	215
THREAD 1.1.2	9,716
THREAD 1.1.3	8,611
THREAD 1.1.4	10,105
THREAD 1.1.5	8,777
THREAD 1.1.6	10,000
THREAD 1.1.7	8,724
THREAD 1.1.8	10,092
THREAD 1.1.9	8,509
THREAD 1.1.10	9,851
THREAD 1.1.11	8,175
THREAD 1.1.12	9,625
Total	102,400
Average	8,533.33
Maximum	10,105
Minimum	215
StDev	2,594.91

Fig.19. Histogram of explicit task execution

We clearly see at Fig.18. that the creation of tasks is exclusive to thread one, that created all the tasks of the execution. At Fig.19. we can observe the distribution of tasks per thread. The tasks executed in thread one are significantly lower because it's the one that creates the tasks and the other threads have nearly the same workload.

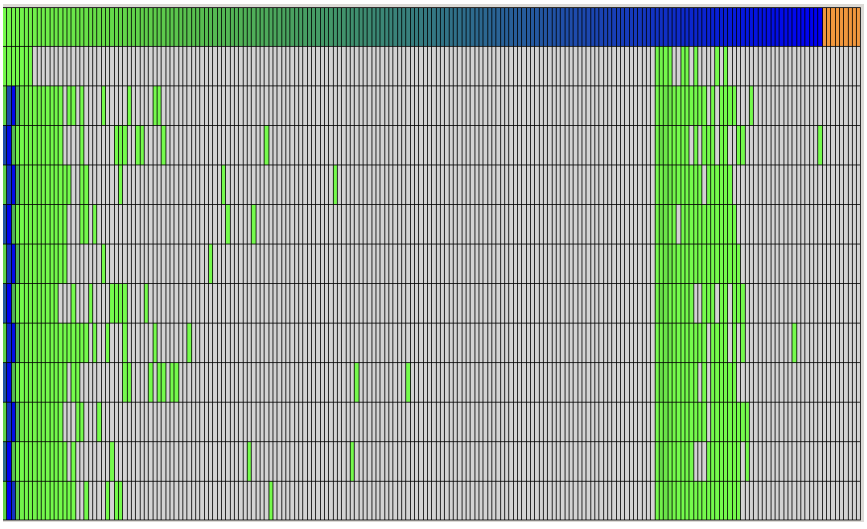


Fig.18. Histogram of explicit task execution

If we take a look at the histogram (Fig.20.) we can observe the duration of each explicit task per thread. We note that the tasks are created in certain margins of time, that is because the tasks are being created at the beginning of every row loop. (That creates the "green bars" that the Fig.20. show).

Taking into account that there are large intervals of time with no explicit task being executed (Fig.20) and that the scalability shown in Fig.16. decreases adding more than 4 threads due to overheads, we can conclude that the granularity of the tasks in this strategy is not appropriate.

Point strategy with optimization: granularity control using taskloop

In this section we modify the code of the file mandel-omp.c using the directive *taskloop*.

```
#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;
```

Fig.19. Code version with taskloop

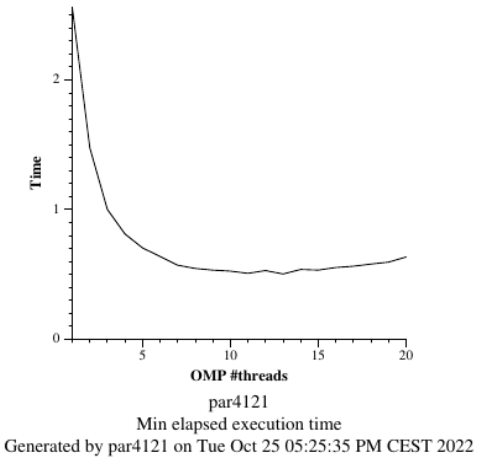
We executed this version of the program with one and with eight threads and we obtained the following times, observing that are sensitively smaller:

Execution time with one thread:
Total execution time (in seconds): 2.562445

Execution time with eight threads:
Total execution time (in seconds): 0.544543

Executing *submit-strong-omp.sh* and *submit-strong-extrae.sh* with *sbatch* and *mandel-omp* as a parameter, we obtain the scalability plot and the whole program execution metrics.

Observing the first table and the first plot of scalability, we note that the performance of this program is better than the version based on *task*, due to the average elapsed time of this version being better than the other, we can say that it is more scalable.



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	2.92	3.14	2.66	2.09
Efficiency	1.00	0.73	0.39	0.22	0.13

Table 1: Analysis done on Tue Oct 25 05:16:05 PM CEST 2022, par4107

Fig.20. Plot and modelfactor analisis

Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.57%	72.78%	39.16%	22.13%	13.03%
Parallelization strategy efficiency	99.57%	76.46%	44.46%	26.14%	15.66%
Load balancing	100.00%	95.54%	95.61%	95.00%	93.06%
In execution efficiency	99.57%	80.04%	46.50%	27.52%	16.83%
Scalability for computation tasks	100.00%	95.19%	88.09%	84.64%	83.22%
IPC scalability	100.00%	97.37%	97.20%	96.77%	96.24%
Instruction scalability	100.00%	99.43%	98.67%	97.92%	97.19%
Frequency scalability	100.00%	98.32%	91.85%	89.33%	88.98%

Table 2: Analysis done on Tue Oct 25 05:16:05 PM CEST 2022, par4107

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)	128.67	33.8	18.26	12.67	9.66
Overhead per explicit task (synch %)	0.07	26.99	110.99	259.55	505.01
Overhead per explicit task (sched %)	0.36	3.8	13.98	23.12	33.85
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Tue Oct 25 05:16:05 PM CEST 2022, par4107

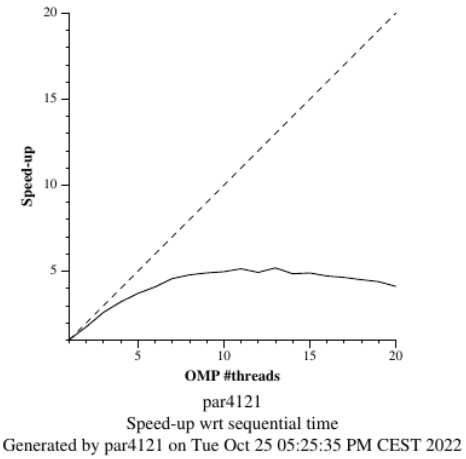


Fig.21. Plot and modelfactor analisis

If we take a look at the second table of *modelfactors*, we observe that the Parallelization strategy efficiency has slightly improved with respect to the last version (Fig.17.) from 1 to 12 threads but equals its performance in 16. Load balancing is the key factor to that change as it maintains its performance with many threads.

The number of explicit tasks generated is much lower than the other version, but in the taskloop version it increases in function of the number of threads as opposed to the previous where the tasks are constant. On the other hand, synchronization and scheduling overheads are less noticeable than the last version, as there aren't as many tasks that need to synchronize.

We think that the granularity is better than the last version as it improves many metrics because it distributes tasks more efficiently. (number of task per taskloop)

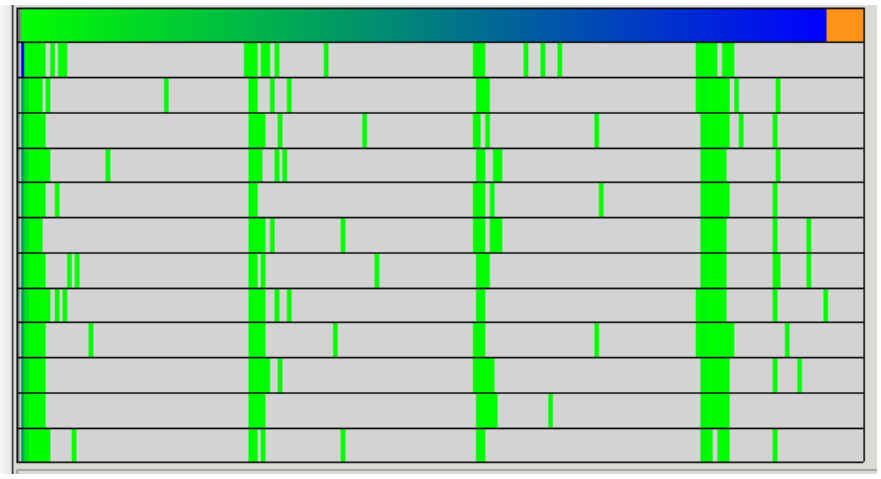


Fig.22. Histogram - Duration of explicit tasks

If we compare this histogram (Fig.22.) with the previous one (Fig.18.), we note that each execution of the row loop generates a group of tasks more frequently due to the taskloop directive assigning all the column loop iterations to a task group.

Now if we take a look at (Fig.23.) we can see that thread 12 is the thread dedicated to the creation of all explicit tasks, that makes the thread execute much less task than the others, being this situation a great contributor to load unbalance.

This can be seen at (Fig.24.), where we can observe the timeline of the execution of the explicit tasks that each thread executes. We observe that the load of the execution of tasks is similar in the threads one to eleven, but as we commented before, the load in thread twelve is unbalanced due to the tasks creation.

	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	-
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
THREAD 1.1.9	-
THREAD 1.1.10	-
THREAD 1.1.11	-
THREAD 1.1.12	320
Total	320
Average	320
Maximum	320
Minimum	320
StDev	0
Avg/Max	1

Fig.23. Histogram of explicit task creation

The other timeline (task function creation) confirms what we said about the load balance and tasks creation.

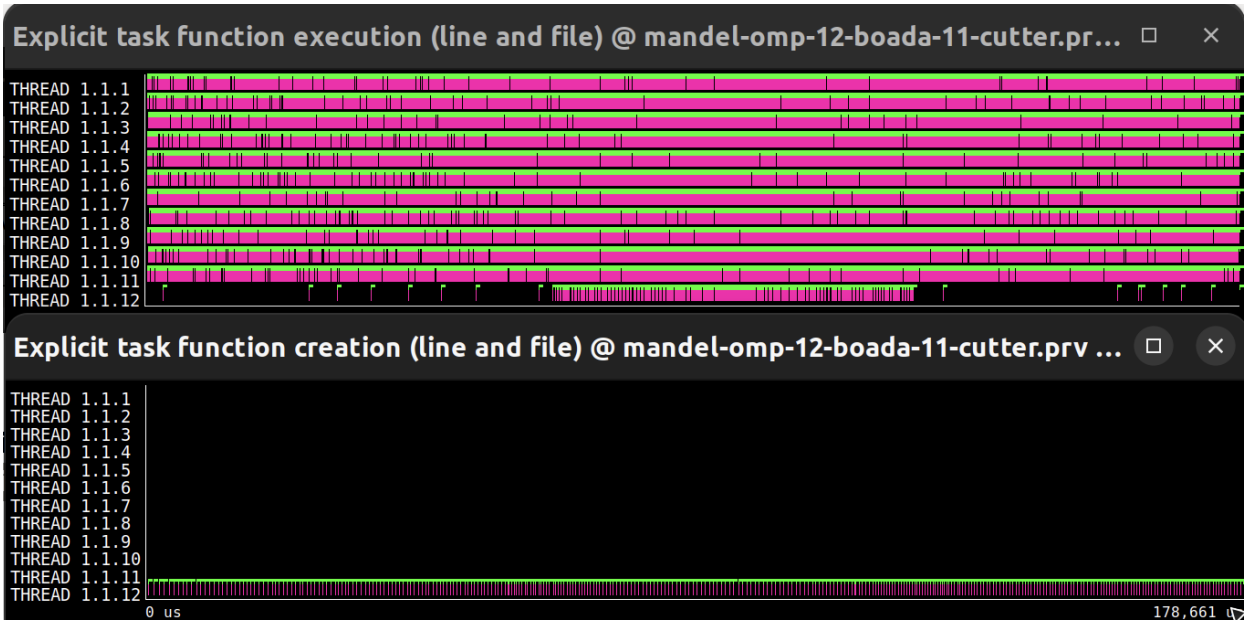


Fig.24. Timelines of explicit tasks execution and creation

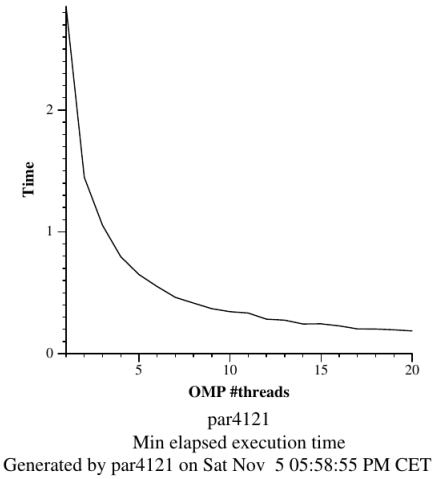
Row decomposition strategy

In this section we modify the code of mandel-omp.c changing the *#pragma omp taskloop* to affect the row loop to execute the row strategy.

```
// 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;
```

Fig.25. Code with row strategy

We can observe that the relation between time and threads (Fig.26.) is way more strong in this version, with a notable decrease in time with more threads and maintaining the efficiency with more threads better than the point strategy .

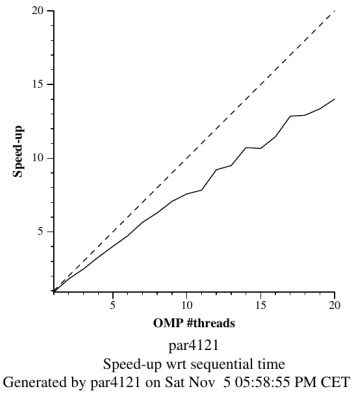


Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.07	0.05	0.04
Speedup	1.00	3.50	6.62	9.46	12.29
Efficiency	1.00	0.88	0.83	0.79	0.77

Table 1: Analysis done on Sat Nov 5 05:47:27 PM CET 2022, par4121

Fig.26. Plot and Modelfactor analysis

We also can see that the speed up grows at a greater pace than in the point strategy (Fig.27), and that the load balance is up to 50% better with 16 threads than the previous versión. That's due to having row level tasks with less dependence on other tasks, and hence, with less overheads.



Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.98%	87.57%	83.08%	79.24%	77.34%
Parallelization strategy efficiency	99.98%	89.87%	92.64%	90.65%	88.93%
Load balancing	100.00%	89.93%	92.82%	90.89%	89.39%
In execution efficiency	99.98%	99.94%	99.81%	99.74%	99.48%
Scalability for computation tasks	100.00%	97.44%	89.67%	87.41%	86.97%
IPC scalability	100.00%	98.61%	97.38%	96.64%	96.14%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	98.81%	92.09%	90.46%	90.47%

Table 2: Analysis done on Sat Nov 5 05:47:27 PM CET 2022, par4121

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)	45685.57	11721.19	6367.48	4354.72	3281.89
Overhead per explicit task (synch %)	0.0	11.24	7.89	10.24	12.32
Overhead per explicit task (sched %)	0.01	0.02	0.04	0.05	0.08
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Sat Nov 5 05:47:27 PM CET 2022, par4121

Fig.27. Plot and Modelfactor analysis

If we take a look at the histogram of explicit task execution and creation (Fig.28.), we observe that using the Row strategy has much less number of tasks than the Point strategy. That is because now the taskloop instruction is outside the row loop causing a less total number of tasks than in Point Strategy.

Now thread six is the creator of these tasks and the rest of threads have between five or seven tasks causing a slight load unbalance.

	102 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	6
THREAD 1.1.2	7
THREAD 1.1.3	5
THREAD 1.1.4	7
THREAD 1.1.5	6
THREAD 1.1.6	53
THREAD 1.1.7	6
THREAD 1.1.8	6
THREAD 1.1.9	6
THREAD 1.1.10	6
THREAD 1.1.11	6
THREAD 1.1.12	6
Total	120
Average	10
Maximum	53
Minimum	5
StDev	12.97
Avg/Max	0.19

	102 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	-
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	1
THREAD 1.1.7	-
THREAD 1.1.8	-
THREAD 1.1.9	-
THREAD 1.1.10	-
THREAD 1.1.11	-
THREAD 1.1.12	-
Total	1
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Fig.28. Histograms of creation and execution of explicit tasks

Optional: task granularity tune

In order to obtain the plots that shows the granularity of both Point Strategy and Row Strategy to study their behavior with different task granularities, we modified the *mandel-omp.c* file by adding the *num_task* clause with *user_param* as a parameter next to the *taskloop clause*: ***#pragma omp taskloop num_tasks(user_param)***. The *user_param* is the number of tasks executed in every taskloop.

Once done, we executed it with the *script submit-numtasks-omp.sh* with batch.

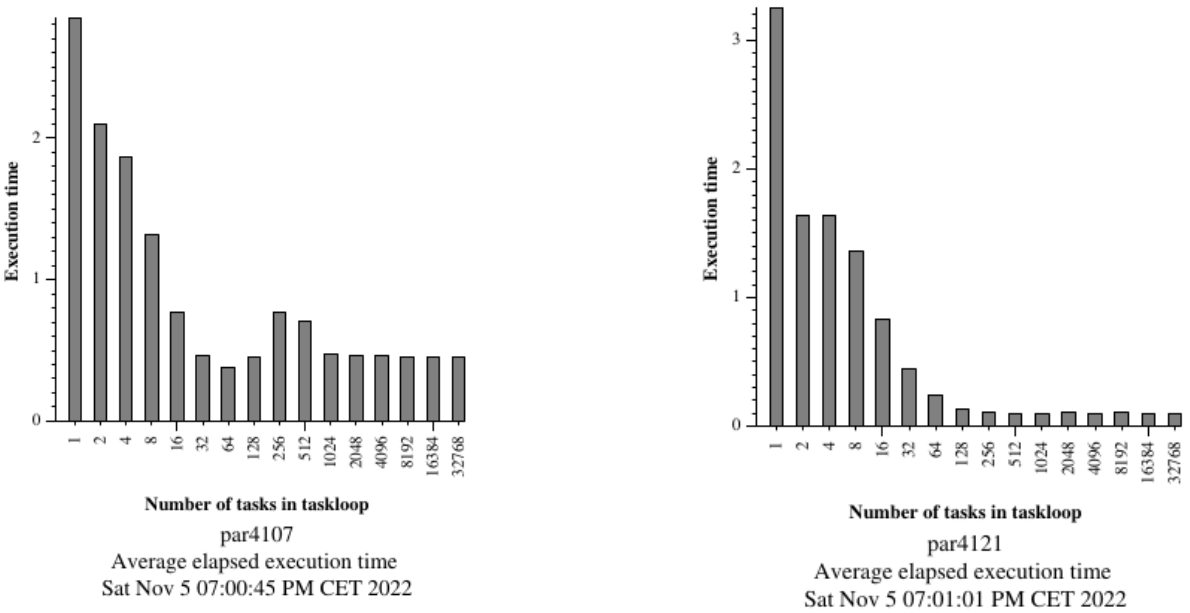


Fig.29. Plots of tasks per thread in Point Strategy (left) and Row strategy (right)

In these plots (Fig.29.) we can conclude that both strategies decrease execution time as we add more threads. But if we look closely, we see that the Row strategy decreases the execution time faster than in Point Strategy, and reaches smaller execution times, making it a better option.