UNIVERSITAT POLITÈCNICA DE CATALUNYA

Intel·ligència Artificial

Deliverable3 - Mandelbrot set

Analysis of task
decompositions in OpenMP

Autores:
Guo Haobin
Jin Haonan

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

April Q2 2023-2024

INDEX

1.Introduction

During the lab session, we were taught on the tasking model in OpenMP which is utilized to present iterative task decompositions. The assignments we worked on in this session were focused on computing the Mandelbrot set. This set is a representation of a group of points in the complex domain, and its boundary produces a well-known two-dimensional fractal shape.

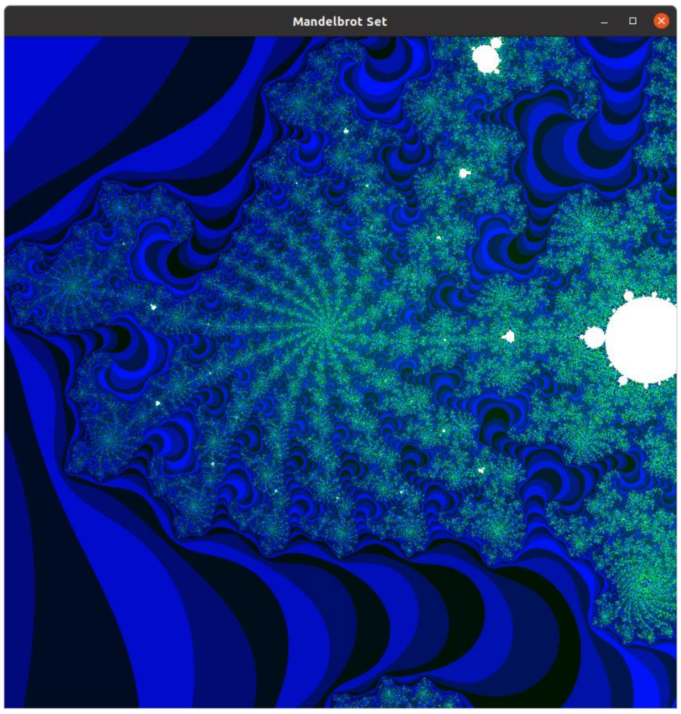


Figure 1: Execution of mandel-seq -d -c -0.737 0.207 -s 0.01 -i 100000

For each point c in a delimited two-dimensional space, the algorithm of Mandelbrot set it consist of a complex quadratic polynomial recurrence $z_{n+1} = {z^2}_n + c$ which is iteratively applied n to determine if it belongs or not to the Mandelbrot set.

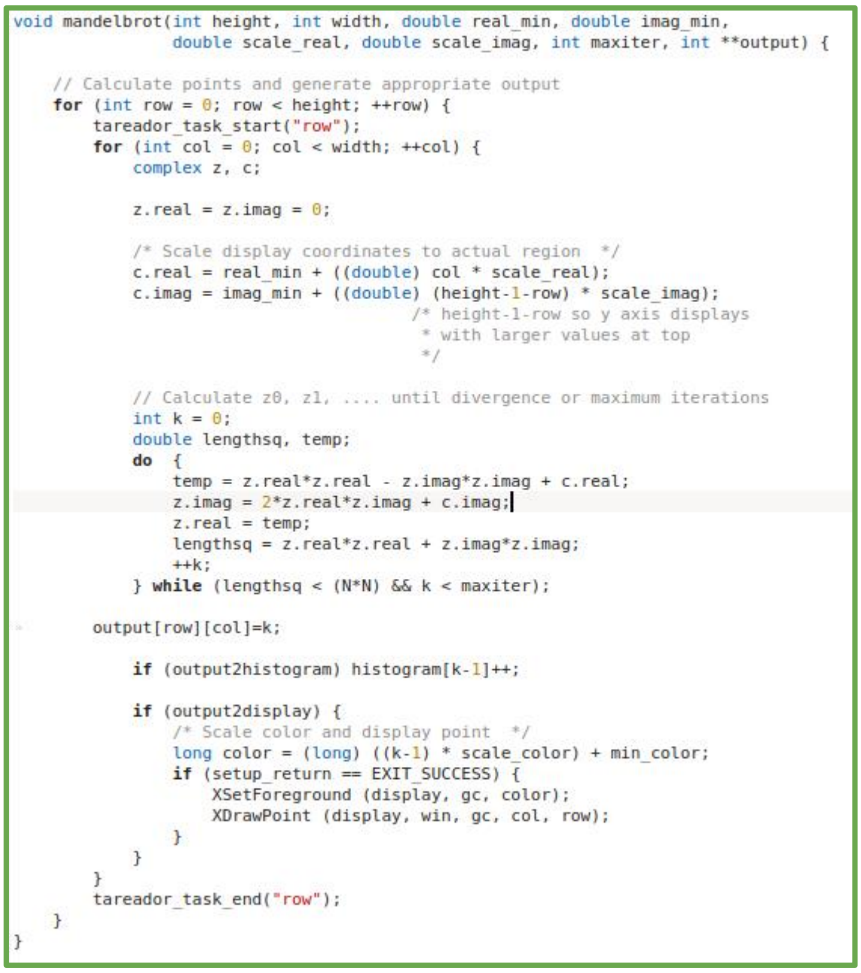
Before we start with the exercises, let's take a look at the performance of the mandel-seq.c file, along with the various output choices the program can be run with. These options include: -h, which computes the histogram for the values within the Mandelbrot set along with the program; -d, which displays the Mandelbrot set (as exemplified in figure 1); -o, which writes the set and/or histogram values to the disk for comparison with the reference output (this option is often implicit in the execution of most of the given files)

2.Task decomposition analysis with Tareador

This section will cover an analysis of the key features of each proposed task decomposition approach (Row and Point) with the help of Tareador, wherein both approaches will be tested with a granularity of one iteration per task. For each one with the help of the Tareador, we will generate a TDP so we can study their characteristics in order to get any conclusions. (scripts needed: mandel-tar.c, run-tareador.sh)

2.1 Row strategy

Our initial focus is on examining the parallelism possibilities of the Row strategy. To accomplish this, we edited the code by introducing tasks within the row loop and using tareador_task_start("ROW") and tareador_task_end("ROW") to denote their start and end points. The resulting code is presented below:



```c
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_task_start("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_task_end("row");
    }
}
```

Figure 2: Implementation of row strategy

After compiling the files and executing the mandel-tar without options we get the following graph:

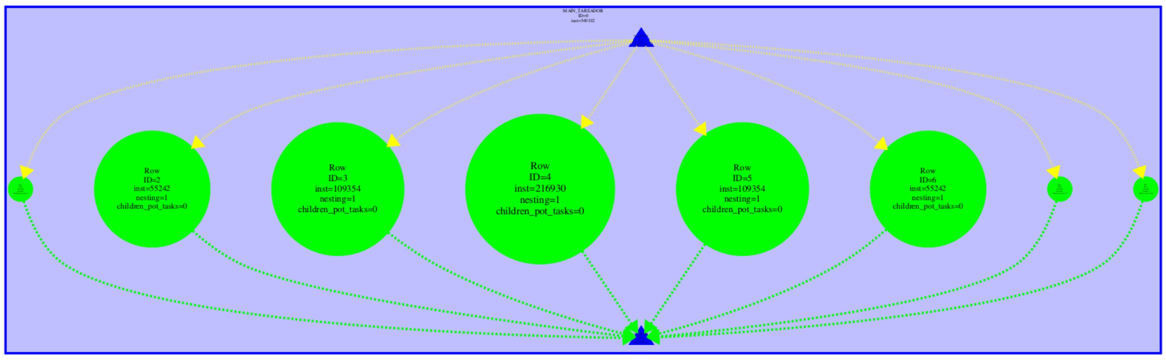


Figure 3: TDG of None option with Row Strategy

Upon generating the graph, it becomes evident that there are tasks that are significantly larger than others, which is suboptimal for parallelism purposes. Additionally, we observed that there are no dependencies between tasks, and all of them depend on the main task.

We then executed mandel-tar again, this time using the -d option. The resulting graph on Tareador, shown in Fig. Showing that each task has a dependency with the previous one, creating a serialization of tasks. The reason why the graph has become dependent on the preceding task is that we cannot paint two pixels at the same time.Notably, we identified the section of code responsible for this behavior, which is located inside the “if(output2display)” clause, where XSetForeground and XDrawPoint functions are used to set the color of pixels and display them on the screen.

To prevent different threads from accessing the same variable that stores the color, which can cause a data race, it is recommended to use #pragma omp critical before the XSetForeground and XDrawPoint functions in the subsequent sections.

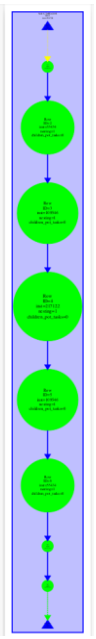


Figure 4: TDG of -d option with Row Strategy

Finally, we executed the program with the -h option, which computes the histogram along with the set execution. The resulting Tareador graph, depicted in Figure, showed that some tasks are much larger than others and that some tasks have multiple dependencies. In this scenario, threads access the k-1 position of the histogram (“if (output2histogram)” as we have binded the -h option), causing the code to become sequential since they cannot access nonexistent elements in that position. Moreover, a data race may occur because threads access the same memory position simultaneously. To protect this code region, we create a section that only one thread can access at a time, and we use #pragma omp atomic instead of #pragma omp critical because it is faster and ensures the serialization of a particular operation. This strategy will also be used in the upcoming sections.

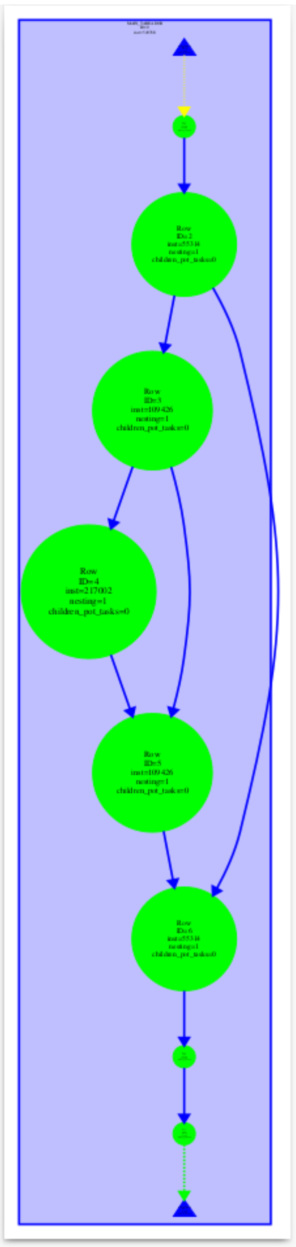
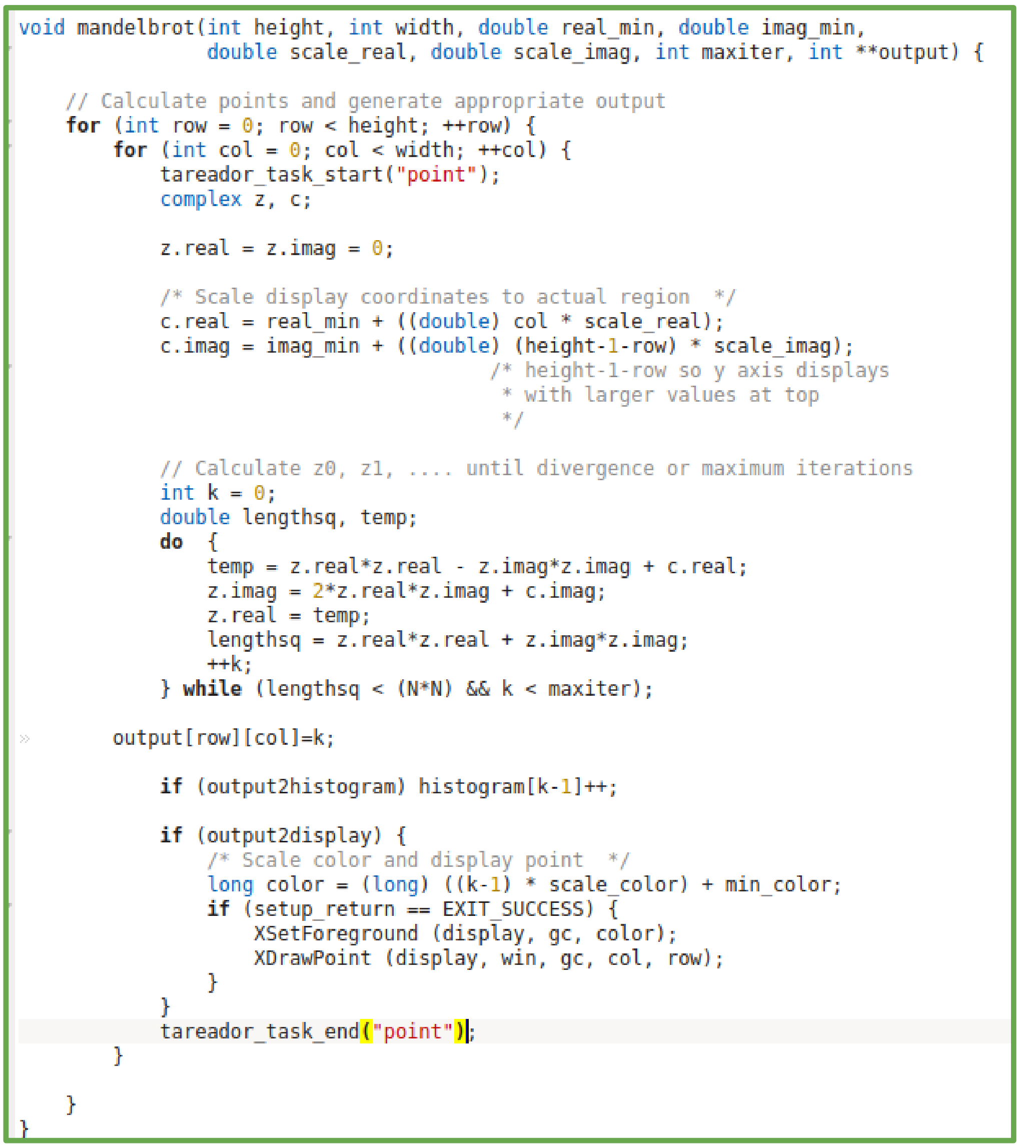


Figure 5: TDG of with -h option with Row Strategy

2.2 Point strategy

We will do the same study as we have done with the row strategy, once we get the results we will extract the conclusion by analyzing the features of the graph obviously.

First of all we need to specify to the tareador where it needs to create the new task with the following edition of the 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_task_start("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_task_end("point");
        }

    }
}
```

Figure 6: Implementation of Point Strategy

After compiling the files and executing the mandel-tar without options we get the following graph:

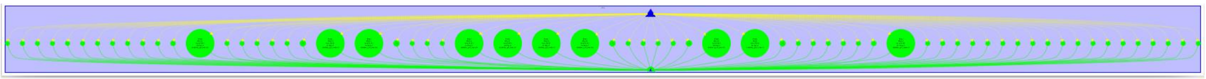


Figure 7: TDG of None option with Point Strategy

It was not difficult to see that after changing the previous code we will get an TDP like that as we are creating a huge amount of tasks compared with the row strategy. So even if the tasks are smaller than the row's, we still have larger tasks than the others so we have the same characteristics as the previous strategy: no dependencies between tasks and some tasks are bigger than others.

Last but not the least, the Row and Point strategies have the same part of the code that makes a difference in execution when using either the -d or the -h option. We find that in both strategies, there is a size imbalance between tasks, and tasks have only one dependency with the previous task (serialization). However, in the Point strategy, there are a lot more tasks because the creation is in the innermost loop.

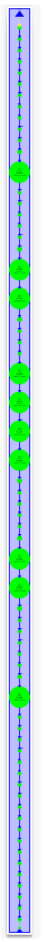


Figure 8: TDG with -d option with Point Strategy

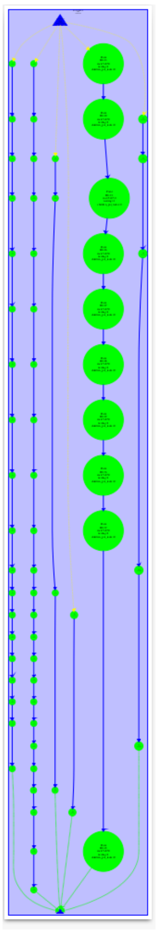


Figure 9: TDG with -h option with Point Strategy

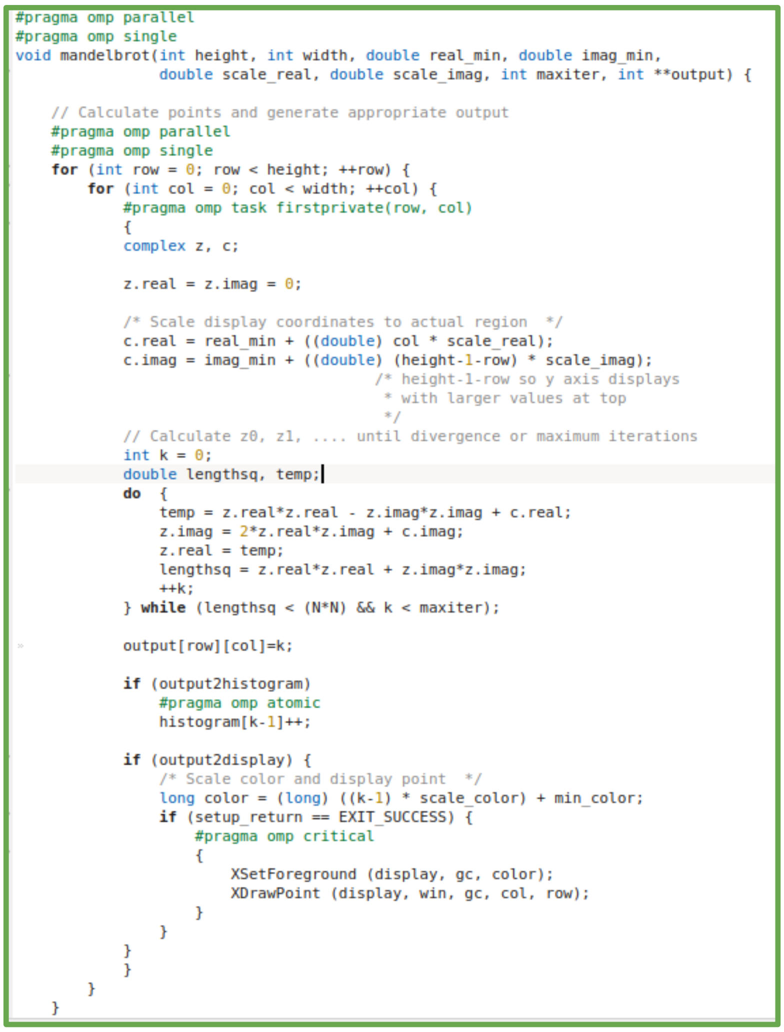
Finally, when executing with the -h option, we observe a different task dependence graph in the Point strategy. Although it appears that the tasks are executing in a parallel formation, but fact is that the bigger tasks are still being executed in a serialization.

In conclusion, the Row strategy is the most appropriate in this case because it has a lot fewer tasks to compute, and the tasks are executed sequentially in every execution. Additionally, the Point strategy has a problem with overheads when creating tasks every iteration in both loops which will become a significant number if we are parallelizing a real code for anything.

3. Point decomposition strategy

In the following sections, we will explore various ways to express the iterative task decomposition strategies for the Mandelbrot computation program. We will examine the scalability and other characteristics of our code.

The Point strategy will be implemented in the mandel-omp.c code using “task_start” and “taks_end” as we did with the previous sections. Initially, we will check if the code is utilizing Point parallelization and then proceed to add the OpenMP directives atomic and critical to safeguard the identified dependencies from the previous section, which were detected using Tareador. The edited code will be like this:



```c
#pragma omp parallel
#pragma omp single
void mandelbrot(int height, int width, double real_min, double imag_min,
                double scale_real, double scale_imag, int maxiter, int **output) {

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

            z.real = z.imag = 0;

            /* Scale display coordinates to actual region  */
            c.real = real_min + ((double) col * scale_real);
            c.imag = imag_min + ((double) (height-1-row) * scale_imag);
                                        /* height-1-row so y axis displays
                                         * with larger values at top
                                         */
            // Calculate z0, z1, .... until divergence or maximum iterations
            int k = 0;
            double lengthsq, temp;
            do  {
                temp = z.real*z.real - z.imag*z.imag + c.real;
                z.imag = 2*z.real*z.imag + c.imag;
                z.real = temp;
                lengthsq = z.real*z.real + z.imag*z.imag;
                ++k;
            } while (lengthsq < (N*N) && k < maxiter);

            output[row][col]=k;

            if (output2histogram)
                #pragma omp atomic
                histogram[k-1]++;

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

Figure 10: Implementation of point strategy taskloops without using neither num tasks nor grainsize

To compile and generate the parallel binary for this first version of the parallel code. In order to visually check the correctness of your parallelization, we will interactively execute it with the -d option to see what happens when running it with only 1 thread and a maximum of 10000 iterations per point and we will also execute the program with 2 threads and the same number of iterations per point in order to prove the correctness of our parallelization.

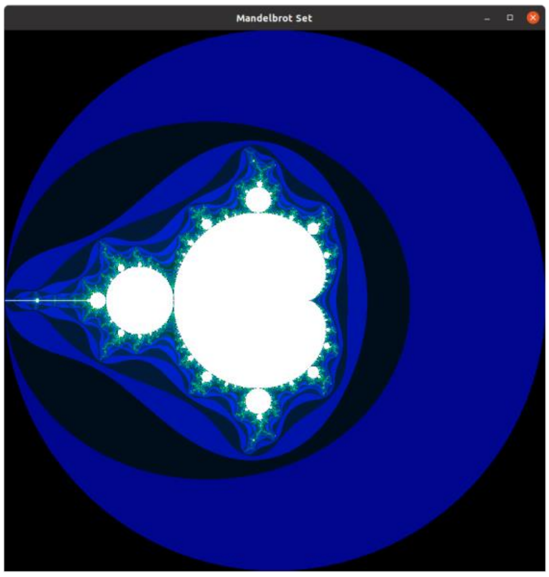


Figure 11: Execution with 1 thread

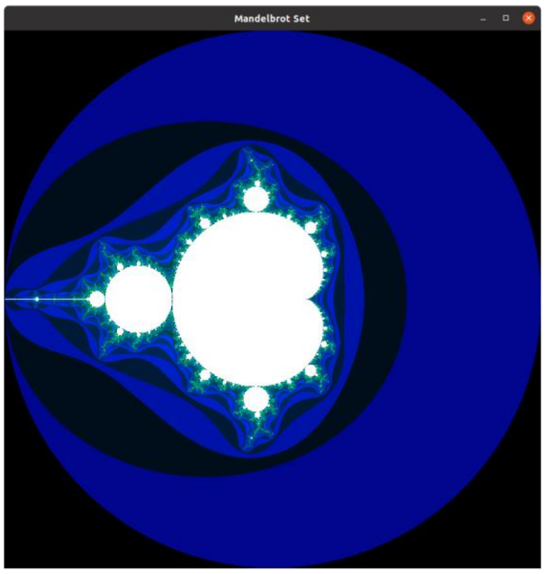


Figure 12: Execution with 2 thread

The results are what we have expected, those images have the same set of points as the image generated with the sequential version.

Our next objective is to determine the reduction in time achieved through parallel execution, as compared to sequential execution. To accomplish this, we will run mandel-seq and mandel-omp using the submit-omp.sh script, with 1 and 8 threads.

File	Time execution (seconds)
mandel-seq	2.609396
mandel-omp 1 thread	2.967533
mandel-omp 8 thread	1.542434

It is evident that there is a significant improvement in the total execution time with the parallel code. This results in a faster execution as compared to the sequential code. In order to provide a clearer representation of the speed-up and execution time, we will use submit-omp-strong.sh to execute the code for the 1-20 processor range and generate plots accordingly.

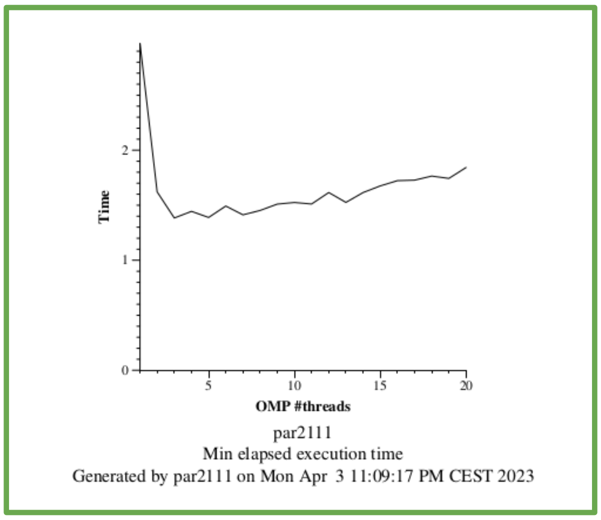


Figure 13: Execution time with 20 threads taskloop without using neither num tasks nor grainsize

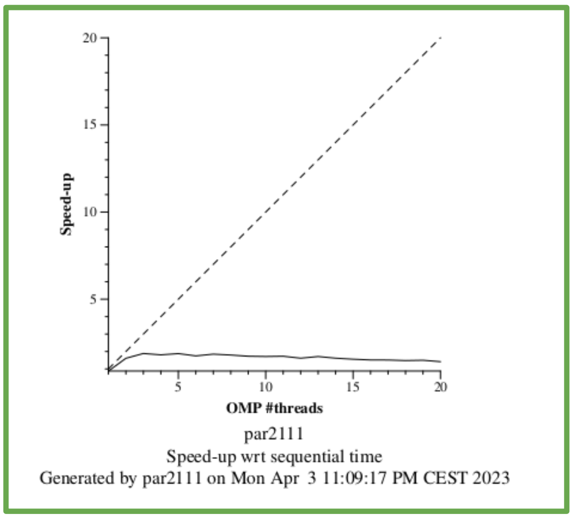


Figure 14: Speedup with 20 threads taskloop without using neither num tasks nor grainsize

We can see that the time execution is decrementing since we get at 3 threads and then it is getting worse as the execution time is incrementing again by the numbers of threads that we are adding. Moreover, the speed-up is so far from the optimum program. In any. case, the efficiency is really low and getting worse as the number of threads is increased. so the scalability is not considered appropriate in this case. Take a look at the execution times that are reported.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.37	0.31	0.35	0.36
Speedup	1.00	1.57	1.86	1.67	1.60
Efficiency	1.00	0.39	0.23	0.14	0.10

Table 1: Analysis done on Mon Apr 3 11:21:59 PM CEST 2023, par2111

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	0.85	0.82	0.84
LB (time executing explicit tasks)	1.0	0.89	0.89	0.9	0.9
Time per explicit task (average us)	4.89	5.7	5.91	6.04	6.03
Overhead per explicit task (synch %)	0.0	105.64	262.44	514.79	776.96
Overhead per explicit task (sched %)	5.31	32.32	23.67	21.61	21.46
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Mon Apr 3 11:21:59 PM CEST 2023, par2111

Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	1	4	8	12	16
Global efficiency	95.38%	37.34%	22.20%	13.26%	9.51%
Parallelization strategy efficiency	95.38%	45.28%	29.92%	19.84%	14.65%
Load balancing	100.00%	92.93%	55.79%	32.39%	23.13%
In execution efficiency	95.38%	48.73%	53.63%	61.26%	63.31%
Scalability for computation tasks	100.00%	82.45%	74.21%	66.81%	64.95%
IPC scalability	100.00%	80.05%	74.28%	69.02%	67.30%
Instruction scalability	100.00%	103.95%	104.40%	104.16%	104.09%
Frequency scalability	100.00%	99.09%	95.69%	92.94%	92.73%

Table 2: Analysis done on Mon Apr 3 11:21:59 PM CEST 2023, par2111

The parallelization strategy provides bad results for efficiency as we can see in the previous tables. One of the major problems is the overhead per explicit task (synchronization). It grows exponentially with the augment of threads. To gain a better understanding of how the execution is working, we will use the submit-extrae.sh script to run the mandel-omp file with 12 threads. Paraver provides different traces that can be visualized, including the implicit and explicit tasks showing the tasks generated and executed. The tables below provide a configuration file, presenting a profile of the explicit tasks.

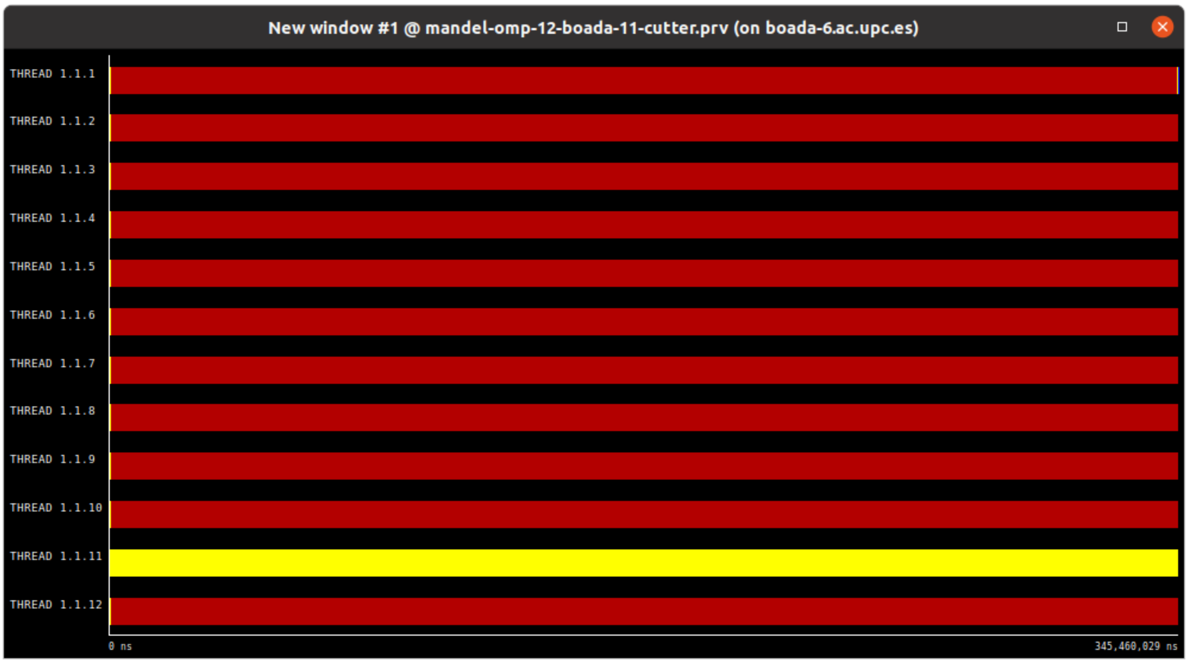
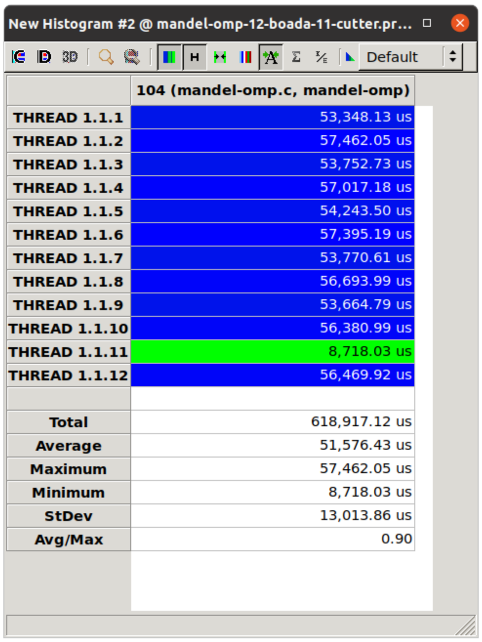
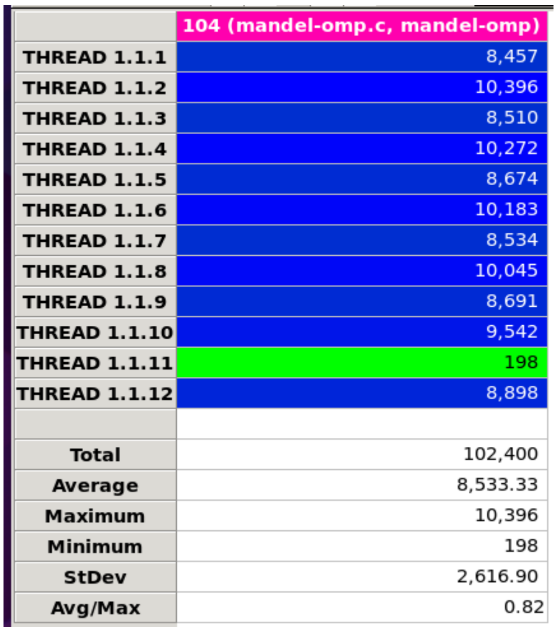


Figure 15: Timeline with 12 thread point taskloop without using neither num tasks nor grainsize



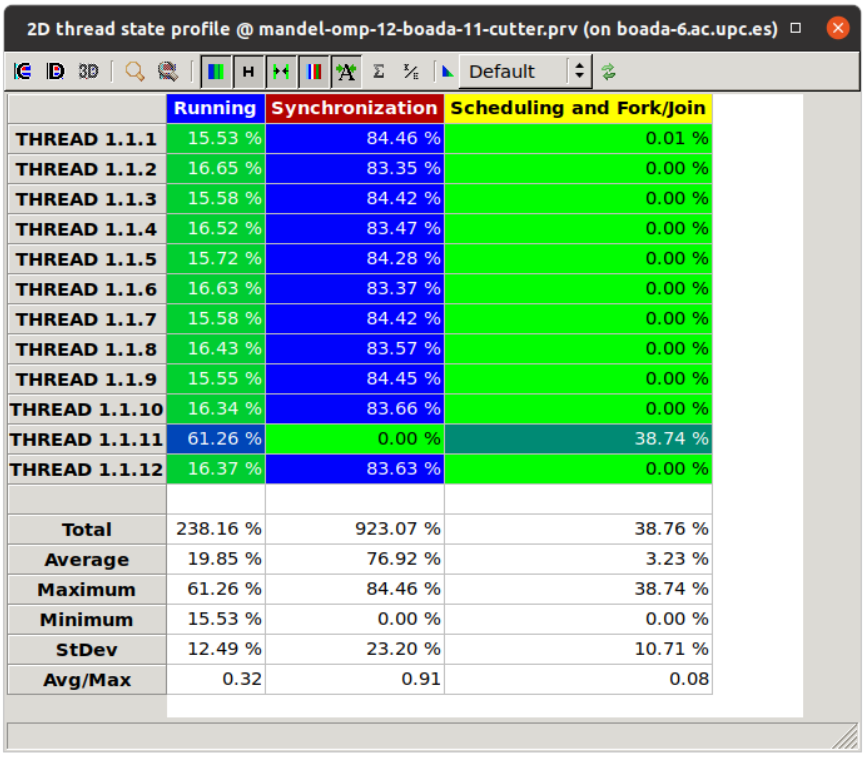
	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	53,348.13 us
THREAD 1.1.2	57,462.05 us
THREAD 1.1.3	53,752.73 us
THREAD 1.1.4	57,017.18 us
THREAD 1.1.5	54,243.50 us
THREAD 1.1.6	57,395.19 us
THREAD 1.1.7	53,770.61 us
THREAD 1.1.8	56,693.99 us
THREAD 1.1.9	53,664.79 us
THREAD 1.1.10	56,380.99 us
THREAD 1.1.11	8,718.03 us
THREAD 1.1.12	56,469.92 us
Total	618,917.12 us
Average	51,576.43 us
Maximum	57,462.05 us
Minimum	8,718.03 us
StDev	13,013.86 us
Avg/Max	0.90

Figure 16: Histogram of the execution time with 12 threads of point taskloop without using neither num tasks nor grainsize



	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	8,457
THREAD 1.1.2	10,396
THREAD 1.1.3	8,510
THREAD 1.1.4	10,272
THREAD 1.1.5	8,674
THREAD 1.1.6	10,183
THREAD 1.1.7	8,534
THREAD 1.1.8	10,045
THREAD 1.1.9	8,691
THREAD 1.1.10	9,542
THREAD 1.1.11	198
THREAD 1.1.12	8,898
Total	102,400
Average	8,533.33
Maximum	10,396
Minimum	198
StDev	2,616.90
Avg/Max	0.82

Figure 17: Histogram of tasks executed with 12 threads of point taskloop without using neither num tasks nor grainsize


2D thread state profile @ mandel-omp-12-boada-11-cutter.prv (on boada-6.ac.upc.es)

Default

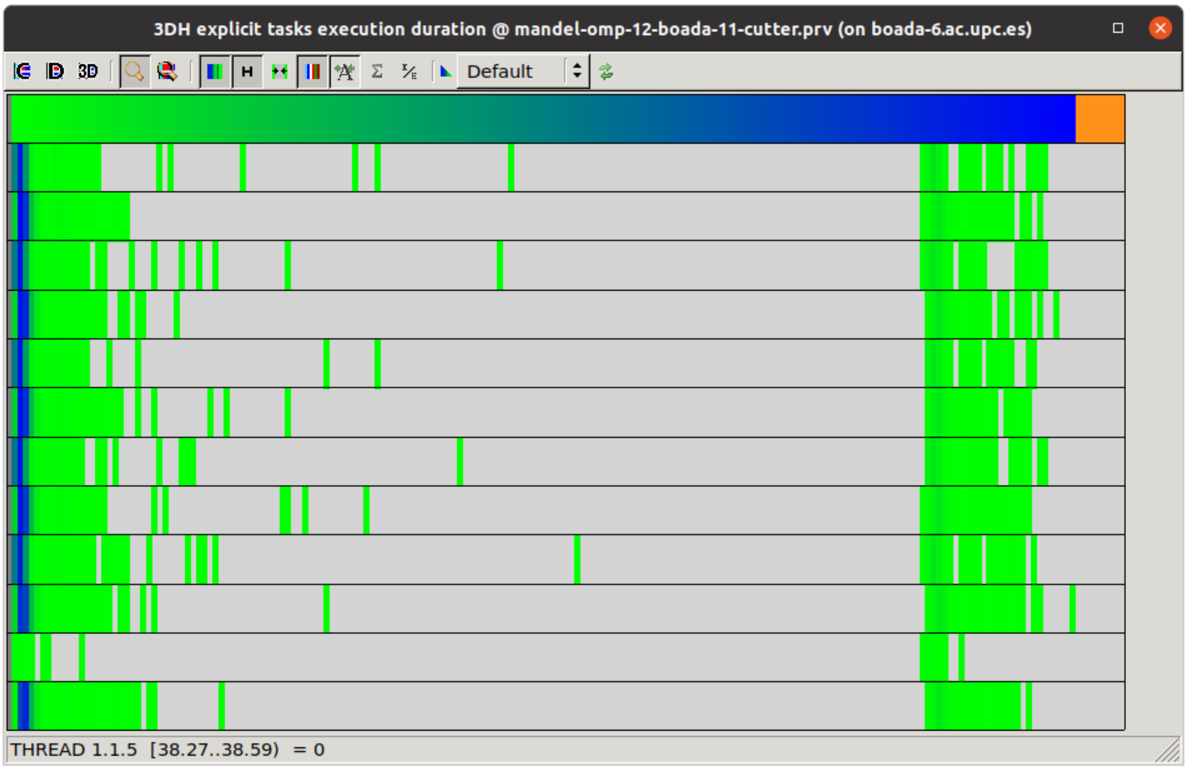
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	15.53 %	84.46 %	0.01 %
THREAD 1.1.2	16.65 %	83.35 %	0.00 %
THREAD 1.1.3	15.58 %	84.42 %	0.00 %
THREAD 1.1.4	16.52 %	83.47 %	0.00 %
THREAD 1.1.5	15.72 %	84.28 %	0.00 %
THREAD 1.1.6	16.63 %	83.37 %	0.00 %
THREAD 1.1.7	15.58 %	84.42 %	0.00 %
THREAD 1.1.8	16.43 %	83.57 %	0.00 %
THREAD 1.1.9	15.55 %	84.45 %	0.00 %
THREAD 1.1.10	16.34 %	83.66 %	0.00 %
THREAD 1.1.11	61.26 %	0.00 %	38.74 %
THREAD 1.1.12	16.37 %	83.63 %	0.00 %
Total	238.16 %	923.07 %	38.76 %
Average	19.85 %	76.92 %	3.23 %
Maximum	61.26 %	84.46 %	38.74 %
Minimum	15.53 %	0.00 %	0.00 %
StDev	12.49 %	23.20 %	10.71 %
Avg/Max	0.32	0.91	0.08

Figure 18: 2D thread state profile with 12 thread point taskloop without using neither num tasks nor grainsize

The tables above display the number of tasks per thread and the execution time per thread. Based on these tables, we can infer that all threads, except for thread 11 that are balanced in terms of task execution. Thread number 11 creates fewer tasks as compared to other threads because it is responsible for generating tasks.

The total number of tasks generated is 102,400, which are not well-balanced between threads. Thread 11 executes significantly fewer tasks and has less execution time as compared to the rest of the threads. However, the number of tasks is more relevant than the execution time, as it determines the threads responsible for task creation.

The following histogram shows the duration of the explicit tasks:


3DH explicit tasks execution duration @ mandel-omp-12-boada-11-cutter.prv (on boada-6.ac.upc.es)
Default

THREAD 1.1.5 [38.27..38.59) = 0

Figure 19: 3DH histogram explicitis tasks executions duration 12 threads of point taskloop without using neither num tasks nor grainsize

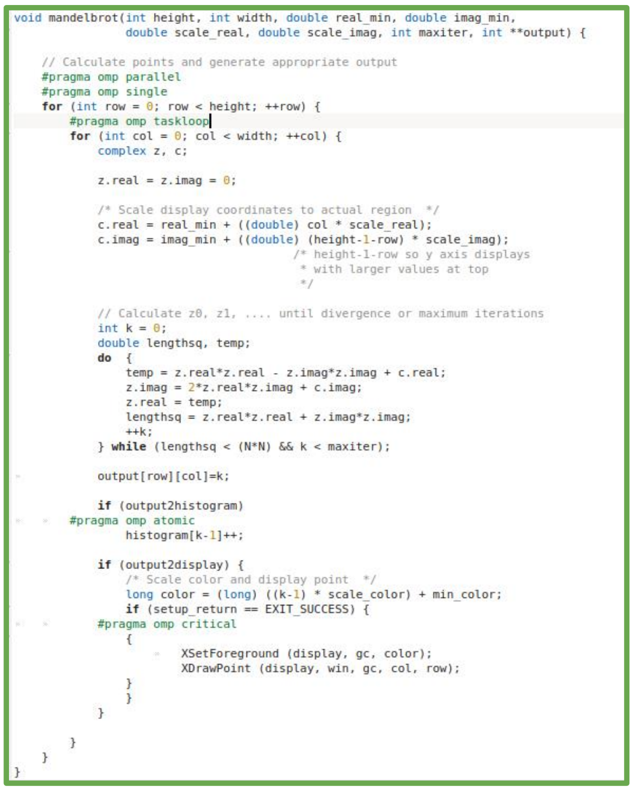
It appears that the code is using OpenMP taskloop directive to create tasks in each iteration of the “row” loop, and each task represents an iteration of the “col” loop. This leads to the creation of multiple tasks at the same time, resulting in the "green barriers" shown in [histogram].

The trace diagram in Paraver indicates that there is a significant imbalance between tasks, possibly due to overheads per thread. This could be due to inappropriate granularity of tasks, which can be adjusted by using other OpenMP directives.

In Section 2.2, the code was modified using the OpenMP collapse directive to improve task granularity, which may have resulted in better load balancing and reduced overheads per thread. It is possible that similar directives could be used to improve task granularity and reduce the imbalance between tasks in the current implementation.

3.1. Granularity control using taskloop

In this section we will edit the the previous code by adding #pragma omp taskloop before the col loop and inside the row loop, just like this:



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

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

            z.real = z.imag = 0;

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

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

            output[row][col]=k;

            if (output2histogram)
        #pragma omp atomic
                histogram[k-1]++;

            if (output2display) {
                /* Scale color and display point  */
                long color = (long) ((k-1) * scale_color) + min_color;
                if (setup_return == EXIT_SUCCESS) {
            #pragma omp critical
                {
                        XSetForeground (display, gc, color);
                        XDrawPoint (display, win, gc, col, row);
                }
                }
            }

        }
    }
}
```

Figure 20: Implementation of the point strategy taskloop

In order to check the correctness of the code we will compile this new version and interactively execute it with 1 and 2 threads with the -d option to verify that the code still visualizes the correct Mandelbrot set

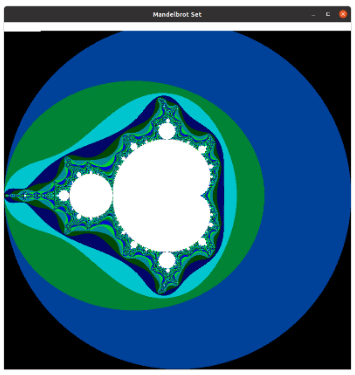


Figure 20: Execution with 1 thread

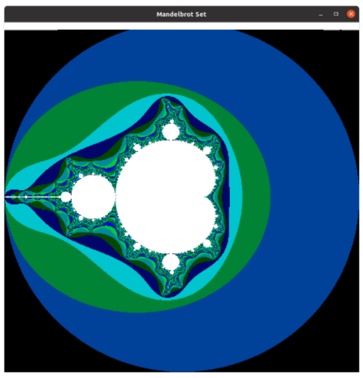


Figure 21: Execution with 2 thread

According to these images we can see that our code is correct, so now we will submit the new execution of the submit-omp.sh script for 1 and 8 threads to validate the output files generated with respect to the output of the original sequential program and to observe the new execution times.

Thread	Time Execution

1	2.561718
8	0.547508

We can see that the execution time with 8 threads has improved compared to the previous one. Now we will execute the code for the 1-20 processor range and generate plots accordingly.

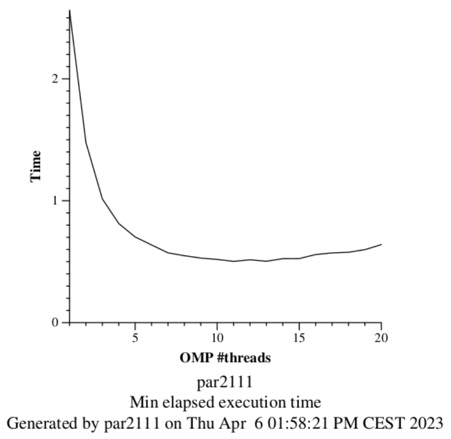


Figure 21: Execution time with
20 threads of point taskloop

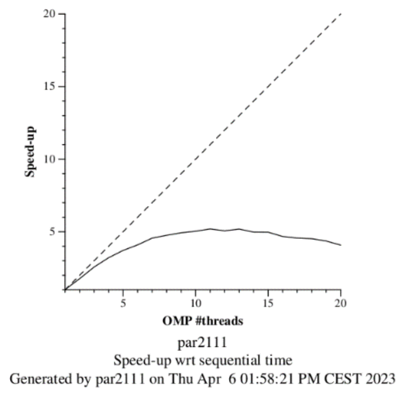
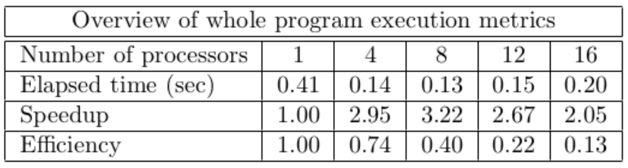


Figure 22: Speedup with
20 threads of point taskloop

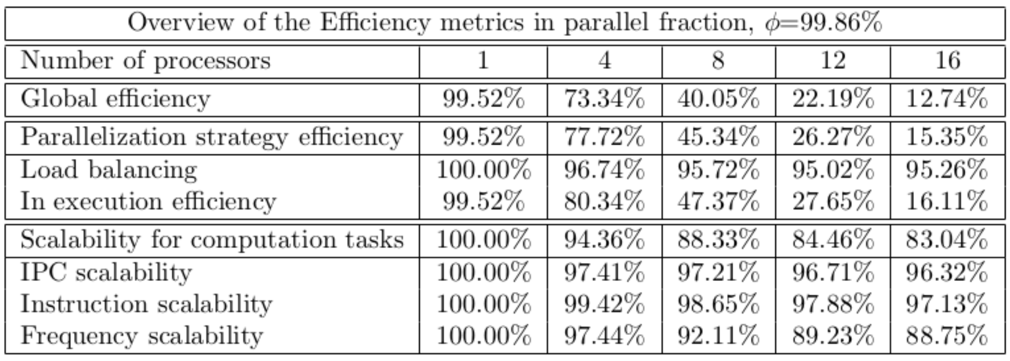
It seems that the implementation using the OpenMP taskloop directive is performing better than the previous implementation using the task directive in terms of speedup and scalability. We see that after thread number 7 or 8 the time execution is getting constant so the speed up happens the same. Thus, the code still doesn't provide an optimal parallelism as we can see in the plots.



Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.41	0.14	0.13	0.15	0.20
Speedup	1.00	2.95	3.22	2.67	2.05
Efficiency	1.00	0.74	0.40	0.22	0.13

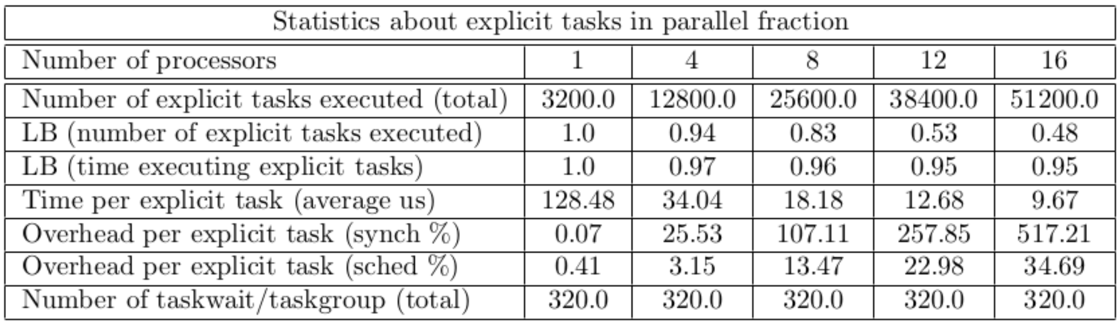
Table 1: Analysis done on Thu Apr 6 01:53:00 PM CEST 2023, par2111

However, the efficiency of the implementation is still not optimal. This could be due to several factors, such as the overheads associated with creating and managing tasks or load imbalances between threads



Overview of the Efficiency metrics in parallel fraction, ϕ=99.86%					
Number of processors	1	4	8	12	16
Global efficiency	99.52%	73.34%	40.05%	22.19%	12.74%
Parallelization strategy efficiency	99.52%	77.72%	45.34%	26.27%	15.35%
Load balancing	100.00%	96.74%	95.72%	95.02%	95.26%
In execution efficiency	99.52%	80.34%	47.37%	27.65%	16.11%
Scalability for computation tasks	100.00%	94.36%	88.33%	84.46%	83.04%
IPC scalability	100.00%	97.41%	97.21%	96.71%	96.32%
Instruction scalability	100.00%	99.42%	98.65%	97.88%	97.13%
Frequency scalability	100.00%	97.44%	92.11%	89.23%	88.75%

Table 2: Analysis done on Thu Apr 6 01:53:00 PM CEST 2023, par2111



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	0.97	0.96	0.95	0.95
Time per explicit task (average us)	128.48	34.04	18.18	12.68	9.67
Overhead per explicit task (synch %)	0.07	25.53	107.11	257.85	517.21
Overhead per explicit task (sched %)	0.41	3.15	13.47	22.98	34.69
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Thu Apr 6 01:53:00 PM CEST 2023, par2111

.

It appears that the current implementation using the OpenMP taskloop directive has improved load balancing, but the "In execution efficiency" metric has worsened, which is affecting the overall efficiency of the program.

Table 3 indicates that the implementation using the taskloop directive has executed fewer explicit tasks compared to the task version, but the average execution time for explicit tasks has increased. This could be due to the larger task granularity or the overheads associated with creating and managing tasks.

Furthermore, the percentage of task synchronization is still high, indicating that there may be synchronization overheads that are affecting program performance. To further analyze this problem, Paraver can be used to visualize the synchronization behavior of the program.

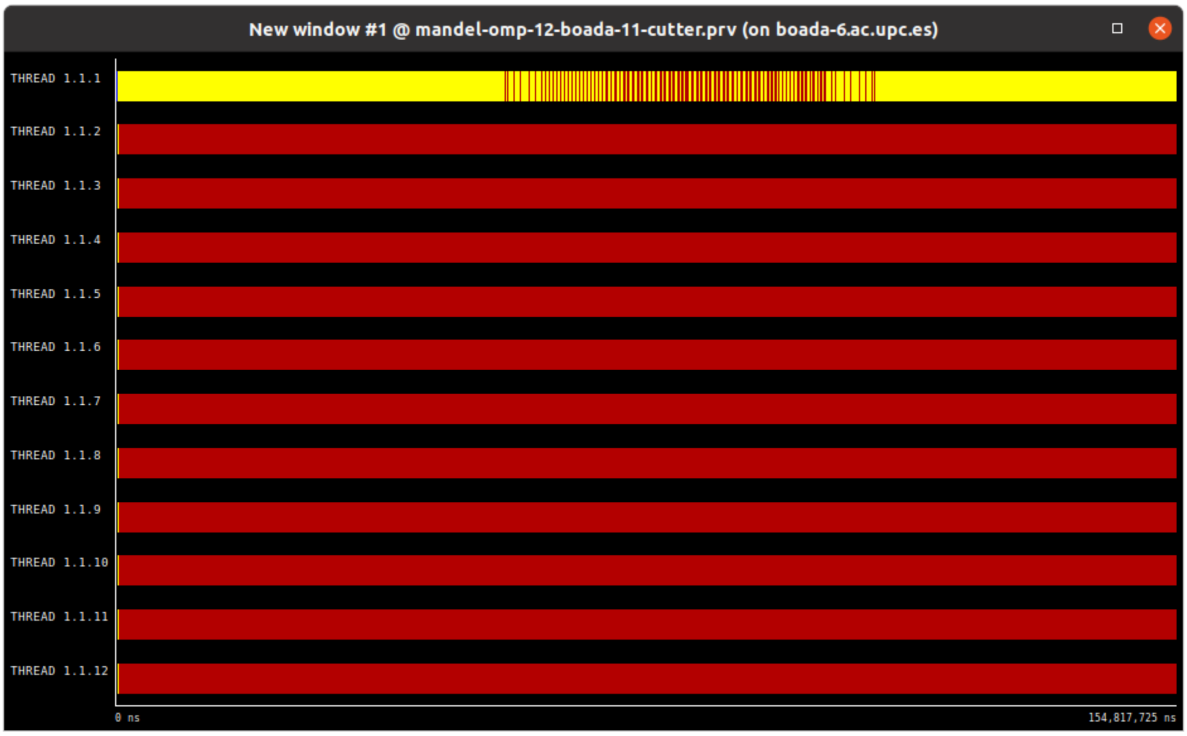
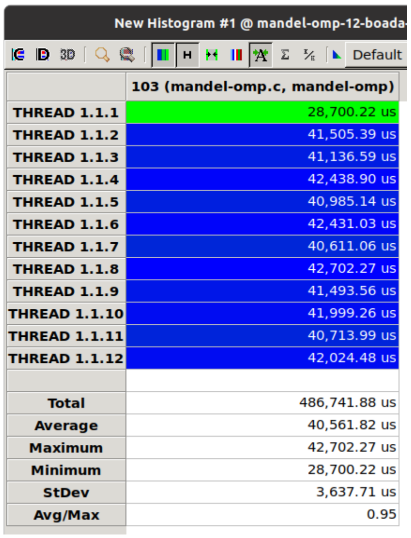
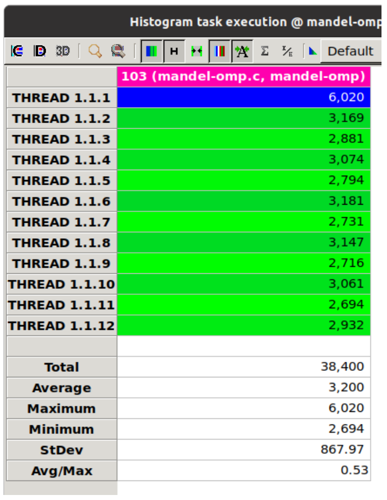


Figure 15: Timeline with 12 thread point taskloop



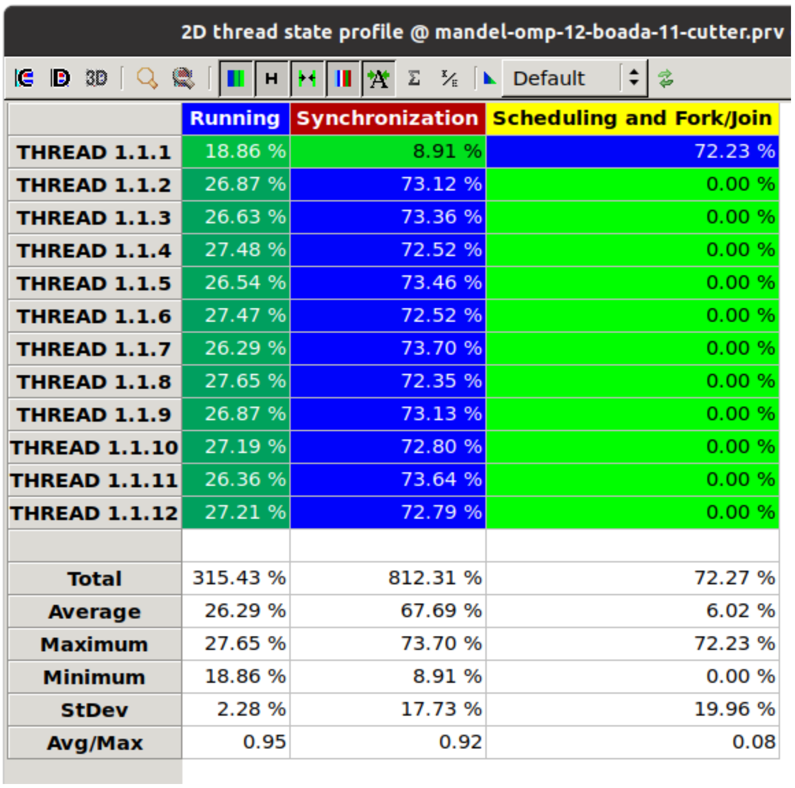
	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	28,700.22 us
THREAD 1.1.2	41,505.39 us
THREAD 1.1.3	41,136.59 us
THREAD 1.1.4	42,438.90 us
THREAD 1.1.5	40,985.14 us
THREAD 1.1.6	42,431.03 us
THREAD 1.1.7	40,611.06 us
THREAD 1.1.8	42,702.27 us
THREAD 1.1.9	41,493.56 us
THREAD 1.1.10	41,999.26 us
THREAD 1.1.11	40,713.99 us
THREAD 1.1.12	42,024.48 us
Total	486,741.88 us
Average	40,561.82 us
Maximum	42,702.27 us
Minimum	28,700.22 us
StDev	3,637.71 us
Avg/Max	0.95

Figure 16: Histogram of the execution time with 12 threads of point taskloop



	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	6,020
THREAD 1.1.2	3,169
THREAD 1.1.3	2,881
THREAD 1.1.4	3,074
THREAD 1.1.5	2,794
THREAD 1.1.6	3,181
THREAD 1.1.7	2,731
THREAD 1.1.8	3,147
THREAD 1.1.9	2,716
THREAD 1.1.10	3,061
THREAD 1.1.11	2,694
THREAD 1.1.12	2,932
Total	38,400
Average	3,200
Maximum	6,020
Minimum	2,694
StDev	867.97
Avg/Max	0.53

Figure 17: Histogram of tasks executed with 12 threads of point taskloop



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	18.86 %	8.91 %	72.23 %
THREAD 1.1.2	26.87 %	73.12 %	0.00 %
THREAD 1.1.3	26.63 %	73.36 %	0.00 %
THREAD 1.1.4	27.48 %	72.52 %	0.00 %
THREAD 1.1.5	26.54 %	73.46 %	0.00 %
THREAD 1.1.6	27.47 %	72.52 %	0.00 %
THREAD 1.1.7	26.29 %	73.70 %	0.00 %
THREAD 1.1.8	27.65 %	72.35 %	0.00 %
THREAD 1.1.9	26.87 %	73.13 %	0.00 %
THREAD 1.1.10	27.19 %	72.80 %	0.00 %
THREAD 1.1.11	26.36 %	73.64 %	0.00 %
THREAD 1.1.12	27.21 %	72.79 %	0.00 %
Total	315.43 %	812.31 %	72.27 %
Average	26.29 %	67.69 %	6.02 %
Maximum	27.65 %	73.70 %	72.23 %
Minimum	18.86 %	8.91 %	0.00 %
StDev	2.28 %	17.73 %	19.96 %
Avg/Max	0.95	0.92	0.08

Figure 18: 2D thread state profile with 12 thread point taskloop

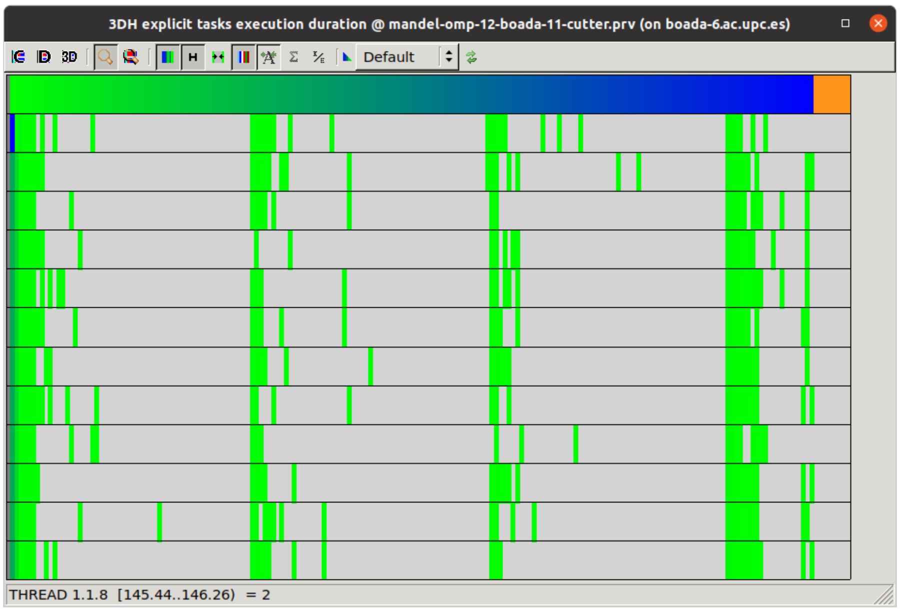

3DH explicit tasks execution duration @ mandel-omp-12-boada-11-cutter.prv (on boada-6.ac.upc.es)
Default
THREAD 1.1.8 [145.44..146.26) = 2

Figure 19: 3DH histogram explicitis tasks executions duration 12 threads of point taskloop

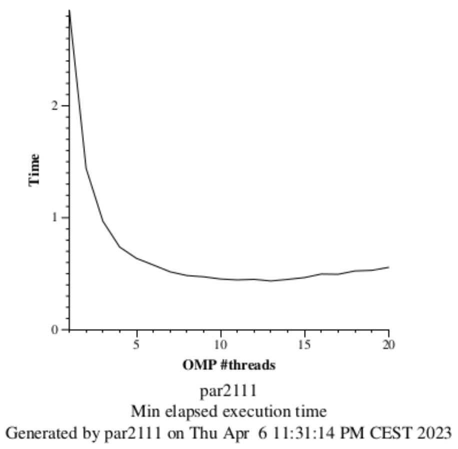
It seems that the implementation using the OpenMP taskloop directive has reduced the number of task creations, but there are still synchronization overheads due to the large number of small tasks. Figure 17 and 18 indicate that the number of task creations has decreased. However, figure 19 shows that the execution time of each task has increased compared to the previous version, which could be a result of the task granularity.Thus, the number of threads waiting has reduced.

Figure 20 indicates that the percentage of synchronization in the execution is still high, which could be due to the synchronization overheads associated with managing a large number of tasks.

Overall, it appears that the current implementation has addressed some of the issues with the previous version, such as load balancing and task granularity. However, there are still synchronization overheads that need to be addressed to further improve efficiency.

3.2. Granularity control using taskloop nogroup

When trying to use the hints in taskgroup construct and in taskwait construct, the program only allows us to select the hint with the option taskgroup because in the other way it generates nothing. Now we will execute the code for the 1-20 processor range and generate plots accordingly.



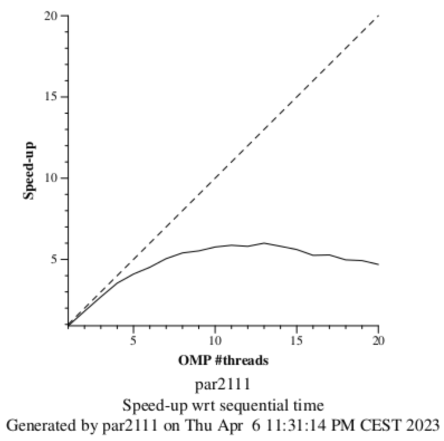


Figure 20: Execution time with
20 threads of point taskloop no group

Figure 21: Speed up with
20 threads of point taskloop nogroup

We can see that these plots compared to the previous ones have a little improvement of time execution and speed up.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.11	0.15	0.18
Speedup	1.00	3.66	4.21	2.97	2.57
Efficiency	1.00	0.92	0.53	0.25	0.16

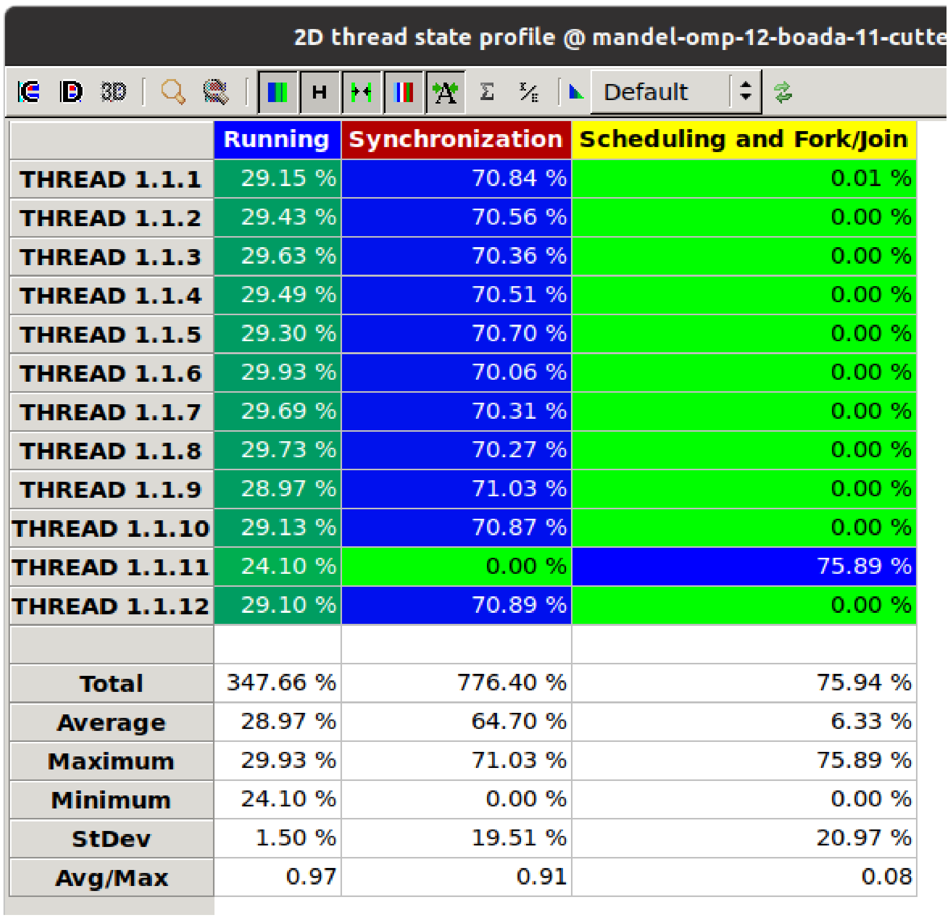
Table 1: Analysis done on Thu Apr 6 11:51:41 PM CEST 2023, par2111

Overview of the Efficiency metrics in parallel fraction, ϕ=99.94%					
Number of processors	1	4	8	12	16
Global efficiency	99.72%	91.42%	52.54%	24.68%	16.07%
Parallelization strategy efficiency	99.72%	95.24%	59.52%	28.96%	19.24%
Load balancing	100.00%	98.98%	98.37%	96.74%	95.72%
In execution efficiency	99.72%	96.22%	60.51%	29.93%	20.10%
Scalability for computation tasks	100.00%	95.99%	88.28%	85.22%	83.51%
IPC scalability	100.00%	98.37%	97.18%	97.24%	96.45%
Instruction scalability	100.00%	99.45%	98.74%	98.02%	97.34%
Frequency scalability	100.00%	98.12%	92.00%	89.41%	88.96%

Table 2: Analysis done on Thu Apr 6 11:51:41 PM CEST 2023, par2111

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	0.5	0.95	0.62	0.94
LB (time executing explicit tasks)	1.0	0.99	0.98	0.97	0.96
Time per explicit task (average us)	143.04	37.24	20.25	13.98	10.7
Overhead per explicit task (synch %)	0.0	3.12	58.51	223.64	392.3
Overhead per explicit task (sched %)	0.28	1.87	9.54	21.87	27.81
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Thu Apr 6 11:51:41 PM CEST 2023, par2111


2D thread state profile @ mandel-omp-12-boada-11-cutte

Default

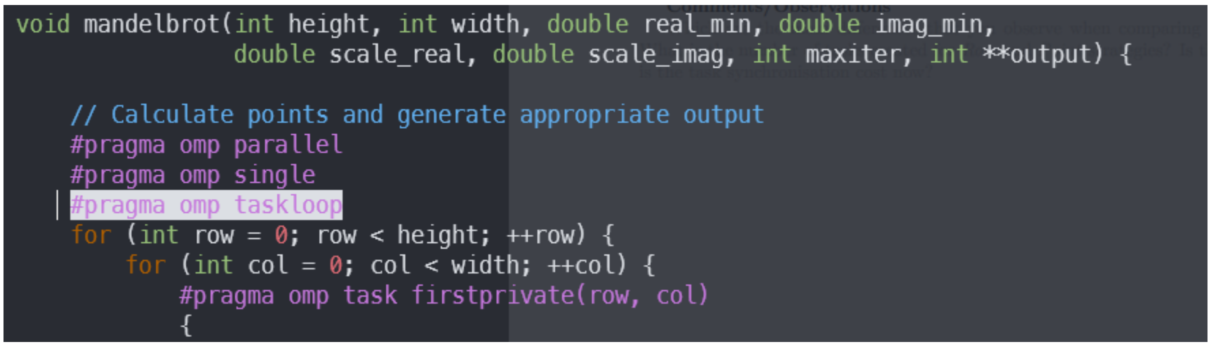
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	29.15 %	70.84 %	0.01 %
THREAD 1.1.2	29.43 %	70.56 %	0.00 %
THREAD 1.1.3	29.63 %	70.36 %	0.00 %
THREAD 1.1.4	29.49 %	70.51 %	0.00 %
THREAD 1.1.5	29.30 %	70.70 %	0.00 %
THREAD 1.1.6	29.93 %	70.06 %	0.00 %
THREAD 1.1.7	29.69 %	70.31 %	0.00 %
THREAD 1.1.8	29.73 %	70.27 %	0.00 %
THREAD 1.1.9	28.97 %	71.03 %	0.00 %
THREAD 1.1.10	29.13 %	70.87 %	0.00 %
THREAD 1.1.11	24.10 %	0.00 %	75.89 %
THREAD 1.1.12	29.10 %	70.89 %	0.00 %
Total	347.66 %	776.40 %	75.94 %
Average	28.97 %	64.70 %	6.33 %
Maximum	29.93 %	71.03 %	75.89 %
Minimum	24.10 %	0.00 %	0.00 %
StDev	1.50 %	19.51 %	20.97 %
Avg/Max	0.97	0.91	0.08

Figure 22: 2D thread state profile with 12 thread point taskloop nogroup

Upon examining figures 20 and 21, it becomes apparent that the percentage of synchronization overheads does not vary significantly between the previous one. However, we observe that there is a change in the distribution of task creations among threads, ranging from thread 1 to thread 11. Table 2 shows that there is no significant change in scalability, and while the efficiency of the parallelization strategy does improve slightly. Overall, incorporating the nogroup clause does not result in any significant enhancements.

4. Row decomposition strategy

In this section, we move *#pragma omp taskloop* before the first loop now. Once the mandel-omp.c code was modified, we verified that it was running correctly.

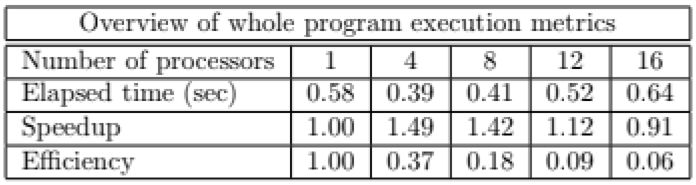


```
void mandelbrot(int height, int width, double real_min, double imag_min,
                double scale_real, double scale_imag, int maxiter, int **output) {

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

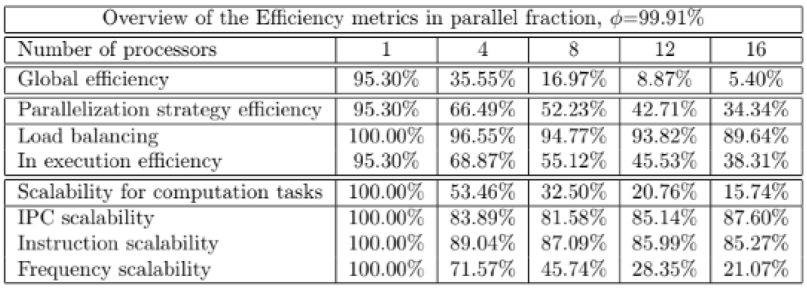
Figure 10: Implementation of Row strategy taskloops

Then we go to study its scalability. First we generated Modelfactor tables using submit-strong-extrae.sh and also the plots generated:



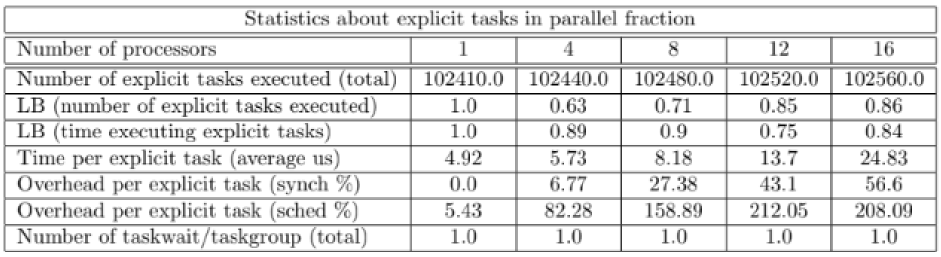
Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.39	0.41	0.52	0.64
Speedup	1.00	1.49	1.42	1.12	0.91
Efficiency	1.00	0.37	0.18	0.09	0.06

Table 1: Analysis done on Thu Apr 6 01:57:22 PM CEST 2023, par2110



Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	1	4	8	12	16
Global efficiency	95.30%	35.55%	16.97%	8.87%	5.40%
Parallelization strategy efficiency	95.30%	66.49%	52.23%	42.71%	34.34%
Load balancing	100.00%	96.55%	94.77%	93.82%	89.64%
In execution efficiency	95.30%	68.87%	55.12%	45.53%	38.31%
Scalability for computation tasks	100.00%	53.46%	32.50%	20.76%	15.74%
IPC scalability	100.00%	83.89%	81.58%	85.14%	87.60%
Instruction scalability	100.00%	89.04%	87.09%	85.99%	85.27%
Frequency scalability	100.00%	71.57%	45.74%	28.35%	21.07%

Table 2: Analysis done on Thu Apr 6 01:57:22 PM CEST 2023, par2110



Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	102410.0	102440.0	102480.0	102520.0	102560.0
LB (number of explicit tasks executed)	1.0	0.63	0.71	0.85	0.86
LB (time executing explicit tasks)	1.0	0.89	0.9	0.75	0.84
Time per explicit task (average us)	4.92	5.73	8.18	13.7	24.83
Overhead per explicit task (synch %)	0.0	6.77	27.38	43.1	56.6
Overhead per explicit task (sched %)	5.43	82.28	158.89	212.05	208.09
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Thu Apr 6 01:57:22 PM CEST 2023, par2110

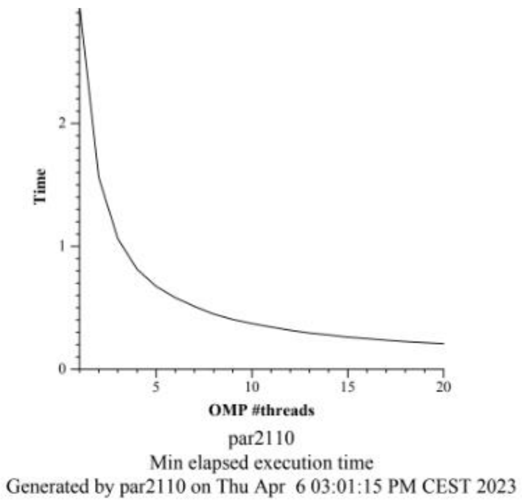


Figure 23: Execution time with
20 threads of row taskloop

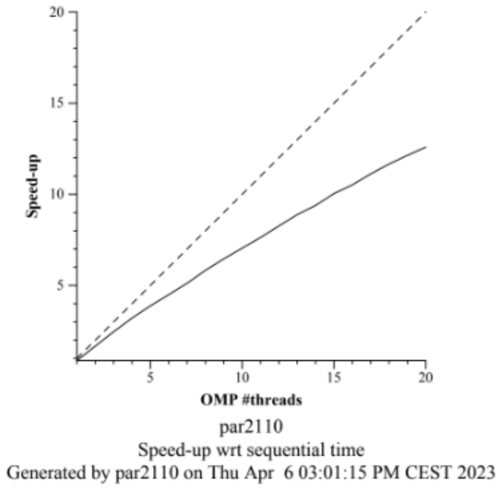


Figure 24: Speedup with
20 threads of row taskloop

Figure 23 and Table 1 demonstrate a considerable improvement in speedup compared to the Point version. The graphic displays a more linear pattern, while the speedup values are closer to the number of threads used, resulting in adequate efficiency.

Table 2 reveals a significant improvement in the execution efficiency, leading to a better global efficiency, despite a slight decrease in load balance. Moreover, Table 3 shows a reduction in the total number of explicit tasks executed and a significant increase in the average time per explicit task due to the coarse granularity established, resulting in fewer synchronization overheads.

The decrease in synchronization overheads is also evident in the Paraver figures.

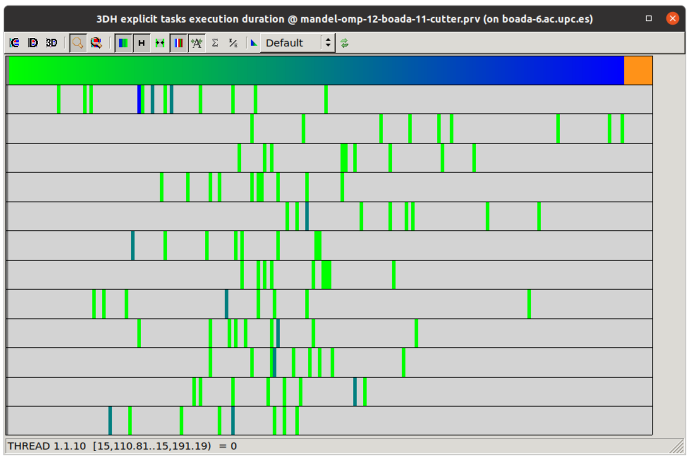

3DH explicit tasks execution duration @ mandel-omp-12-boada-11-cutter.prv (on boada-6.ac.upc.es)
Default
THREAD 1.1.10 [15,110.81..15,191.19) = 0

Figure 25: 3DH histogram explicitis tasks executions duration of 12 threads of row taskloop

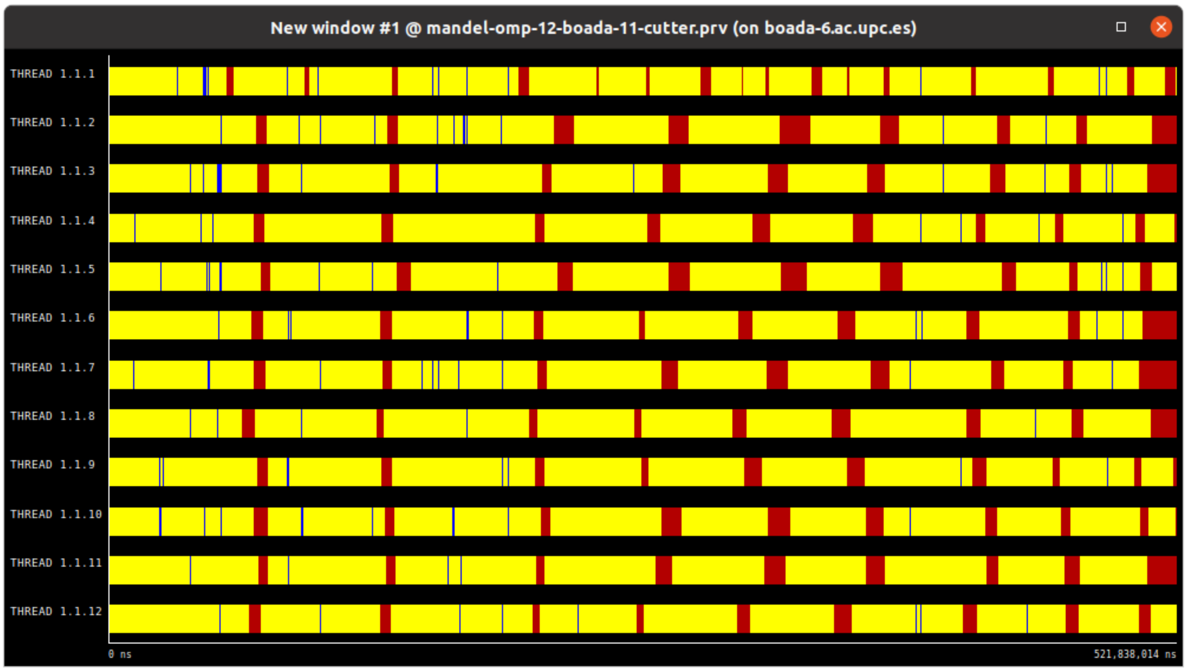
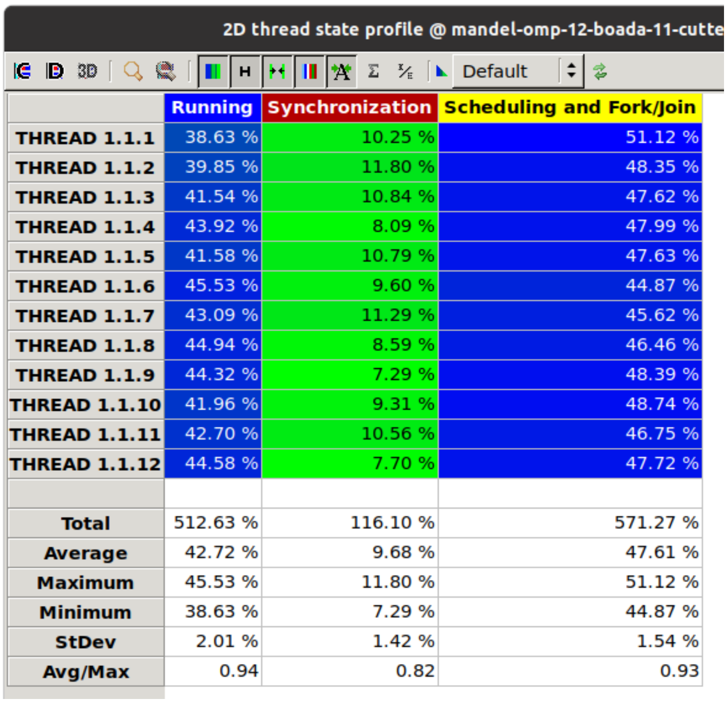


Figure 26: Timeline with 12 thread of row taskloop


2D thread state profile @ mandel-omp-12-boada-11-cutte
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	38.63 %	10.25 %	51.12 %
THREAD 1.1.2	39.85 %	11.80 %	48.35 %
THREAD 1.1.3	41.54 %	10.84 %	47.62 %
THREAD 1.1.4	43.92 %	8.09 %	47.99 %
THREAD 1.1.5	41.58 %	10.79 %	47.63 %
THREAD 1.1.6	45.53 %	9.60 %	44.87 %
THREAD 1.1.7	43.09 %	11.29 %	45.62 %
THREAD 1.1.8	44.94 %	8.59 %	46.46 %
THREAD 1.1.9	44.32 %	7.29 %	48.39 %
THREAD 1.1.10	41.96 %	9.31 %	48.74 %
THREAD 1.1.11	42.70 %	10.56 %	46.75 %
THREAD 1.1.12	44.58 %	7.70 %	47.72 %
Total	512.63 %	116.10 %	571.27 %
Average	42.72 %	9.68 %	47.61 %
Maximum	45.53 %	11.80 %	51.12 %
Minimum	38.63 %	7.29 %	44.87 %
StDev	2.01 %	1.42 %	1.54 %
Avg/Max	0.94	0.82	0.93

Figure 27: 2D thread state profile with 12 thread of row taskloop

Conclusion

In conclusion, between both strategys it is obvious to see that the Row decomposition strategy has a slight advantage over the Point strategy in terms of execution as we can see all the results compared between both strategys. However, this advantage is not very significant due to the trade-off between time per task and the number of tasks but in anyway it is clear to see that using a coarse granularity, such as the Row strategy, is more efficient.