# PAR Laboratory Deliverable
# Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Víctor Asenjo Carvajal - par1302
Ferran De La Varga Antoja - par1305

April 2022
Semester of Spring 2021-2022

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

# Index

## Introduction

In this laboratory section we are going to optimize the code that calculates the Mandelbrot set. This algorithm computes the expression $z_{n+1} = z_n^2 + c$ starting with $z_0 = 0$ and $z_n$ never exceeds a certain number.

A plot of the Mandelbrot set is created by colouring each point c in the complex plane with the number of steps max for which $|z_{max}| \geq 2$. In order to make the problem doable, the maximum number of steps is also limited: if that number of steps is reached, then the point c is said to belong to the Mandelbrot set.

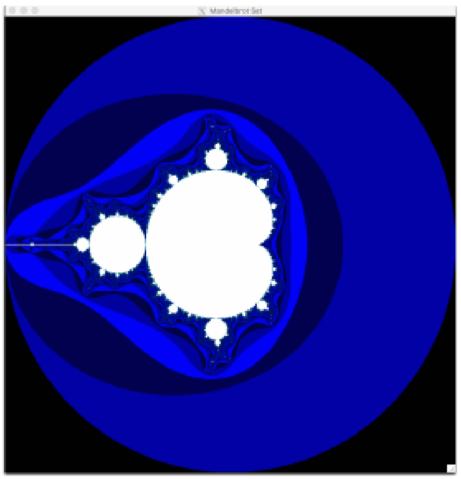


Figure 1: Fractal shape of the Mandelbrot set

# 1 Task decomposition and granularity analysis

## 1.1 Row task decomposition strategy

In the Figure 2 we can see a simplified version of the Mandelbrot's code. Let's analyze it. At first glance we can see two relevant code regions. The first one consists of two nested loops that iterate the matrix of points in the complex plane. Inside that, there is a do-while loop that checks if that point belongs to the set. The k value in the code is from where we are going to extract the color of the output. The code is ready to run with many options:

- "-d" to specify that we want to display the set;
- "-h" to compute a histogram of the set;
- "-i" to specify the number of iterations;
- "-o" to save to disk the histogram to compare with later parallel executions;
- "-c" to set the center of the displayed set;
- "-help" to check the other parameters.

```
// Calculate points and generate appropriate output
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            //initialization lines
            ...

            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;

            //output and siplay lines
            ...
        }
    }
```

Figure 2: Essential Mandelbrot code

In order to see the dependencies that our code has, we will analyze *mandel-tar.c*. The objective of doing this analysis is to observe possible dependencies that may affect the parallelism of the code.

Let's define a task for each iteration of *row* loop.

We are using `-w 8` as the size for the Mandelbrot image in order to generate a reasonable task graph in a reasonable execution time. So let's see the graph:

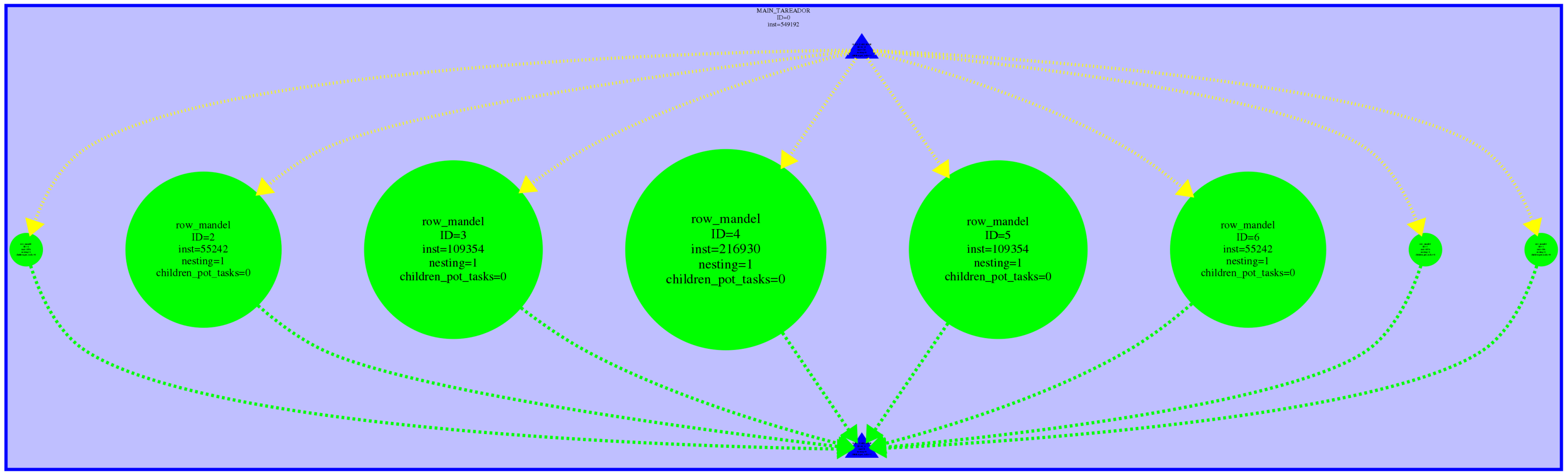


Figure 3: Task row loop decomposition with Tareador ./run-tareador.sh mandel-tar

The first thing we see is the size difference between nodes. It's because of the threshold. Each time a point in the plane exceeds the threshold, we break the iteration and we jump to the next row point. Small nodes in the graph are those where most iterations are cancelled immediately or before the maximum iterations we have set. If a point does not pass the threshold in 10,000 iterations, we will assume it never will. Even if we have an imbalance, we can still parallelize all tasks without dependencies.

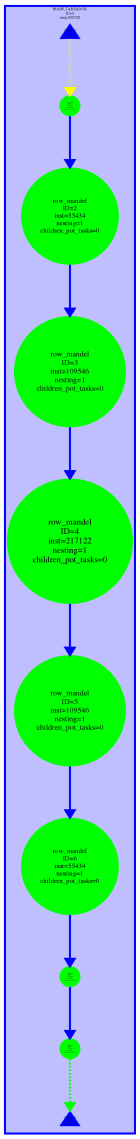


Figure 4: Task row loop decomposition with Tareador ./run-tareador.sh mandel-tar -d

We can observe that the task dependence graph above (Figure 5) is sequential. The weight of the first task is light, then the weight of the tasks increases to reach the middle and thereafter starts decreasing the size. Now, in the code there is a conditional "if (output2display)" that returns true, and it has the work of painting the set to the screen. For protecting this, we will use `pragma_omp_critical`.

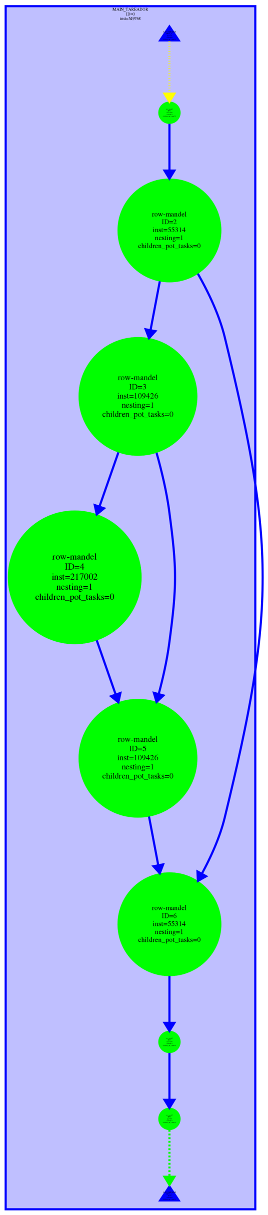


Figure 5: Task row loop decomposition with Tareador ./run-tareador.sh mandel-tar -h

With the option `-h` the code activates the option to compute a histogram. Each chain represents an iteration of the row loop, but now there are new dependences between nodes due to a vector. For solving it, we can add a `pragma_omp_atomic` to the addition.

## 1.2 Point task decomposition strategy

If we change to the point task decomposition strategy, we can see that the graph with no options is embarrassingly parallel. There are some tasks that have a huge weight, whereas the others tasks don't. There are more tasks than the row task decomposition because now there is a finer granularity.

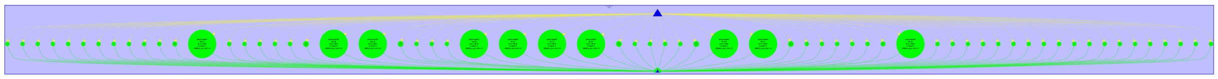


Figure 6: Task point loop decomposition with Tareador ./run-tareador.sh mandel-tar

If we put the option -d, all the tasks that we have seen before are now sequentially executed. As we have seen that in row loop decomposition there are tasks that are bigger than others (Figure 4), with point decomposition (Figure 7) also there are some of them bigger than the others.

*Here we would have liked to insert Figure 7, but it is too big so you can see it on the next page.*

With the option `-h` for computing the histogram of the set, we can observe in Figure 8 that is similar to Figure 5 but as there are more tasks there are more dependences.

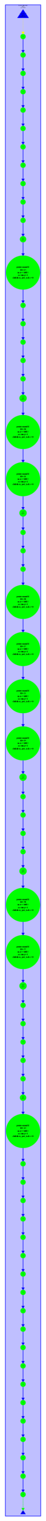


Figure 7: Task point loop decomposition with Tareador ./run-tareador.sh mandel-tar -d

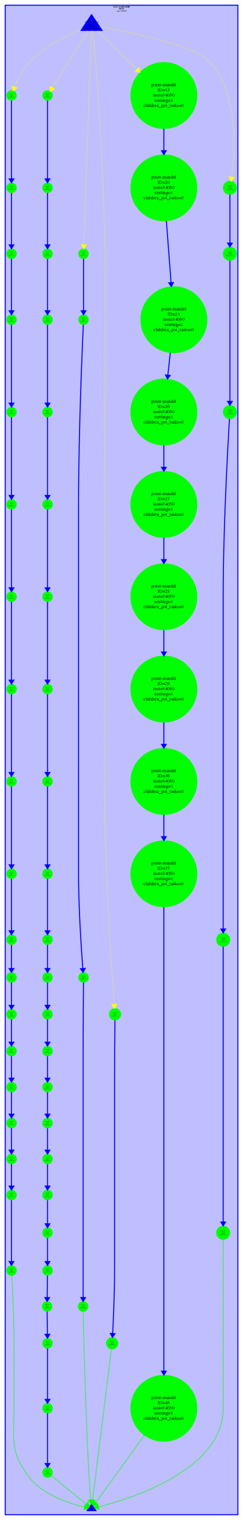


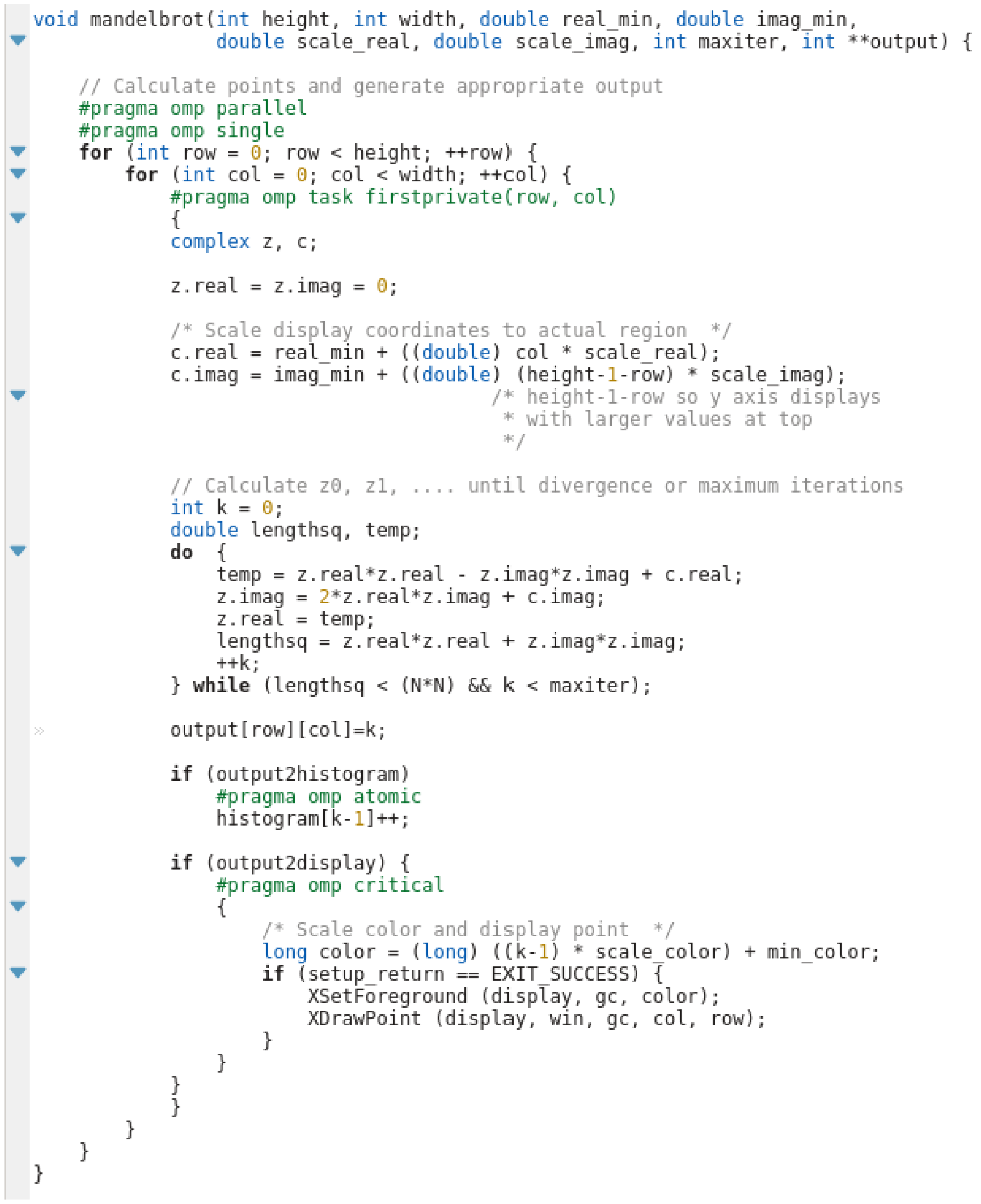
Figure 8: Task point loop decomposition with Tareador ./run-tareador.sh mandel-tar -h

## 2 Implementation in *OpenMP* and performance analysis

### 2.1 *Point* decomposition in *OpenMP*

#### 2.1.1 *Point* strategy implementation using `task`

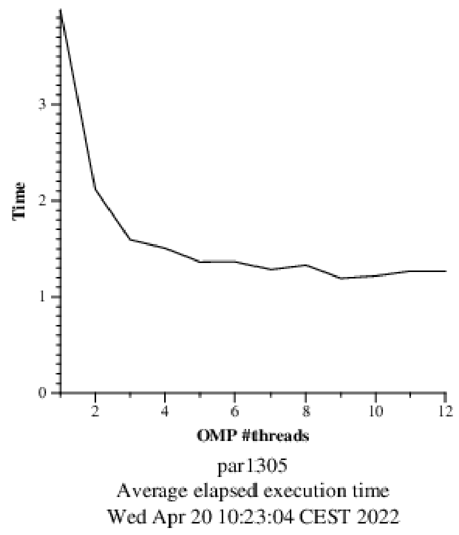
Here we can see the modifications we've done to the `mandel-omp.c`, adding task the OpenMP directives to make sure that all dependencies are honoured:



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

Figure 9: The edition of the initial task version in mandel-omp.c.

Submitting with the `submit-strong-omp.sh` script with `sbatch` we could observe that, if we increase the number of threads from 1 to 12, the average elapsed execution time decreases a lot with 3 threads and after, only slightly decreases. The speed-up obtained and the scalability is quite poor.



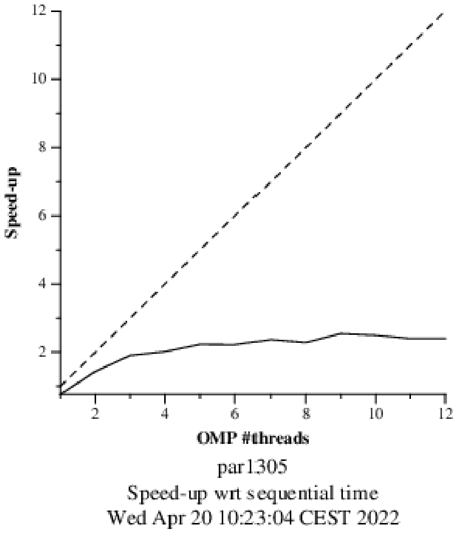


Figure 10: Speed-up and time elapsed (in seconds) point task decomposition plots

We are going to use Paraver to inspect the trace corresponding to the execution with 8 threads. As seen in Figure 9, there are tasks that have to execute more instructions than other tasks.

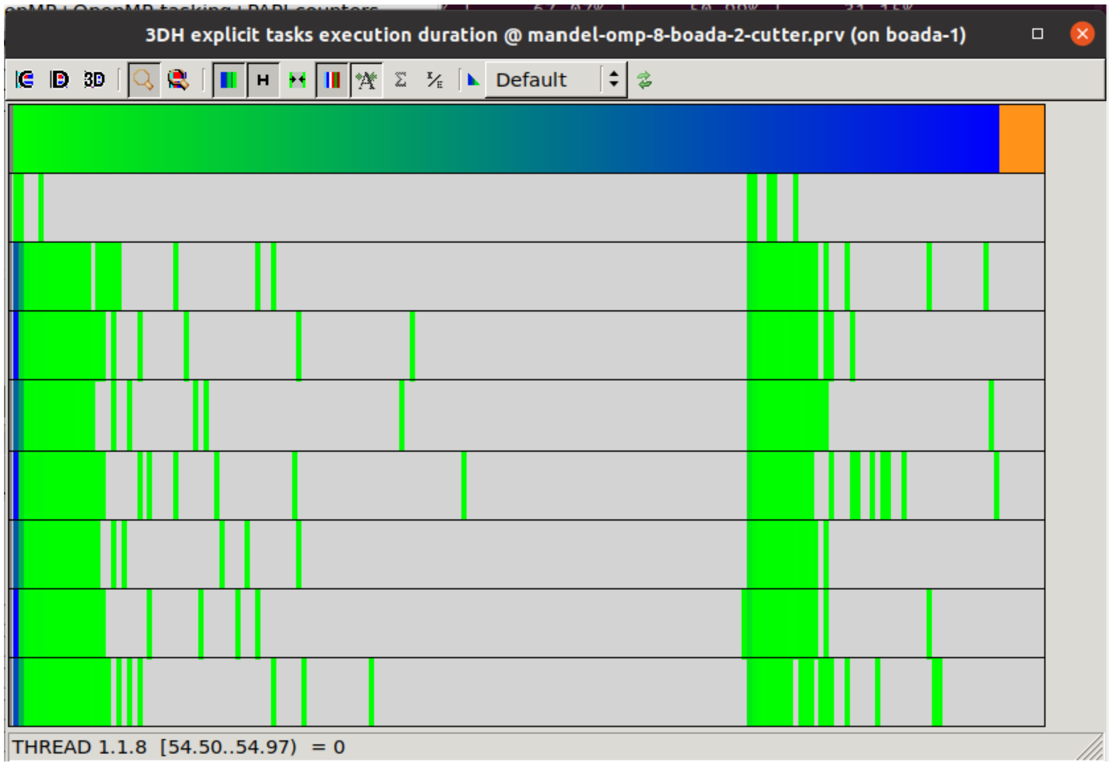
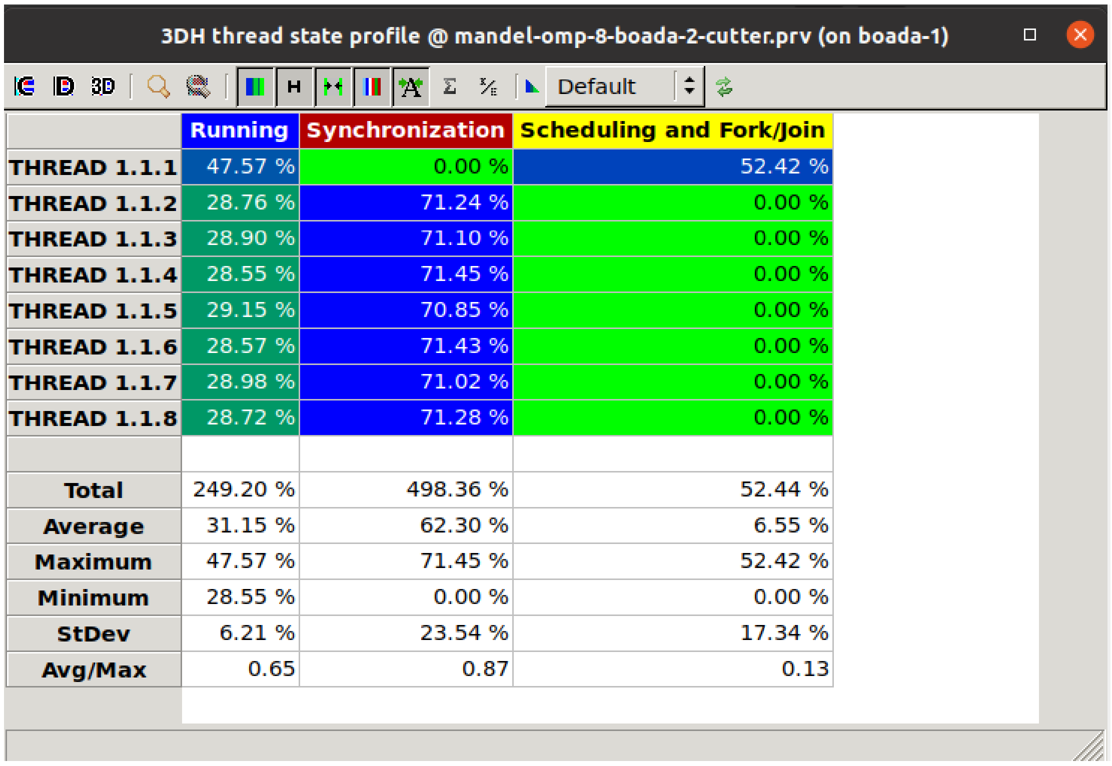

3DH explicit tasks execution duration @ mandel-omp-8-boada-2-cutter.prv (on boada-1)
Default
THREAD 1.1.8 [54.50..54.97) = 0

Figure 11: Explicit task duration with 8 threads, task point decomposition

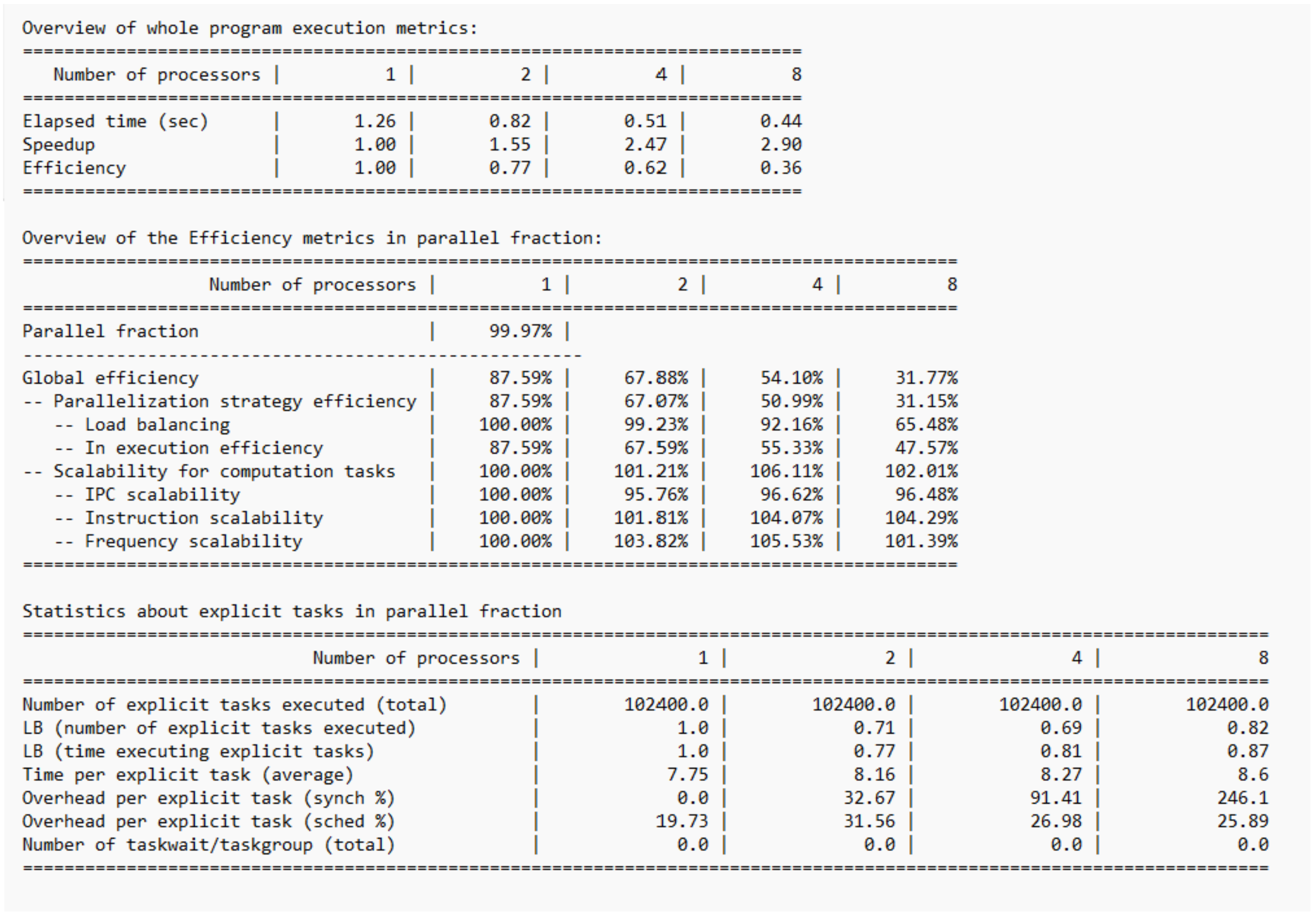

3DH thread state profile @ mandel-omp-8-boada-2-cutter.prv (on boada-1)
Default

| | Running | Synchronization | Scheduling and Fork/Join |
|---|---|---|---|
| THREAD 1.1.1 | 47.57 % | 0.00 % | 52.42 % |
| THREAD 1.1.2 | 28.76 % | 71.24 % | 0.00 % |
| THREAD 1.1.3 | 28.90 % | 71.10 % | 0.00 % |
| THREAD 1.1.4 | 28.55 % | 71.45 % | 0.00 % |
| THREAD 1.1.5 | 29.15 % | 70.85 % | 0.00 % |
| THREAD 1.1.6 | 28.57 % | 71.43 % | 0.00 % |
| THREAD 1.1.7 | 28.98 % | 71.02 % | 0.00 % |
| THREAD 1.1.8 | 28.72 % | 71.28 % | 0.00 % |
| | | | |
| Total | 249.20 % | 498.36 % | 52.44 % |
| Average | 31.15 % | 62.30 % | 6.55 % |
| Maximum | 47.57 % | 71.45 % | 52.42 % |
| Minimum | 28.55 % | 0.00 % | 0.00 % |
| StDev | 6.21 % | 23.54 % | 17.34 % |
| Avg/Max | 0.65 | 0.87 | 0.13 |

Figure 12: Thread state profile 8 threads task point decomposition

If we observe the result with `modelfactor.py` with a different smaller Mandelbrot set computed now (320x320 instead of 800x800), the elapsed time decreases and the speed-up increases, but the efficiency is really low and getting worse as the number of processors is increased. The parallelism is the one from Figure 6, with a parallel fraction of almost 100%.

We don't think that the granularity of the tasks is appropriate for this parallelization strategy because there is a lot of overhead due to creating so many tasks.



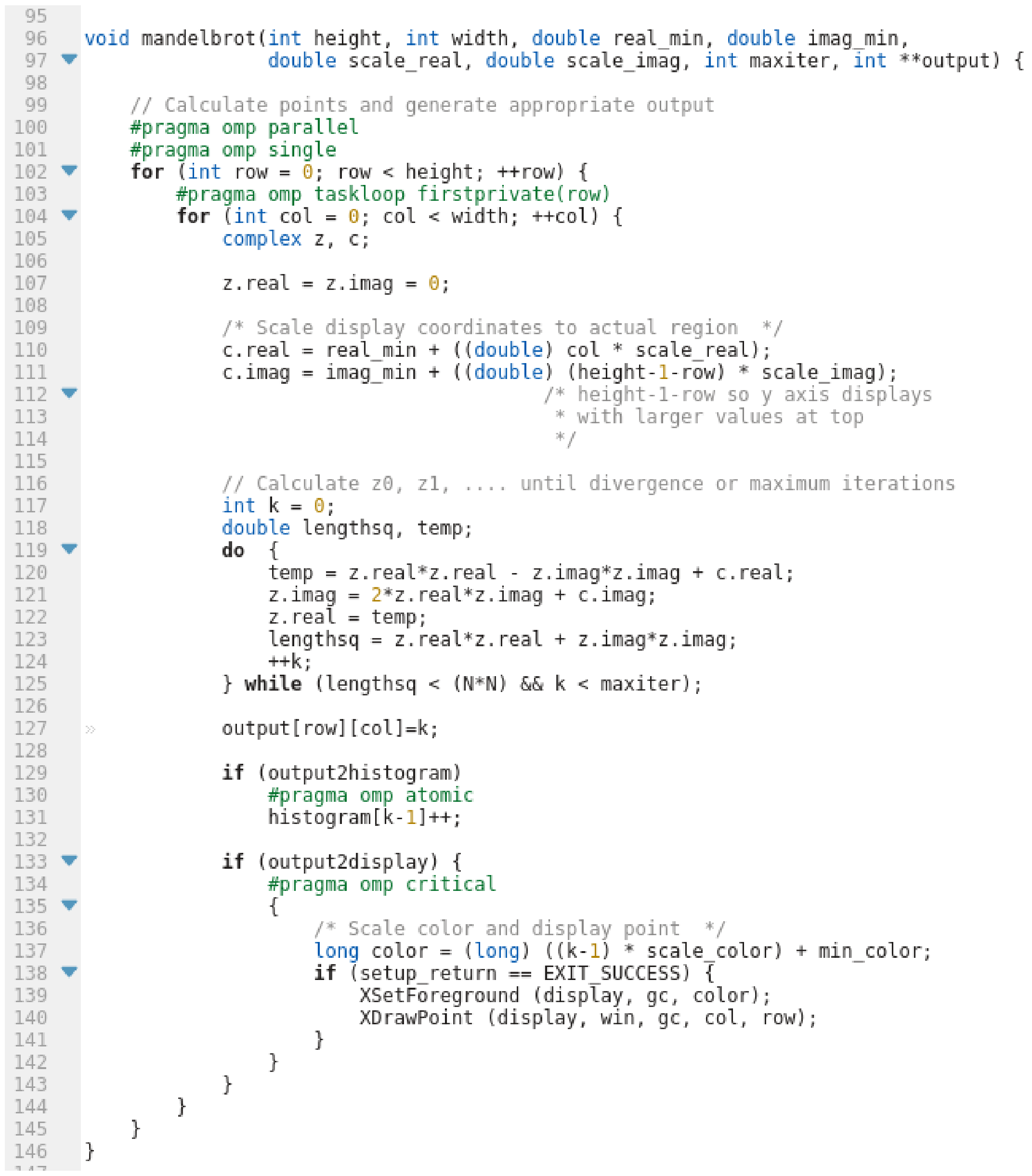
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       1.26 |       0.82 |       0.51 |       0.44
Speedup                 |       1.00 |       1.55 |       2.47 |       2.90
Efficiency              |       1.00 |       0.77 |       0.62 |       0.36
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
=========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
=========================================================================================
Parallel fraction                      |     99.97% |
-----------------------------------------------------
Global efficiency                      |     87.59% |     67.88% |     54.10% |     31.77%
-- Parallelization strategy efficiency |     87.59% |     67.07% |     50.99% |     31.15%
   -- Load balancing                   |    100.00% |     99.23% |     92.16% |     65.48%
   -- In execution efficiency          |     87.59% |     67.59% |     55.33% |     47.57%
-- Scalability for computation tasks   |    100.00% |    101.21% |    106.11% |    102.01%
   -- IPC scalability                  |    100.00% |     95.76% |     96.62% |     96.48%
   -- Instruction scalability          |    100.00% |    101.81% |    104.07% |    104.29%
   -- Frequency scalability            |    100.00% |    103.82% |    105.53% |    101.39%
=========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                              Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)          |          102400.0 |          102400.0 |          102400.0 |          102400.0
LB (number of explicit tasks executed)             |               1.0 |              0.71 |              0.69 |              0.82
LB (time executing explicit tasks)                 |               1.0 |              0.77 |              0.81 |              0.87
Time per explicit task (average)                   |              7.75 |              8.16 |              8.27 |               8.6
Overhead per explicit task (synch %)               |               0.0 |             32.67 |             91.41 |             246.1
Overhead per explicit task (sched %)               |             19.73 |             31.56 |             26.98 |             25.89
Number of taskwait/taskgroup (total)               |               0.0 |               0.0 |               0.0 |               0.0
========================================================================================================================
```

Figure 13: Execution metrics point task decomposition strategy

### 2.1.2 *Point* strategy with granularity control using `taskloop`

The point tasks declaration has caused bad scalability and performance. We will set a task with coarser level of granularity. The previous declaration will be repeated in its entirely.

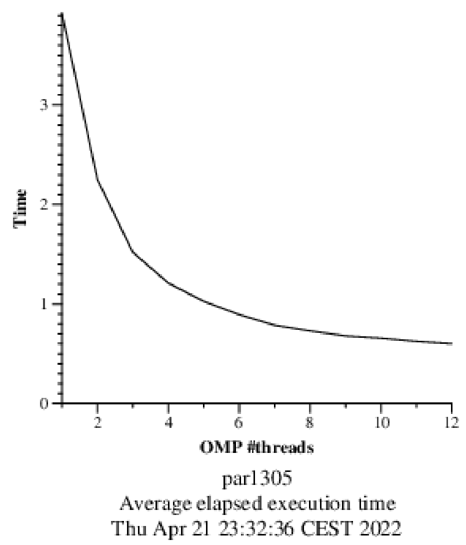
Figure 14 shows how we will modify our code in this case. We have erased the previous tasks declaration, and we have opted for a taskloop. As a result of this clause, OpenMP divides its subsequent loop into bits of a certain size determined by the program.



```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 firstprivate(row)
        for (int col = 0; col < width; ++col) {
            complex z, c;

            z.real = z.imag = 0;

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

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

            output[row][col]=k;

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

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

Figure 14: Speed-up and time elapsed (in seconds) point task decomposition plots

The program takes 3.9 s to be executed with one thread and, with 2 threads, 2.2 s. This is very similar to the previous version. However, if the number of processors are increased, this version has a better performance: the execution time lower and the speed-up higher.



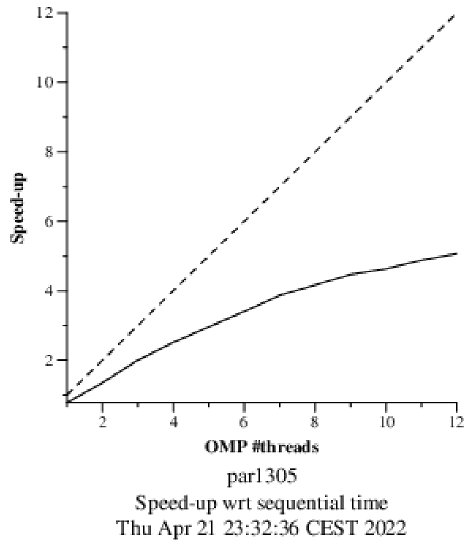
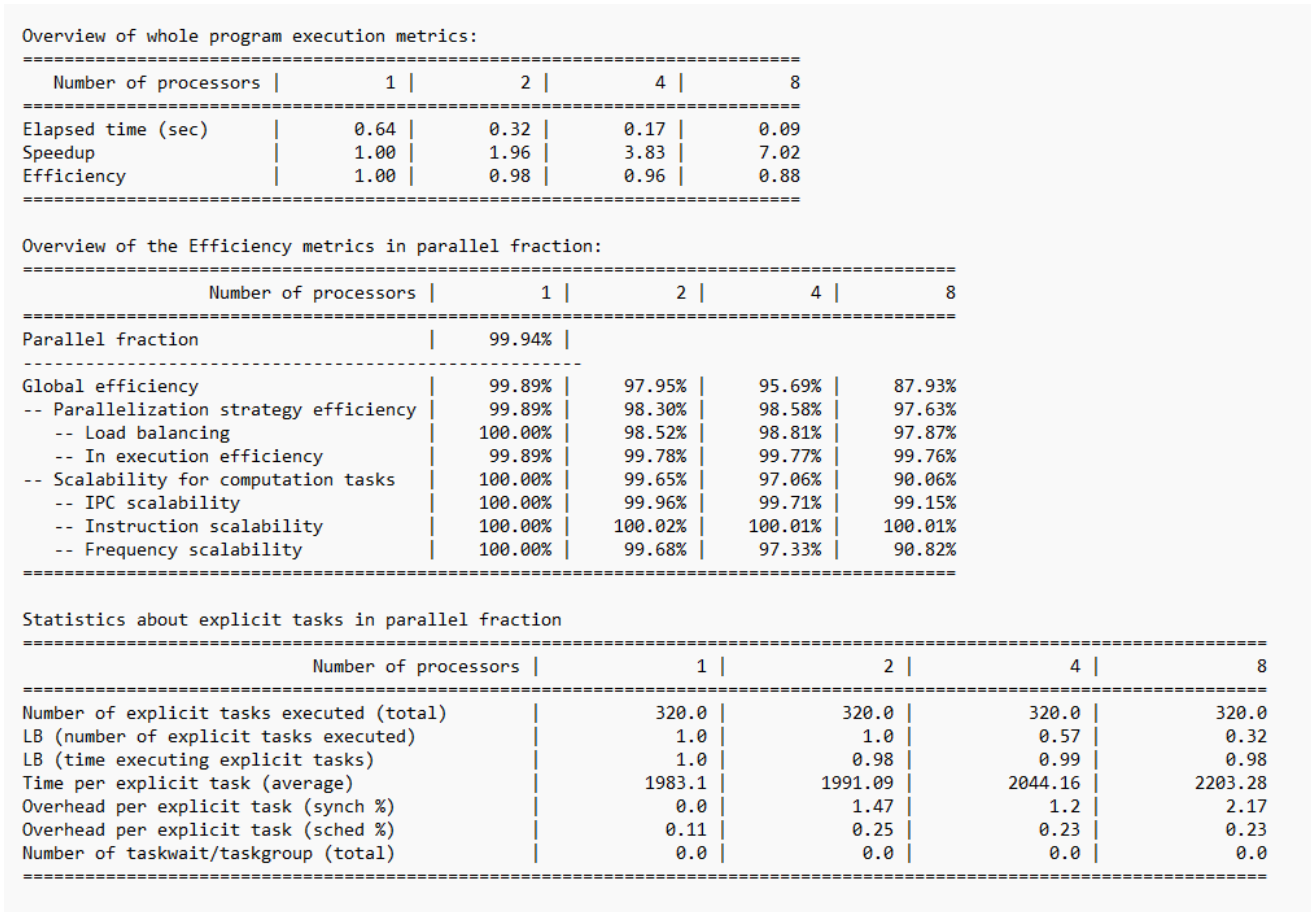


Figure 15: Speed-up and time elapsed (in seconds) point task decomposition plots

With the new task definition, we have reduced the execution time by a considerable amount. When executing one thread we had similar time as when executing sequentially.



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.64 |       0.32 |       0.17 |       0.09
Speedup                 |       1.00 |       1.96 |       3.83 |       7.02
Efficiency              |       1.00 |       0.98 |       0.96 |       0.88
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     99.94% |
------------------------------------------------------
Global efficiency                      |     99.89% |     97.95% |     95.69% |     87.93%
-- Parallelization strategy efficiency |     99.89% |     98.30% |     98.58% |     97.63%
   -- Load balancing                   |    100.00% |     98.52% |     98.81% |     97.87%
   -- In execution efficiency          |     99.89% |     99.78% |     99.77% |     99.76%
-- Scalability for computation tasks   |    100.00% |     99.65% |     97.06% |     90.06%
   -- IPC scalability                  |    100.00% |     99.96% |     99.71% |     99.15%
   -- Instruction scalability          |    100.00% |    100.02% |    100.01% |    100.01%
   -- Frequency scalability            |    100.00% |     99.68% |     97.33% |     90.82%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)        |             320.0 |             320.0 |             320.0 |             320.0
LB (number of explicit tasks executed)           |               1.0 |               1.0 |              0.57 |              0.32
LB (time executing explicit tasks)               |               1.0 |              0.98 |              0.99 |              0.98
Time per explicit task (average)                 |            1983.1 |           1991.09 |           2044.16 |           2203.28
Overhead per explicit task (synch %)             |               0.0 |              1.47 |               1.2 |              2.17
Overhead per explicit task (sched %)             |              0.11 |              0.25 |              0.23 |              0.23
Number of taskwait/taskgroup (total)             |               0.0 |               0.0 |               0.0 |               0.0
========================================================================================================================
```

Figure 16: Execution metrics point taskloop decomposition strategy

Observing the Figure 18, we can ensure that there's an improvement in what it comes to the synchronization and running time compared to the previous version.

Furthermore, in Figure 17 we can see that the execution time of the explicit tasks has been reduced.

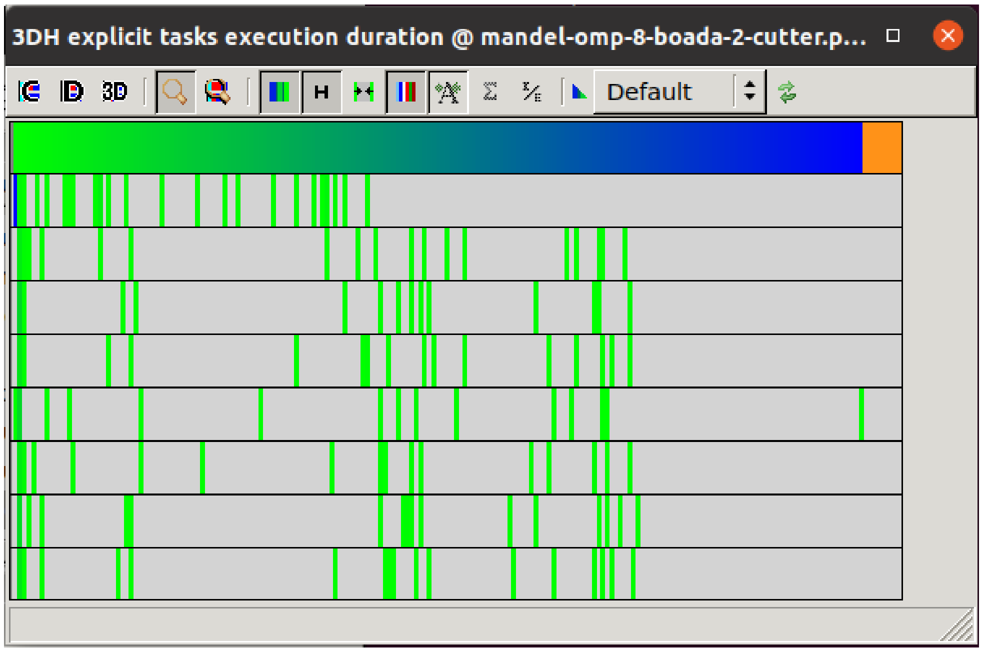
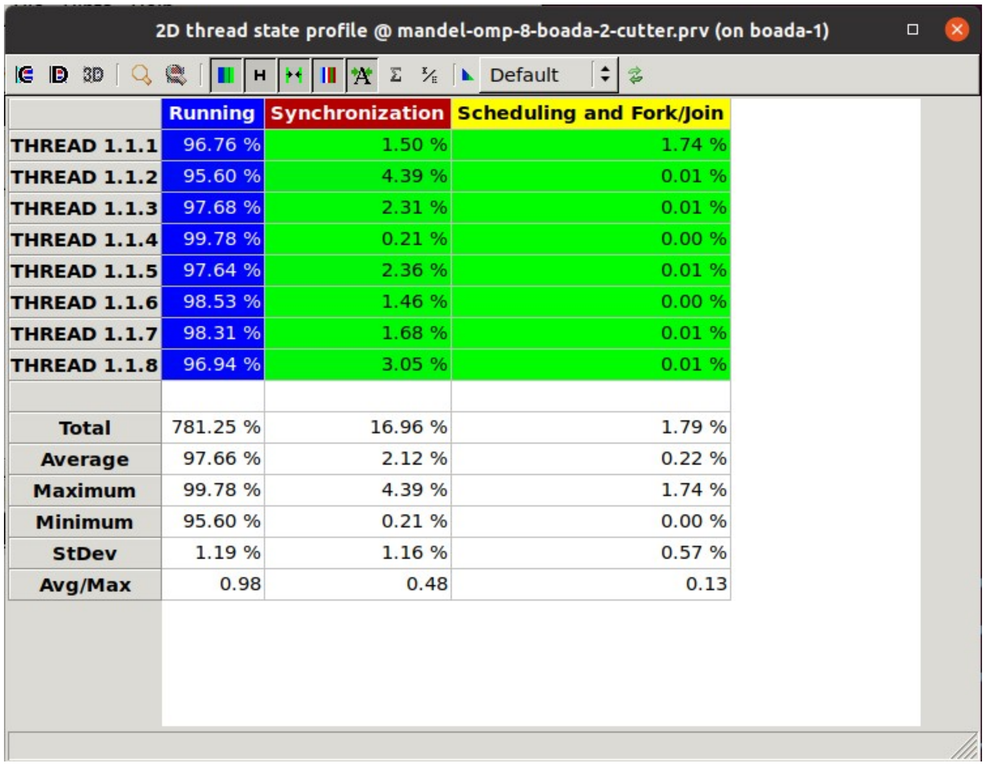


Figure 17: Explicit task duration with 8 threads, point taskloop decomposition

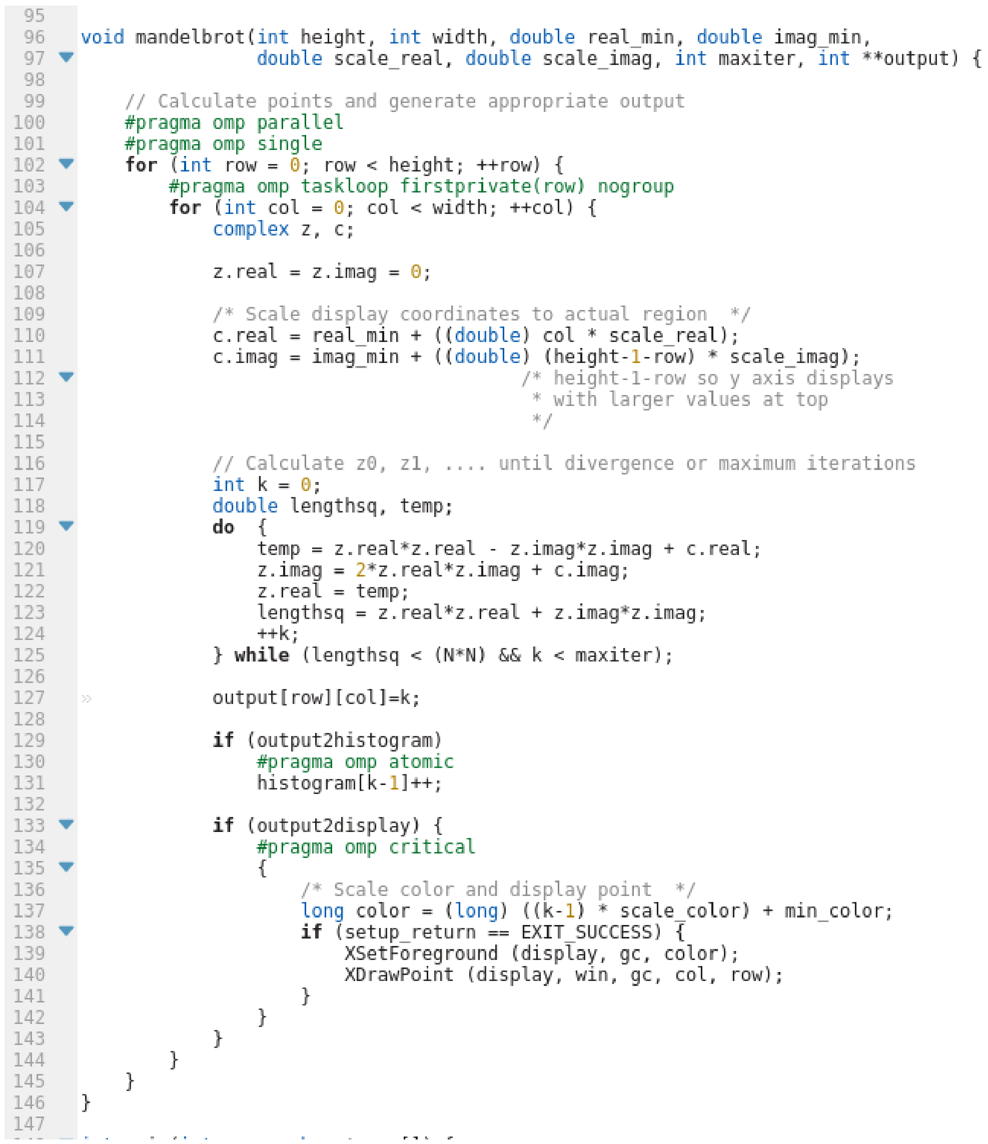

2D thread state profile @ mandel-omp-8-boada-2-cutter.prv (on boada-1)

|  | Running | Synchronization | Scheduling and Fork/Join |
|---|---|---|---|
| THREAD 1.1.1 | 96.76 % | 1.50 % | 1.74 % |
| THREAD 1.1.2 | 95.60 % | 4.39 % | 0.01 % |
| THREAD 1.1.3 | 97.68 % | 2.31 % | 0.01 % |
| THREAD 1.1.4 | 99.78 % | 0.21 % | 0.00 % |
| THREAD 1.1.5 | 97.64 % | 2.36 % | 0.01 % |
| THREAD 1.1.6 | 98.53 % | 1.46 % | 0.00 % |
| THREAD 1.1.7 | 98.31 % | 1.68 % | 0.01 % |
| THREAD 1.1.8 | 96.94 % | 3.05 % | 0.01 % |
|  |  |  |  |
| Total | 781.25 % | 16.96 % | 1.79 % |
| Average | 97.66 % | 2.12 % | 0.22 % |
| Maximum | 99.78 % | 4.39 % | 1.74 % |
| Minimum | 95.60 % | 0.21 % | 0.00 % |
| StDev | 1.19 % | 1.16 % | 0.57 % |
| Avg/Max | 0.98 | 0.48 | 0.13 |

Figure 18: Thread state profile 8 threads point taskloop decomposition

### 2.1.3 *Point* strategy with granularity control using `taskloop nogroup`

We will add the clause nogroup to the task declaration to decouple each task from the batch it belongs to, so that, as soon as that thread finishes the task, it can continue with tasks belonging to other creation batches.

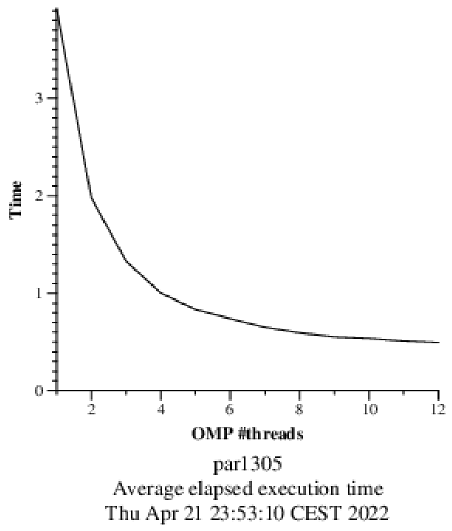


```c

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

```

Figure 19: The edition of the taskloop nogroup version in mandel-omp.c.

This change has resulted in even faster time and speedup due to the elimination of the restriction of waiting until all tasks of a batch have completed.

It is possible to have to wait until a particular thread has completed the 10.000 iterations per pixel before proceeding to the next batch of tasks. It is because all except one of the tasks have four points that do not belong to the set, while the remaining task has four points that do belong. The nogroup clause has eliminated all this wasted time. Fig. 22 give us an idea of creation and the time execution of explicit tasks and fig. 23 shows the reduction of synchronization.



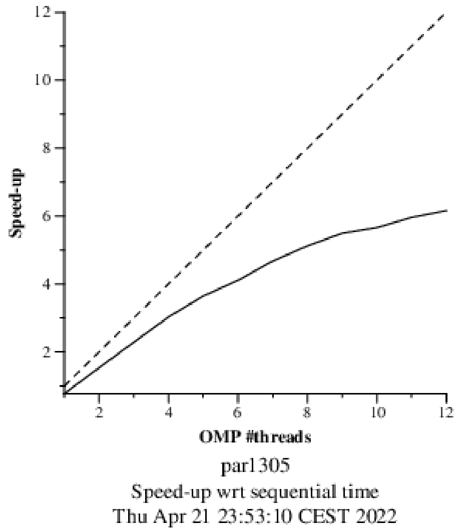
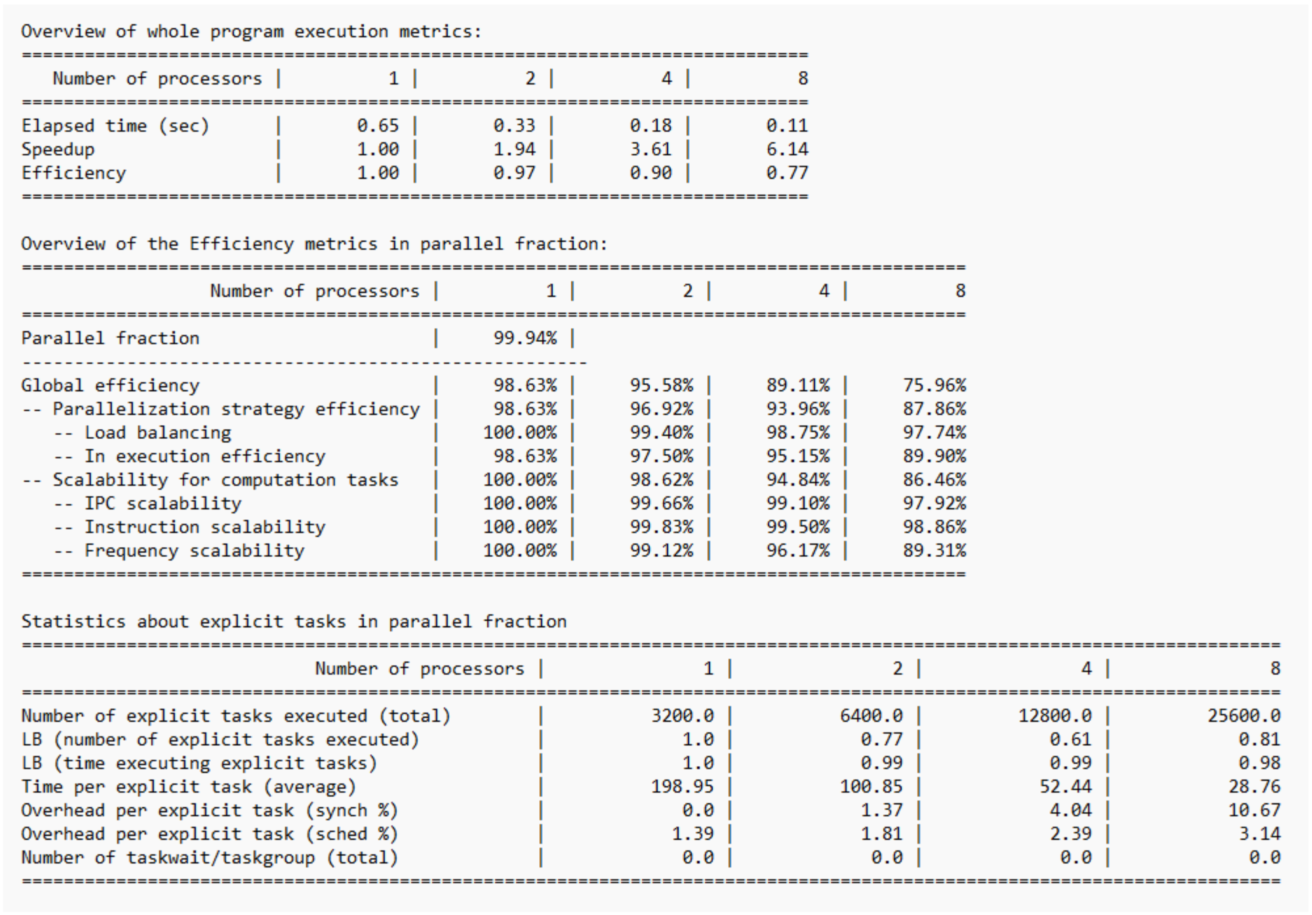


Figure 20: Speed-up and time elapsed (in seconds) taskloop nogroup point decomposition plots



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.65 |       0.33 |       0.18 |       0.11
Speedup                 |       1.00 |       1.94 |       3.61 |       6.14
Efficiency              |       1.00 |       0.97 |       0.90 |       0.77
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     99.94% |
------------------------------------------------------
Global efficiency                      |     98.63% |     95.58% |     89.11% |     75.96%
-- Parallelization strategy efficiency |     98.63% |     96.92% |     93.96% |     87.86%
   -- Load balancing                   |    100.00% |     99.40% |     98.75% |     97.74%
   -- In execution efficiency          |     98.63% |     97.50% |     95.15% |     89.90%
-- Scalability for computation tasks   |    100.00% |     98.62% |     94.84% |     86.46%
   -- IPC scalability                  |    100.00% |     99.66% |     99.10% |     97.92%
   -- Instruction scalability          |    100.00% |     99.83% |     99.50% |     98.86%
   -- Frequency scalability            |    100.00% |     99.12% |     96.17% |     89.31%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)        |            3200.0 |            6400.0 |           12800.0 |           25600.0
LB (number of explicit tasks executed)           |               1.0 |              0.77 |              0.61 |              0.81
LB (time executing explicit tasks)               |               1.0 |              0.99 |              0.99 |              0.98
Time per explicit task (average)                 |            198.95 |            100.85 |             52.44 |             28.76
Overhead per explicit task (synch %)             |               0.0 |              1.37 |              4.04 |             10.67
Overhead per explicit task (sched %)             |              1.39 |              1.81 |              2.39 |              3.14
Number of taskwait/taskgroup (total)             |               0.0 |               0.0 |               0.0 |               0.0
========================================================================================================================
```

Figure 21: Execution metrics point taskloop nogroup decomposition strategy

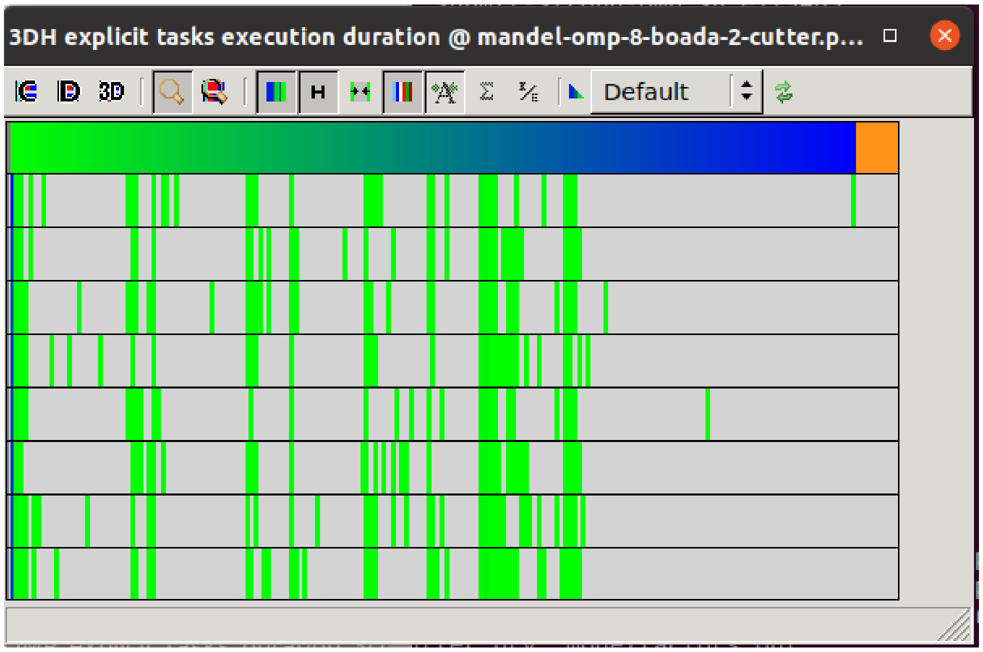
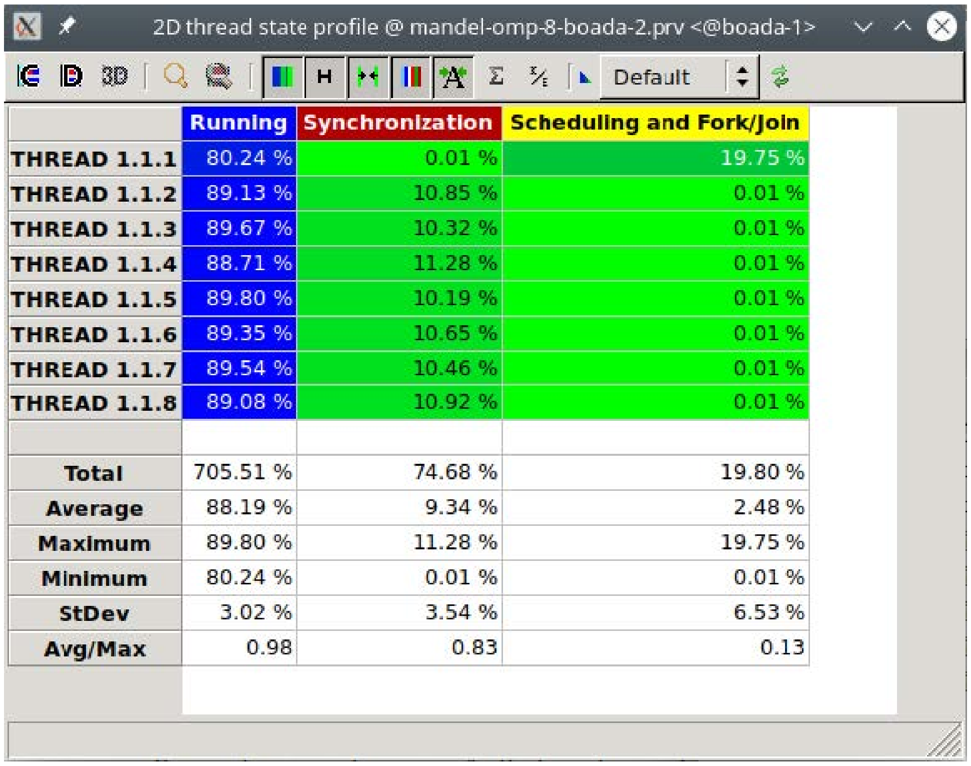

3DH explicit tasks execution duration @ mandel-omp-8-boada-2-cutter.p...
Default

Figure 22: Explicit task duration with 8 threads, point taskloop nogroup decomposition


2D thread state profile @ mandel-omp-8-boada-2.prv <@boada-1>
Default

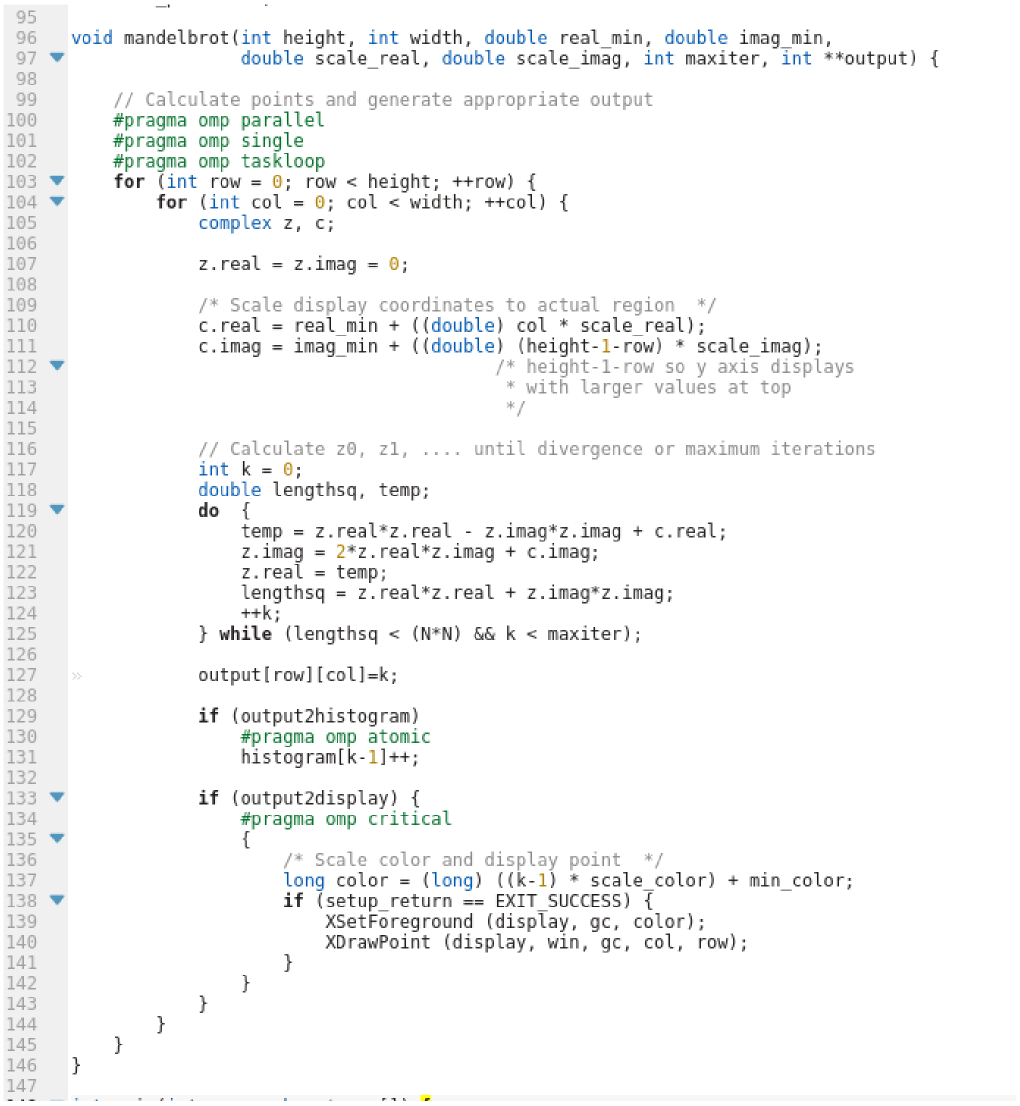
| | Running | Synchronization | Scheduling and Fork/Join |
|---|---|---|---|
| THREAD 1.1.1 | 80.24 % | 0.01 % | 19.75 % |
| THREAD 1.1.2 | 89.13 % | 10.85 % | 0.01 % |
| THREAD 1.1.3 | 89.67 % | 10.32 % | 0.01 % |
| THREAD 1.1.4 | 88.71 % | 11.28 % | 0.01 % |
| THREAD 1.1.5 | 89.80 % | 10.19 % | 0.01 % |
| THREAD 1.1.6 | 89.35 % | 10.65 % | 0.01 % |
| THREAD 1.1.7 | 89.54 % | 10.46 % | 0.01 % |
| THREAD 1.1.8 | 89.08 % | 10.92 % | 0.01 % |
| | | | |
| Total | 705.51 % | 74.68 % | 19.80 % |
| Average | 88.19 % | 9.34 % | 2.48 % |
| Maximum | 89.80 % | 11.28 % | 19.75 % |
| Minimum | 80.24 % | 0.01 % | 0.01 % |
| StDev | 3.02 % | 3.54 % | 6.53 % |
| Avg/Max | 0.98 | 0.83 | 0.13 |

Figure 23: Thread state profile 8 threads point taskloop nogroup decomposition

Adding the nogroup creates a little improvement because it makes all threads stay more time in execution state instead of in syncronization state. We can see on figures 20 and 21 that the improvement compared to the taskloop without nogroup is not so significantly.

## 2.2 *Row* decomposition in *OpenMP*

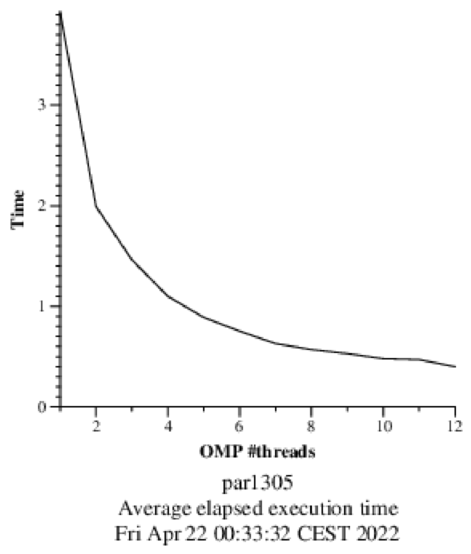
For our final exploration of the parallelism of the computation of the Mandelbrot set, we will set the definition of each task using the clause taskloop, but this time we will assign the taskloop to the row loop instead of the col loop (Figure 24).



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

Figure 24: The edition of the taskloop row version in mandel-omp.c.

As the variables row and col have not yet been created and initialized, we will not need the firstprivate() clause nor private().



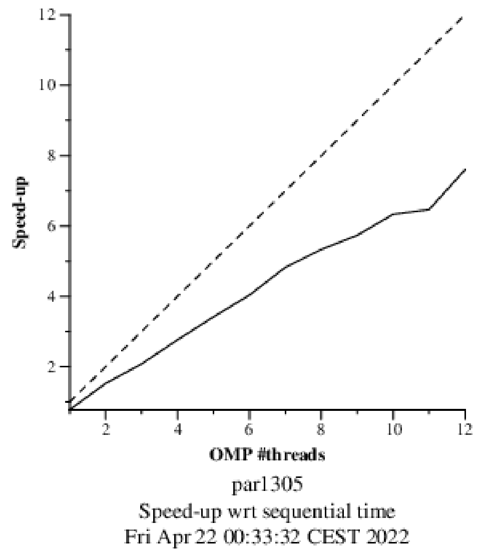
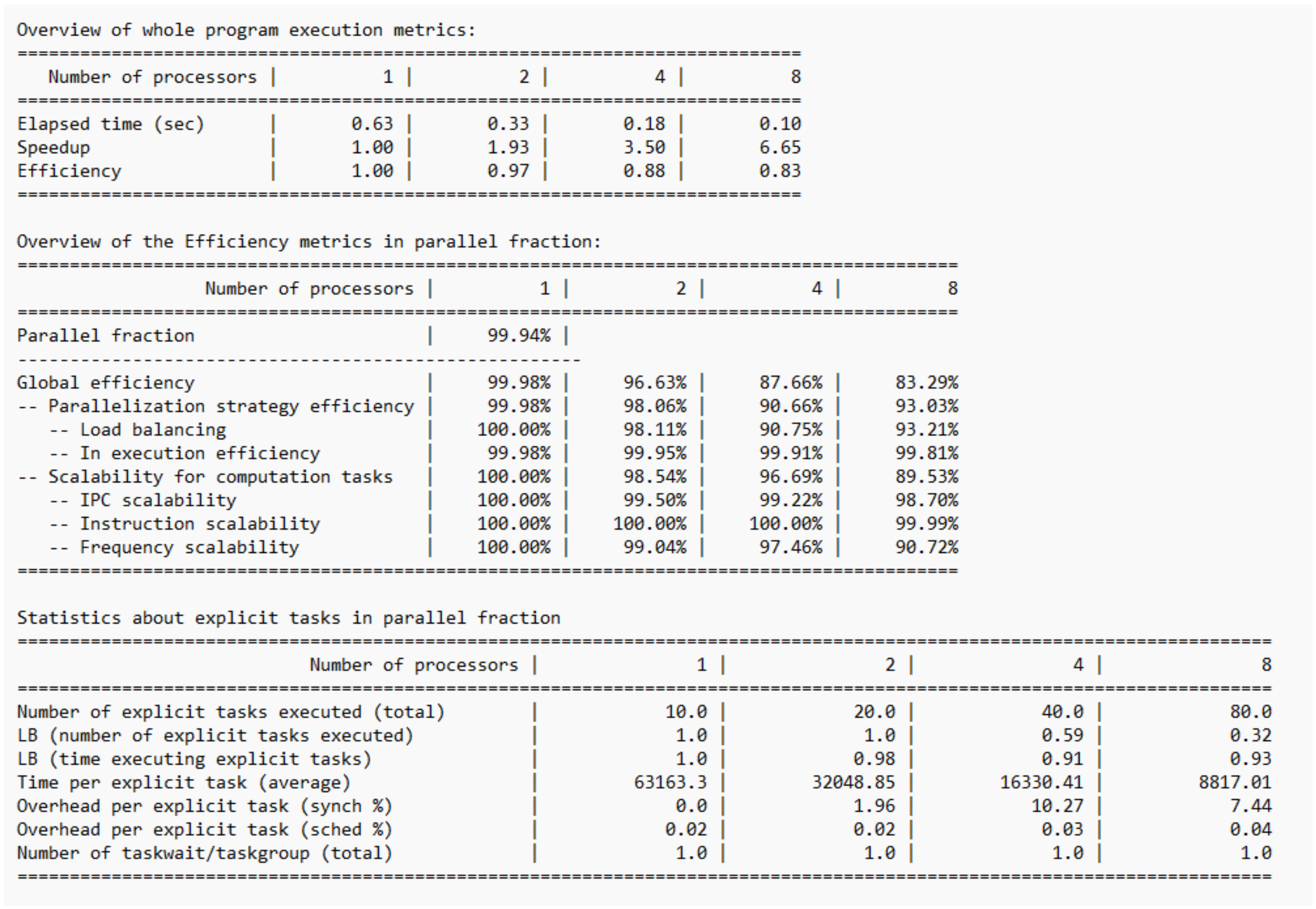


Figure 25: Speed-up and time elapsed (in seconds) taskloop row decomposition plots

As we can see, the execution times are nearly identical to the ones with the point-nogroup task declaration, so we can conclude that these two types of task definition are equally effective at parallelizing this problem. Check figure 25.

Figure 28 shows more synchronization time than the other version. However, the scalability plot (Figure 25) is the best of all versions because if we add more threads, the execution time will decrease and the speed-up will increase.

Due to the similarity of the graphs and results of other strategies obtained (as what we did at point-taskloop-nogroup declaration), we decided that the nogroup clause has no effect on execution speed and that parallelization is unnecessary.



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.63 |       0.33 |       0.18 |       0.10
Speedup                 |       1.00 |       1.93 |       3.50 |       6.65
Efficiency              |       1.00 |       0.97 |       0.88 |       0.83
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     99.94% |
------------------------------------------------------
Global efficiency                      |     99.98% |     96.63% |     87.66% |     83.29%
-- Parallelization strategy efficiency |     99.98% |     98.06% |     90.66% |     93.03%
   -- Load balancing                   |    100.00% |     98.11% |     90.75% |     93.21%
   -- In execution efficiency          |     99.98% |     99.95% |     99.91% |     99.81%
-- Scalability for computation tasks   |    100.00% |     98.54% |     96.69% |     89.53%
   -- IPC scalability                  |    100.00% |     99.50% |     99.22% |     98.70%
   -- Instruction scalability          |    100.00% |    100.00% |    100.00% |     99.99%
   -- Frequency scalability            |    100.00% |     99.04% |     97.46% |     90.72%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                1 |                2 |                4 |                8
========================================================================================================================
Number of explicit tasks executed (total)        |             10.0 |             20.0 |             40.0 |             80.0
LB (number of explicit tasks executed)           |              1.0 |              1.0 |             0.59 |             0.32
LB (time executing explicit tasks)               |              1.0 |             0.98 |             0.91 |             0.93
Time per explicit task (average)                 |          63163.3 |         32048.85 |         16330.41 |          8817.01
Overhead per explicit task (synch %)             |              0.0 |             1.96 |            10.27 |             7.44
Overhead per explicit task (sched %)             |             0.02 |             0.02 |             0.03 |             0.04
Number of taskwait/taskgroup (total)             |              1.0 |              1.0 |              1.0 |              1.0
========================================================================================================================
```

Figure 26: Execution metrics point taskloop row decomposition strategy

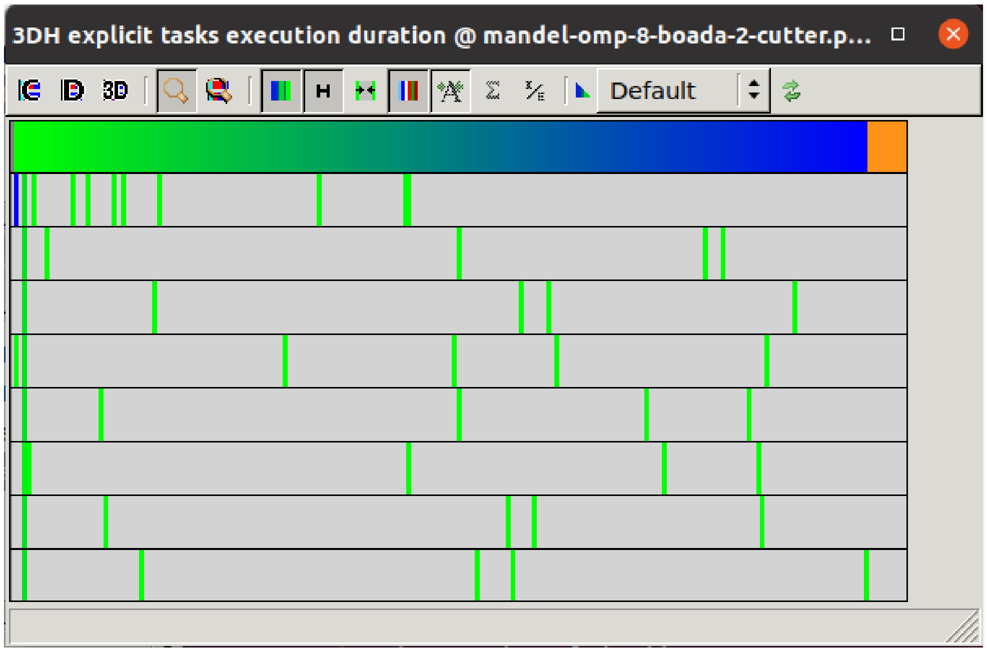
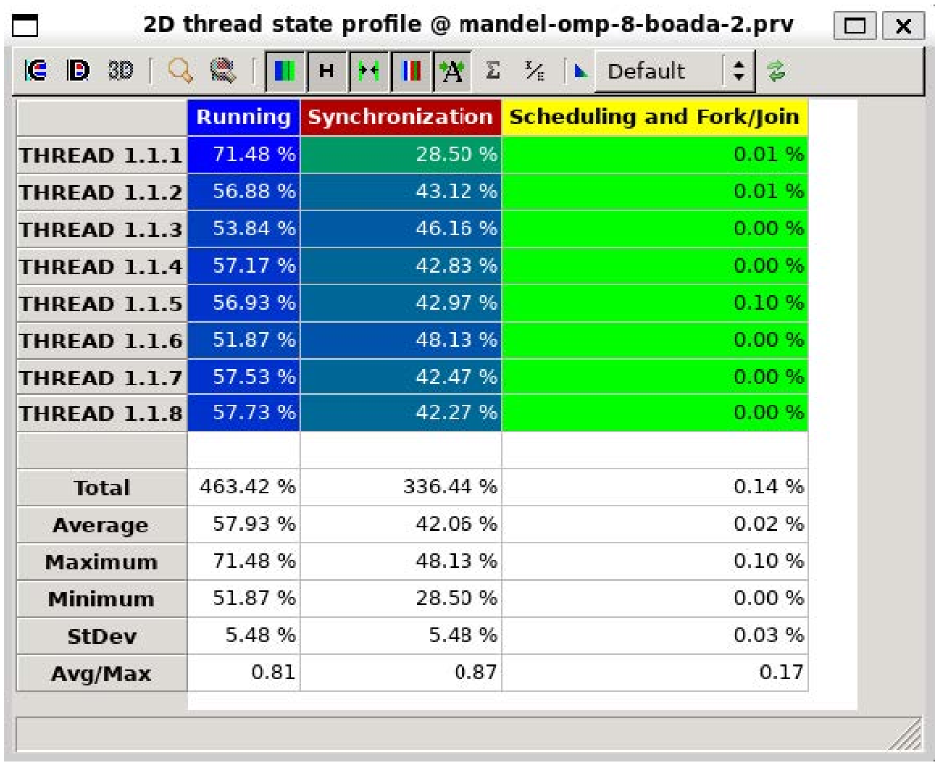

3DH explicit tasks execution duration @ mandel-omp-8-boada-2-cutter.p...
Default

Figure 27: Explicit task duration with 8 threads, row taskloop decomposition



| | Running | Synchronization | Scheduling and Fork/Join |
|---|---|---|---|
| THREAD 1.1.1 | 71.48 % | 28.50 % | 0.01 % |
| THREAD 1.1.2 | 56.88 % | 43.12 % | 0.01 % |
| THREAD 1.1.3 | 53.84 % | 46.16 % | 0.00 % |
| THREAD 1.1.4 | 57.17 % | 42.83 % | 0.00 % |
| THREAD 1.1.5 | 56.93 % | 42.97 % | 0.10 % |
| THREAD 1.1.6 | 51.87 % | 48.13 % | 0.00 % |
| THREAD 1.1.7 | 57.53 % | 42.47 % | 0.00 % |
| THREAD 1.1.8 | 57.73 % | 42.27 % | 0.00 % |
| | | | |
| Total | 463.42 % | 336.44 % | 0.14 % |
| Average | 57.93 % | 42.06 % | 0.02 % |
| Maximum | 71.48 % | 48.13 % | 0.10 % |
| Minimum | 51.87 % | 28.50 % | 0.00 % |
| StDev | 5.48 % | 5.48 % | 0.03 % |
| Avg/Max | 0.81 | 0.87 | 0.17 |

Figure 28: Thread state profile 8 threads point taskloop row decomposition

## Final conclusions

In conclusion, we can say that Mandelbrot's set is clearly embarrassingly parallel:

1. an image with n x m pixels evaluates n x m recurrences
2. each series can be evaluated independently
3. no particular order is required (row by row, column by column, ...)

Parallelizing a program can be done in multiple ways. We have seen some of them and the programmer needs to experiment with all the variables to get the best configuration for the desired results.

Mandelbrot set is an instance in which the point-taskloop-nogroup and row task definitions easily overthrow the point-taskloop-nogroup and row task strategies.

Finally, to conclude, it is important to keep two things in mind:

1. Increasing the number of tasks improves the performance of the row strategy, since we depart from a very coarse level of granularity (1 taskloop).
2. The point strategy performs worse with an increasing number of tasks since we depart from a very fine granularity (a lot of taskloops).