Task decomposition analysis for the Mandelbrot set computation

Ismael Douha Prieto, Eloi Cruz Harillo - Group 1305

Q1 2019/20

1 Introduction

In this paper we will see how create a Mandelbrot Set. This image is created with the succession $z_{n+1} = z_n^2 + c$, where c is each point of the window. For every c, we compute the succession until reaches the divergence point, when we paint the pixel with the corresponding colour and we start the next point computation.

In the figure 1 we can see the graphic representation of the Mandelbrot set. This figure is composed by zones of different colours. This colours indicate the the number of iteration that a point needs to reach the divergence point. All points of the same color represent points that reach the divergence whith the same number of iterations. Color white means that the point never reaches the divergence and color black means that the divergence is reached in the first iteration.

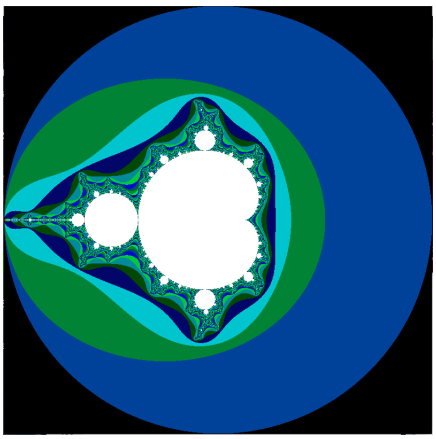


Figure 1: Mandelbrot set

2 Parallelization strategies

In this assignment we saw two differents parallelisation strategies for the Mandelbrot set. These are row (every row is a task) and point decomposition (every combination of row and column is a task).

2.1 Task decomposition and granularity analysis

We know that are this two parallelization strategies thanks to the analysis done, where we focused on task decomposition and granularity.

First of all, we completed the sequential `mandel-tar.c`file in order to analyze the two decomposition strategies.

```c
for (row = 0; row < height; ++row) {
    tareador_start_task("row");
    for (col = 0; col < width; ++col) {
        complex z, c;

        z.real = z.imag = 0;

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

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

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

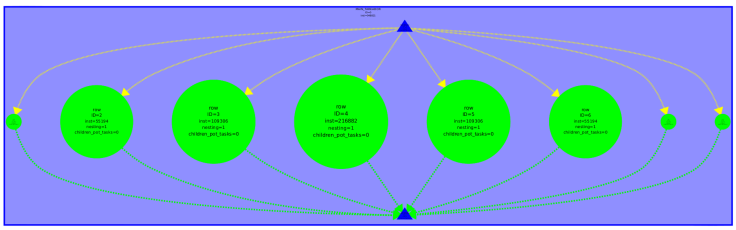
(a) Row decomposition code.

```c
/* Calculate points and save/display */

for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        tareador_start_task("row");
        complex z, c;

        z.real = z.imag = 0;

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

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

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

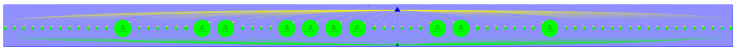
(b) Point decomposition code.

Figure 2: Decomposition strategies.

Once we have doned the code, we proceed to execute it using the two types of execution, with display and without it.



(a) Row decomposition.

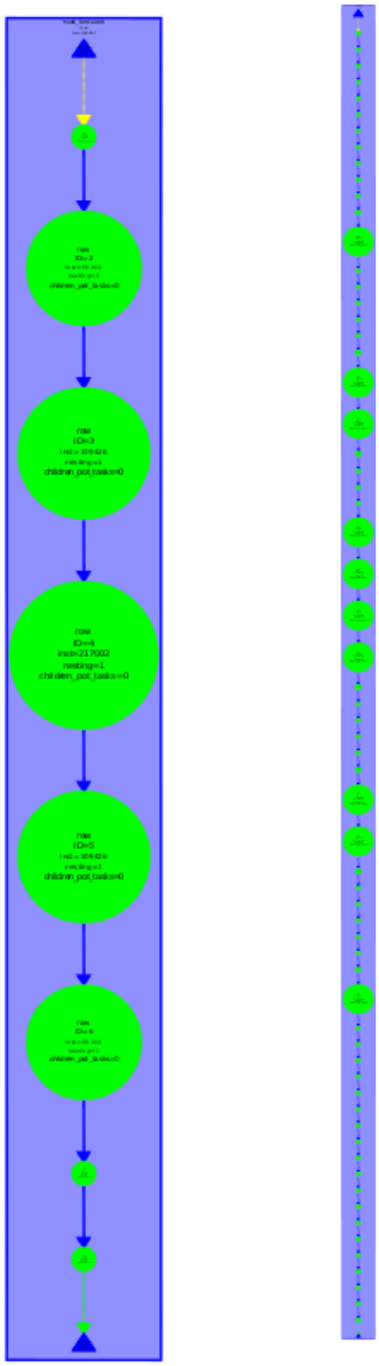


(b) Point decomposition.

Figure 3: Tareador decomposition for no graphical execution.

In this decomposition we observe that both have no dependences and we can experience a maximum parallelization between tasks. Another important characteristic is that the circles that represent the tasks have different radius. The radius indicates the amount of work that the task is doing. The tasks that compute points that will execute a lot of iterations (i.e. white points) are represented by big circles. In the other hand tasks that compute points that will execute few iterations (i.e. black points) are represented by little circles.

If we do the same for the graphical execution we obtain these decompositions.



(a) Row.

(b) Point.

Figure 4: Tareador decomposition for graphical execution.

As we can see, this decomposition is completely serial. This happens because for display the graphics we need to access to a global variable which pertains to `XWindow.`This variable is which paint the pixels in the screen, creating a datarace to access it and paint the pixel. To avoid this datarace, we must add the clause `\#pragmaompcritical,`which ensures that the two functions are executed together.

```
#pragma omp critical
{
    XSetForeground(display_gc, color);
    XDrawPoint(display_win, gc, col, row);
}
```

Analyzing both decomposition we can deduct that point startegy is the best because provides a better parallelism than row decomposition

3 Performance evaluation

Now we will evaluate between differents strategies and implementations the evolutions and which performs better.

3.1 Point decomposition in OpenMP

3.1.1 Point strategy implementation using OpenMPtask

We started parallelizing point decomposition code. In the code provided by Figure 2.1., we need to add the clause "pragma omp critical" to ensure that the instructions which paint the pixels are executed together.

Once we execute `OMP_NUM_THREADS=1` and `./mandeld-omp-i10000`we see that the image generated is correct but execution time in comparison to the sequential version is higher. This happens because the clause "pragma omp critical" generated a great overhead. If we execute the code with 8 threads, the image generated is also correct but the execution time is reduced, in spite of still being high.

```
par1305@boada-1:~/lab3$ ./mandeld -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 3.052829s
```

(a) Sequential.

```
par1305@boada-1:~/lab3$ ./mandeld-omp -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 4.998158s
```

(b) Parallel with 1 thread.

```
par1305@boada-1:~/lab3$ OMP_NUM_THREADS=8
par1305@boada-1:~/lab3$ ./mandeld
mandeld      mandeld-omp  mandeld-tar
par1305@boada-1:~/lab3$ ./mandeld
mandeld      mandeld-omp  mandeld-tar
par1305@boada-1:~/lab3$ ./mandeld-omp -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 3.249845s
```

(c) Parallel with 8 threads.

Figure 5: Execution time for sequential and parallel versions.

Also we verified that the output of all versions are correct. We did this with the linux command `diff`,which shows the differences between two files. If no differences apperas, as is our case, appears nothing.

```
par1305@boada-1:~/lab3$ ./mandel -o

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 1000

Total execution time: 0.318091s
par1305@boada-1:~/lab3$ ./mandel-omp -o

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 1000

^[[A^[[ATotal execution time: 1.915653s
par1305@boada-1:~/lab3$ diff parallel.out serial.out
par1305@boada-1:~/lab3$ OMP_NUM_THREADS=8
par1305@boada-1:~/lab3$ ./mandel-omp -o

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 1000

Total execution time: 1.942832s
par1305@boada-1:~/lab3$ ls -la
total 332012
drwx------  6 par1305 par1305      4096 Oct 31 11:17 .
drwx------ 12 par1305 par1305      4096 Oct 31 10:53 ..
drwxr-xr-x  2 par1305 par1305      4096 Oct 25 11:42 codigos_entregables
-rwxr-xr-x  1 par1305 par1305       979 Sep 26  2018 extrae.xml
drwxr-x--x  2 par1305 par1305      4096 Oct 25 10:24 logs
-rwxr-xr-x  1 par1305 par1305      1186 Mar 26  2019 Makefile
-rwxr-xr-x  1 par1305 par1305     32952 Oct 25 10:24 mandel
-rwxr-xr-x  1 par1305 par1305      4390 Sep 26  2018 mandelbrot-gui.h
-rwxr-xr-x  1 par1305 par1305     51528 Oct 25 10:24 mandeld
-rwxr-xr-x  1 par1305 par1305     61800 Oct 31 10:51 mandeld-omp
-rwxr-xr-x  1 par1305 par1305   1150984 Oct 31 10:48 mandeld-tar
-rwxr-xr-x  1 par1305 par1305     41488 Oct 31 10:51 mandel-omp
-rwxr-xr-x  1 par1305 par1305      8702 Oct 31 10:51 mandel-omp.c
-rwxr-xr-x  1 par1305 par1305      8533 Sep 26  2018 mandel-seq.c
-rwxr-xr-x  1 par1305 par1305   1093536 Oct 31 10:48 mandel-tar
-rwxr-xr-x  1 par1305 par1305      8733 Oct 31 10:48 mandel-tar.c
-rw-r--r--  1 par1305 par1305   2560000 Oct 31 11:19 parallel.out
-rwxr-xr-x  1 par1305 par1305       250 Sep 26  2018 run-tareador.sh
-rw-r--r--  1 par1305 par1305   2560000 Oct 31 11:18 serial.out
-rwxr-xr-x  1 par1305 par1305       642 Mar 20  2019 strong-omp.jgr
-rwxr-xr-x  1 par1305 par1305       785 Sep 26  2018 submit-omp-i.sh
-rwxr-xr-x  1 par1305 par1305       615 Sep 26  2018 submit-omp.sh
-rwxr-xr-x  1 par1305 par1305      2666 Sep 26  2018 submit-strong-omp.sh
-rw-r--r--  1 par1305 par1305 332301731 Oct 31 10:49 tareador_llvm.log
drwxr-xr-x  3 par1305 par1305      4096 Oct 25 11:39 .tareador_precomputed_mandeld-tar
drwxr-xr-x  3 par1305 par1305      4096 Oct 25 11:39 .tareador_precomputed_mandel-tar
-rw-r--r--  1 par1305 par1305      1124 Oct 31 10:49 verbose.log
par1305@boada-1:~/lab3$ diff parallel.out serial.out
par1305@boada-1:~/lab3$
```

Figure 6: Differences between sequential and parallel versions.

After knowing that all are generated correctly, we will explore the strong scalability for the parallel code.

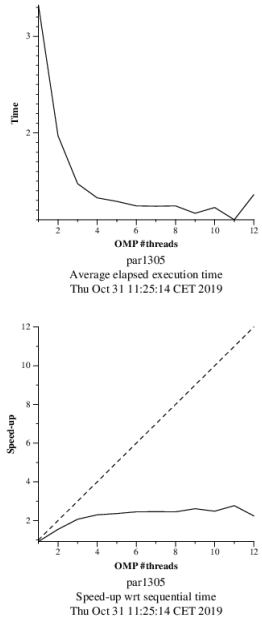


Figure 7: Speed-up plot for parallel version.

We can deduce that the scalability is far from being appropiate because the line is not close to the discontinue line, which represent the perfect scalability.

The last step of the analysis is check the trace generated with Paraver. With this, we can know more informarion like the number of tasks created or executed.

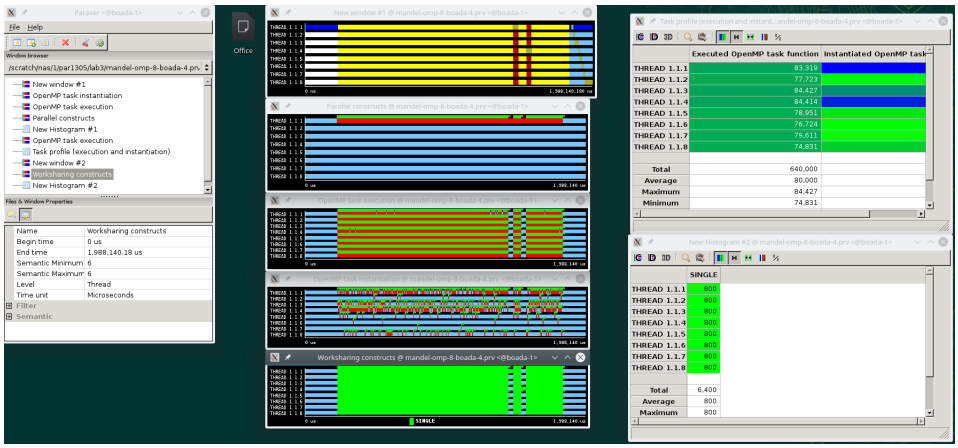


Figure 8: Some information extracted from trace by paraver.

Thanks to this tool, we know how many tasks are created and executed by every threads (top right of the image), how many times the paralellel and single construct are invoked(bottom right of the image. We know that only appears for single contruct, but how is the same, we decide to put the image only for one of them.). For some of the information, you need to add cfg files which provide it.

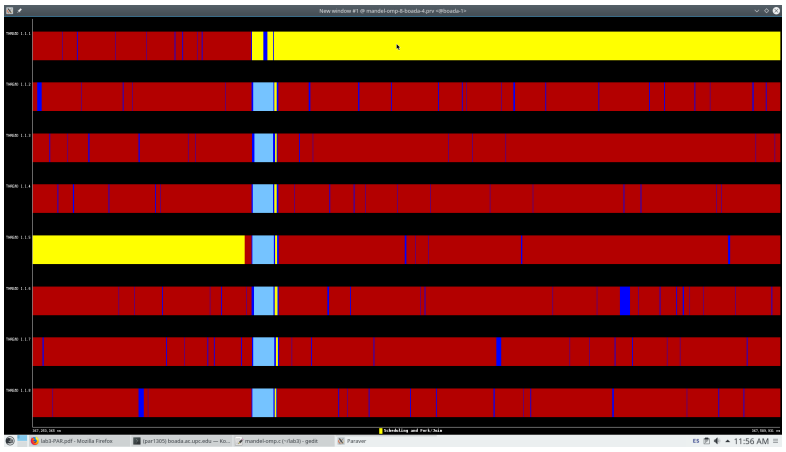


Figure 9: Trace zoom for the parallel version with 8 threads.

In this trace we can see that the amount of time spent painting points (blue) are very low in comparison to the time waiting to access to the critical region (red).

Now we will change the task descomposition strategy to the one shown in the Figure2.2 of the statement. After doing this we will make a complete analiys of the new version. The changes produced in the code are show in the Figure 9.

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

(a) Simplest tasking code for Point granularity

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

(b) Second tasking code for Point granularity

Figure 10: Comparison between two code versions.

As you can see, now the parallel and single region includes both for, reducing the creation overhead for this regions. Once the code was modified, we did the scalabilty and tracing evaluations.

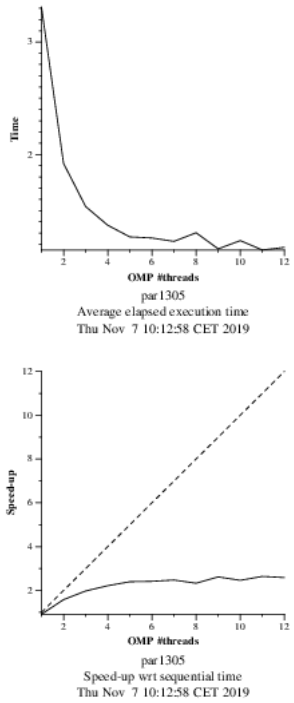


Figure 11: Speed-up plot for the second version of parallel code.

In this case the scalability is similar to which we saw in the Figure 7. This is because the granularity is the same.

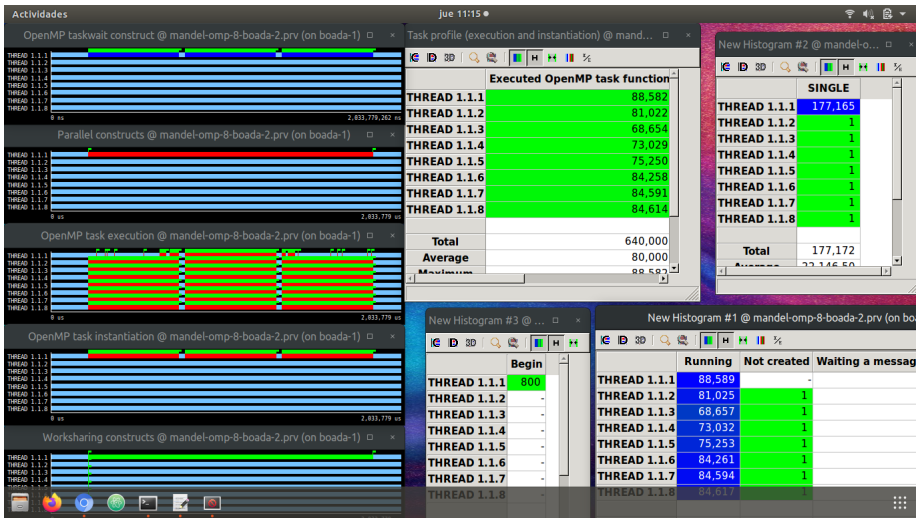


Figure 12: Information extracted from trace of second version of parallel code by paraver.

With this figure we can know that parallel and single construct are invoked once, taskwait construct is executed 800 times. Also we can know the number of tasks created/executed and which thread did it (Center top window in the image). The granularity is the same.

In addition, we zoomed the trace to extract more information from it.

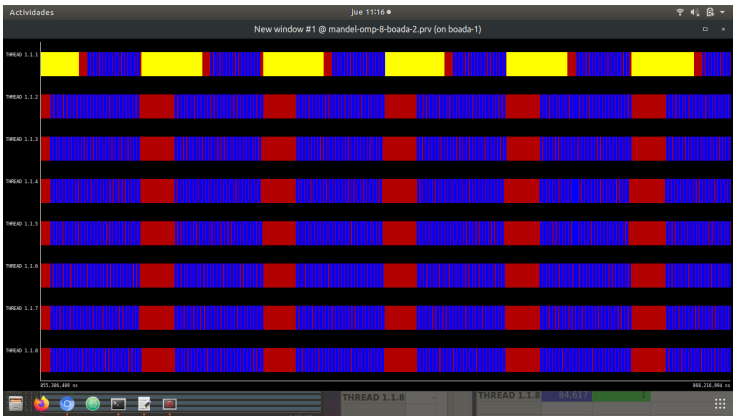


Figure 13: Trace zoom for the second parallel version with 8 threads.

Now, the threads wait less time between executions.

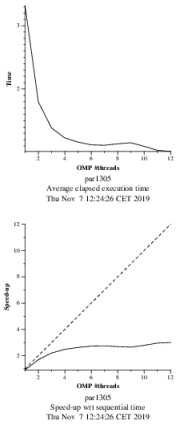
The last part of Point granularity is eveluate the code changing the construct `taskwait`for the `taskgroup`which have similiar behavior. For this, we used the code provided in the assignment Figure 2.3.

```
1  #pragma omp parallel
2  #pragma omp single
3  for (row = 0; row < height; ++row) {
4      #pragma omp taskgroup
5      {
6          for (col = 0; col < width; ++col) {
7              #pragma omp task firstprivate(row, col)
8              {
9              ...
10             }
11         }
12     } // waiting point for all descendant tasks in ↩
           taskgroup region
13 }
```

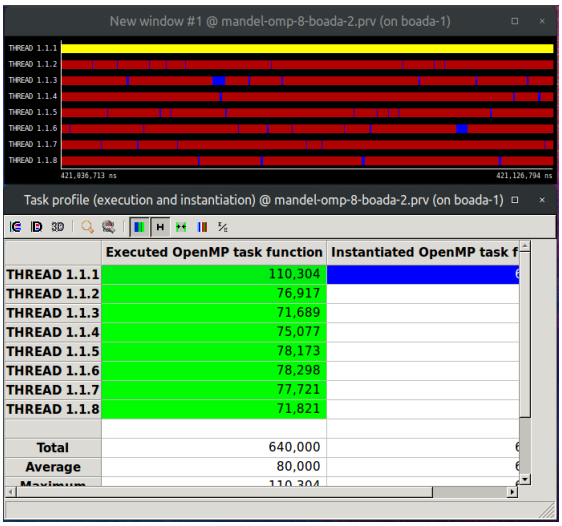
Figure 14: Code for the 3rd part of Point granularity.

In this case, we have no differences between use taskwait and taskgroup, we have the same behavior.

We think that is not necessary to use the taskwait construct in this code because the are no dependences between tasks. Now we remove the barrier and we will do the same evaluations as we did with the previous codes.



(a) Strong scalability



(b) Trace

Figure 15: Analysis of code without barrier.

We see that the number of tasks created/executed is the same. We supose that the behavior of create tasks, execute them and create tasks again is because taskpool reaches his limit of capacity.

3.1.2 Granularity control with taskloop

Now we will control the granularity of the tasks created with the "taskloop" construct. We modified the code in order to use taskloop construct, which is provided by the assignment Figure 2.4.

```
1 #pragma omp parallel
2 #pragma omp single
3 for (row = 0; row < height; ++row) {
4     #pragma omp taskloop firstprivate(row) num_tasks↩
          (64) // grainsize(width/64)
5     for (col = 0; col < width; ++col) {
6         ...
7     }
8 }
```

Figure 16: Code for the 3rd part of Point granularity.

After that, we checked it and generate the image correctly. Also we executed

the non-graphical version in order to generate the parallel binary.

We changed the granularity for the requested by the assignment to check why taskloop version performs better than task version. For know it, we generated a Paraver trace and we got this.

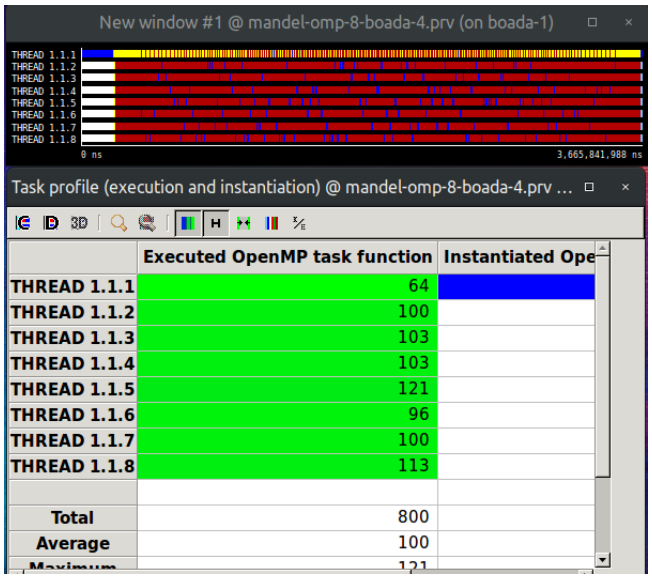
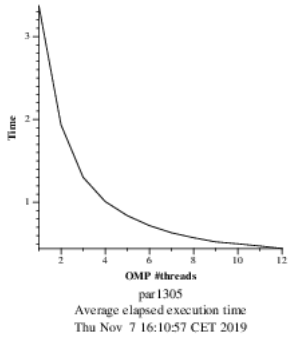


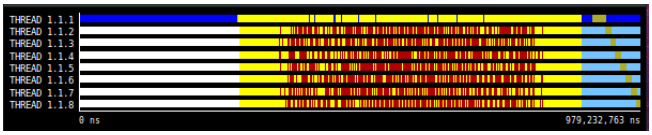
Figure 17: Paraver analysis for taskloop version.

Thanks to this trace, we know that instead of generate 640000 tasks we only generate 800 with bigger granularity, this means that the overhead caused by the huge amount of tasks with small granularity is palliated and performs better.

Now we add the clause `nogroup`to avoid the implicit barrier at the end of the construct and we did the evaluation.



(a) Strong scalability



(b) Trace

Figure 18: Analysis of code with nogroup clause.

We know that performs better for 2 reasons: less execution time and more blue zones in the trace. We also added the strong scalabilty plot and now is nearly to a good scalability

To explore the behavior with differents task granularities, nogroup clause and 8 threads, we did the following table

Number of tasks	Execution time
800	0.9 s
400	0.8 s
200	1 s
100	0.79 s
25	0.66
5	0.88
1	0.68 s

Table 1: Performance for taskloop version with nogroup clause and 8 for different number of tasks

With this results we can deduce that the number of tasks do not influence in the execution time. The clause nogroup is critical in the execution time and with it, improve a lot.

3.2 Row decomposition in OpenMP

This section contemplates a new parallelization strategy. We will create less tasks but with a heavier workload. To do this we will create tasks at a row level of the two for loops that calculate each point of the matrix.
To begin with, we execute the mandel script in a serial way, in a parallel way with one thread and with eight threads. The time of execution of every one is shown in the figure 19. As we can see the time execution for the parallel version with one thread is a little higher that the time of serial execution due to the overhead introduced by the `#pragma`clauses. But at the moment that we use 8 threads to execute the program the time is reduced exponentially.

```
par1305@boada-1:~/lab3.2$ ./mandel -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 3.041186s
par1305@boada-1:~/lab3.2$ OMP_NUM_THREADS=1 ./mandel-omp -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 3.263100s
par1305@boada-1:~/lab3.2$ OMP_NUM_THREADS=8 ./mandel-omp -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 0.451409s
```

Figure 19: Time of execution using different strategies.

As we have done in the previous section we become sure that the execution

of the three programs has been correct using the command diff as shown in the figure number 20.

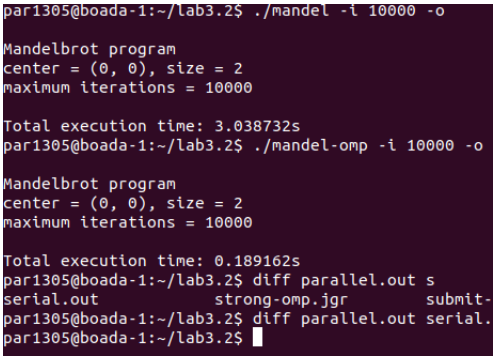


Figure 20: Check correction of row decomposition strategy execution.

Now it arrives the time to analyse the strong scalability of the program. To do it we submit the `submit-strong-omp.sh`script to boada that returns the file `mandel-omp-10000-strong-omp.ps`.In the figure number 21 we can see that the strong scalability is very good using this task decomposition strategy. This is because if we augment the number of resources of the machine the speedup grows in a proportional way. This didn't happen with the point strategy decomposition. This was because the overhead created by the task creation was very elevate. So now we can think that creating tasks to calculate entire rows of the matrix is more efficient that creating tasks to calculate unique points in the matrix.

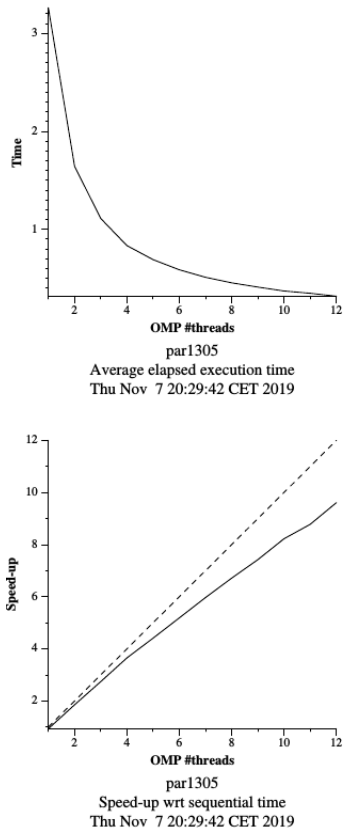


Figure 21: Strong scalability plot for row decomposition strategy.

To make a more exhaustive analysis of the program execution we need to use paraver as we have done in the previous sections. First of all we will analyse the timeline. It is represented in figures 22 and 23. Is incredible how much the time of synchronisation is deleted. It occupies a very low percentage of the total time. Another characteristic of the program is the initialisation time (serial part of the code). It takes an amount of time that represents nearly the half of the total.

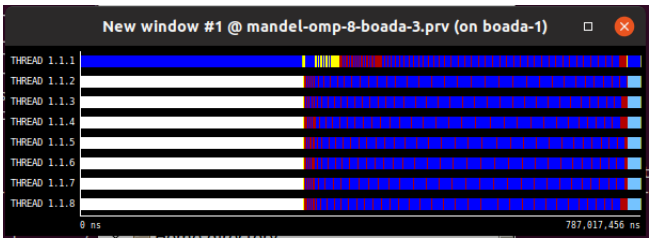


Figure 22: Timeline representing execution in 8 threads of the program.

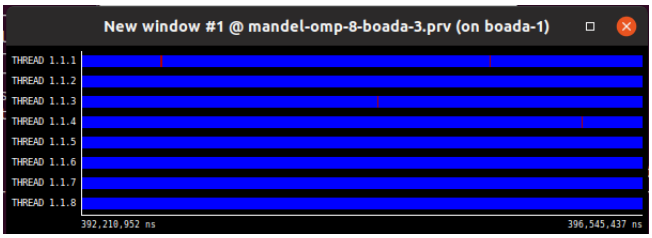


Figure 23: Zoom of timeline of the previous figure.

During the execution of the program 800 tasks are created. As we can see in the figure 24 all of them are distributed between all threads. This information is extracted from the file `OMP_task_profile.cfg`.Another important information that we can obtain from the table is that all this tasks are created in the thread one. This could be seen in figure 22 too. An average of 100 tasks are executed in each thread. During the execution of the execution only one parallel construct is executed by the thread 1. This characteristic is registered in the `OMP_parallel_contructs.cfg`.The figure 25 represents it.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	312	800
THREAD 1.1.2	72	-
THREAD 1.1.3	69	-
THREAD 1.1.4	70	-
THREAD 1.1.5	70	-
THREAD 1.1.6	70	-
THREAD 1.1.7	65	-
THREAD 1.1.8	72	-
Total	800	800
Average	100	800
Maximum	312	800
Minimum	65	800
StDev	80.15	0
Avg/Max	0.32	1

Figure 24: OMP-task-profile.cfg

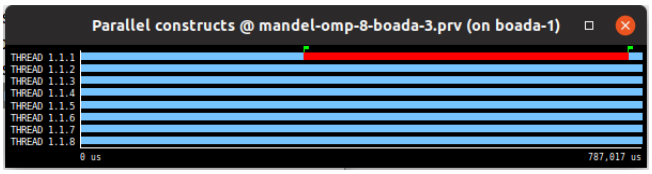


Figure 25: OMP-parallel-contructs.cfg

Due to overhead created when we abuse in the number of tasks created (one for every point in the matrix) the point decomposition do not have a good strong-scalability. However if we introduce coarser-grain decomposition strategy in which every row of the matrix is calculated by only one thread the overhead is reduced greatly and the strong-scalability is strengthened.

3.3 Optional: for–based parallelization

In this section we will apply different schedule strategies to divide the iteration between threads. Doing this we will try to reduce the load balancing problem which is caused because tasks are defined by rows and rows have different weight of work.

```
1 #pragma omp parallel for
2 for (int row = 0; row < height; ++row) {
3     for (int col = 0; col < width; ++col) {
4         ...
5     }
6 }
```

Figure 26: Simple work sharing construct

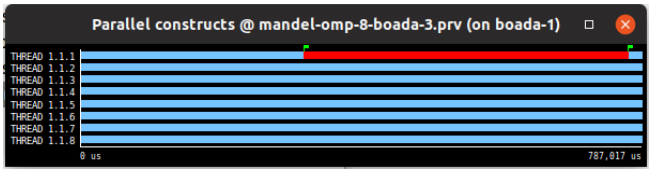


Figure 27: Strong scalability plot for sharing construct

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

Figure 28: Static divided iterations between tasks.

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

Figure 29: Dynamic divided iterations between tasks.

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

Figure 30: Guided divided iterations between tasks.

Executing the different code versions we have checked that using a guided scheduling strategy with an appropriate chunk we reduce the execution time. This is because distribution of iterations between tasks is done considering the amount of work that each iteration is going to do.

4 Conclusions

Finally we reached the conclusion that with row granularity we have better performance and scalability than point granularity. This happens because with

so fine granularity we introduce a huge overhead in comparison to the load of each tasks. Another reason is the we have limited processor and we can not perform the perfect parallelism with many tasks. The last reason for that performance is the work balancing, we have a huge different of load between tasks and this implies a worst performances than if we did with tasks with the same granularity