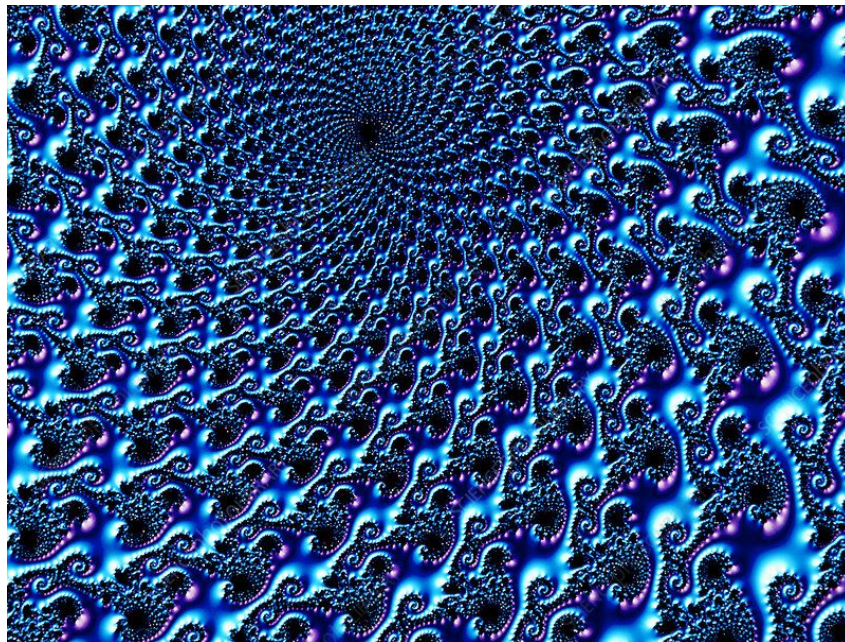


PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set



Edgar Perez Blanco
Bartomeu Perelló Comas
Group: **PAR2301**

PAR - GEI FIB
Spring 2019-2020

Contents

Introduction	4
Finally there's an optional section which makes use of a for based parallelization.	4
1. Task decomposition analysis for the Mandelbrot set computation	5
Non Graphical Version	5
Graphical Version	6
2. Point decomposition in OpenMP	7
2.1. Point strategy implementation using OpenMP task	8
Point Strategy - V1	8
Point Strategy - V2	11
Point Strategy - V3	15
2.2. Granularity control with taskloop	17
3. Row decomposition	21
3.1. Taskloop	21
3.2. for-based Parallelization	24
CONCLUSION	28

Introduction

In this report we will be experimenting with the mandelbrot set different task decomposition strategies.

First of all, in order to analyze the potential task granularity decompositions, we will be using taredor.

We have tried two different decomposition strategies: the point and the row one. The first one focuses into the most inner loop, taking each iteration as a task meanwhile the second one creates a task for each iteration of the outer loop. Furthermore, there is an optional section which does not create tasks, a for based parallelization.

1. Task decomposition analysis for the Mandelbrot set computation

Because the Mandelbrot set computation belongs to the embarrassingly parallel problem set, there are an almost infinite number of parallelization strategies. In this report we are going to focus into two different strategies: creating a task for each point and for each row.

Non Graphical Version

When trying to parallelize the non graphical version, we obtained the following task distribution. In order to create one task for each point, we added an start task directive inside the deepest for-loop. The following graphs



Point Parallelization



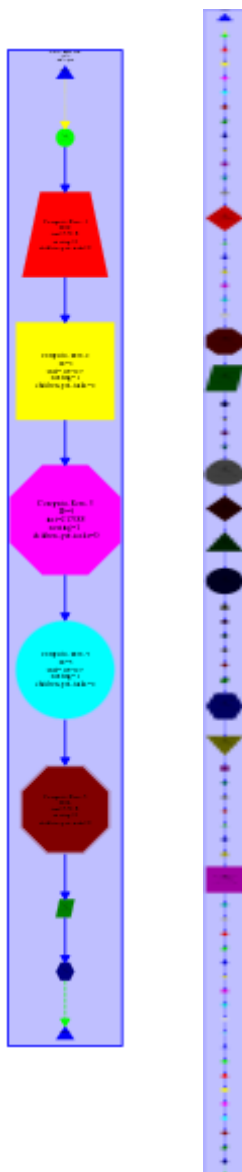
Row Parallelization

As we can see on the graphs, with a finer granularity, we obtain a big number of small tasks and with a coarser one, we obtain a few big tasks. The row parallelization appear to be better balanced. Unless we use bi quantity of threads, due to the overheads, the row version would be better.

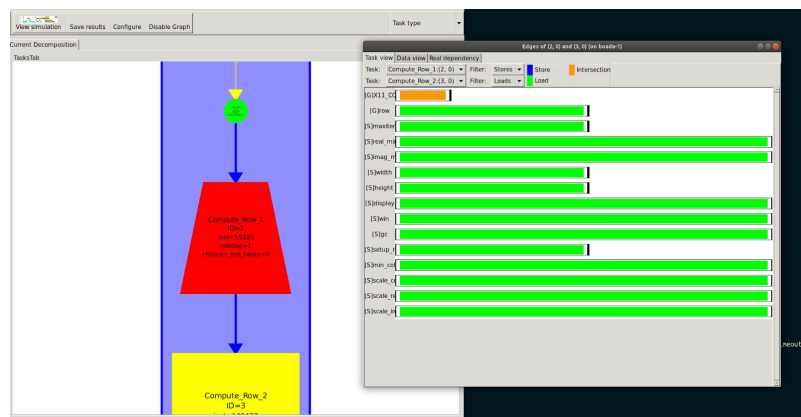
Graphical Version

On the process of parallelizing the version with the graphical interface active we get a very different graph. Despite the task distribution is the same as before, due to the dependencies generated while printing on display, we get a serial execution.

The first graph corresponds to the row parallelization which is not as finer as the second one which is easily provable by the number of tasks involved in the execution.



The paraver simulation has found data dependencies between related with X11 function calls with some global variables modified during the task.



In order to protect the printing on the screen code zone, we can add a critical clause surrounding it.

2. Point decomposition in OpenMP

As shown in the previous part of this report, the graphical code version did not work as expected due to some data dependencies during the result displaying. To solve this problem and therefore achieve the expected parallelism, regardless if we run a graphical code version or a non-graphical one, we have protected the `_DISPLAY_` code region with a `#pragma omp critical` as shown in the picture below.

```
#if _DISPLAY_
#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);
    }
}
#else
    output[row][col]=k;
#endif
```

This strategy avoids data races and unexpected changes in variables, so the displaying now works correctly.

2.1. Point strategy implementation using OpenMP task

At this point we have been asked to parallelize our code creating a task for each iteration of the deepest for-loop. As each point computation needs to work with its fixed row/col values, we must first-privatize the row variable. The col value it is considered local on each iteration, so it is not necessary to privatize it.

Point Strategy - V1

The previously described strategy is the simplest way to create a task for each point obtaining a correct output. We called this version: V1.

```
{
    /* Calculate points and save/display */
    for (int row = 0; row < height; ++row) {
        #pragma omp parallel
        #pragma omp single
        for (int col = 0; col < width; ++col)
        {
            #pragma omp task firstprivate(col)
            {
                computation code [...]
            }
        }
    }
}
```

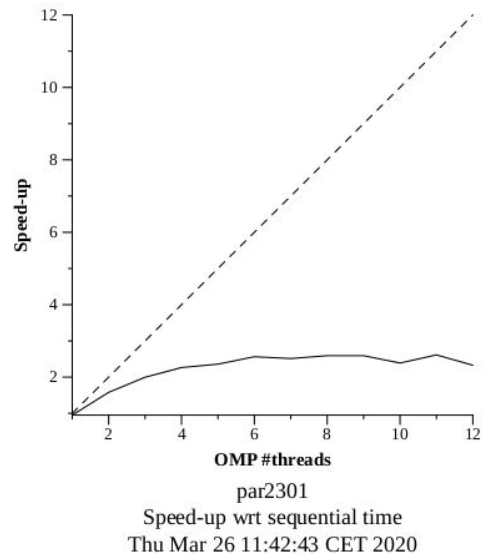
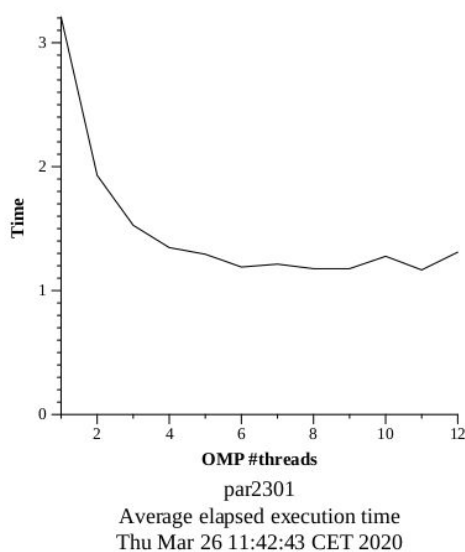
After executing V1 with 1 and 8 threads we got exactly the same output as the serial version one:

```
par2301@boada-1:~/lab3$
par2301@boada-1:~/lab3$ cmp serial.out v1_1.out
par2301@boada-1:~/lab3$ cmp serial.out v1_8.out
par2301@boada-1:~/lab3$
```

The following table show the execution time (in seconds) for each execution:

SERIAL	V1 (1 Thread)	V1 (8 Thread)	SpeedUp
3.05	3.29	1.19	2.56

On the one hand, as the table reveals, running V1 single threaded gave us a worst time than the serial execution due to the task creation overheads. On the other hand, the eight threaded V1 execution reported to be about x2.5 times faster than the serial one. The graphs below show the execution time and speed-up evolution versus the number of used threads (strong scaling analysis).



Once V1 is running over six threads, the execution stabilizes around 1.2 seconds. Therefore the speed-up also does, reporting a peak of x2.6. The use of a bigger number of threads will not report any performance improvement.

TASK CREATION ANALYSIS

When the program has finished running it has created a total of 640,000 tasks, obviously when executed over one thread everything is done by itself, but when it's executed over 8 threads we see some differences.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	640,000	640,000
Total	640,000	640,000
Average	640,000	640,000
Maximum	640,000	640,000
Minimum	640,000	640,000
StDev	0	0
Avg/Max	1	1

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	80,181	220,000
THREAD 1.1.2	78,408	29,600
THREAD 1.1.3	83,451	96,000
THREAD 1.1.4	82,514	192,800
THREAD 1.1.5	79,901	43,200
THREAD 1.1.6	77,817	8,000
THREAD 1.1.7	84,505	2,400
THREAD 1.1.8	73,223	48,000
Total	640,000	640,000
Average	80,000	80,000
Maximum	84,505	220,000
Minimum	73,223	2,400
StDev	3,387.38	78,025.64
Avg/Max	0.95	0.36

Firstly we can appreciate that in our case the first thread and the fourth one have instantiated more than half of the total tasks, but when we take a look at how many tasks every thread has executed we see a very good balance on task distribution with an standard deviation of 3,387 which is almost negligible as the average is 80,000.

The parallel construct is invoked 800 times by the first thread, and the single worksharing one is invoked a total of 6,400 time and 800 times for each thread. We have obtained these numbers by using configurations files on Paraver and changing some options on the task profile configuration.

	REGION (open)		SINGLE
THREAD 1.1.1	800	THREAD 1.1.1	800
THREAD 1.1.2	-	THREAD 1.1.2	800
THREAD 1.1.3	-	THREAD 1.1.3	800
THREAD 1.1.4	-	THREAD 1.1.4	800
THREAD 1.1.5	-	THREAD 1.1.5	800
THREAD 1.1.6	-	THREAD 1.1.6	800
THREAD 1.1.7	-	THREAD 1.1.7	800
THREAD 1.1.8	-	THREAD 1.1.8	800
Total	800	Total	6,400
Average	800	Average	800
Maximum	800	Maximum	800
Minimum	800	Minimum	800
StDev	0	StDev	0
Avg/Max	1	Avg/Max	1

In spite of this, we can manually calculate how many times are executed since we know the constructs are executed inside de first for-loop and the number of iterations (#rows = 800).

Point Strategy - V2

The next version, which we call V2, corresponds to a modification of V1. We wrote the parallel and single worksharing outside of both loops, in order to instantiate them once. Moreover, we added a taskwait clause after the second for-loop. With this strategy, each thread will wait until all tasks for each one of the rows finish; after that, the thread will advance one iteration of the row loop and generate a new bunch of tasks.

```
{  
    /* Calculate points and save/display */  
    #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)  
            {  
                computation code [...]  
            }  
        }  
        #pragma omp taskwait // wait for all child tasks (a full row)  
    }  
}
```

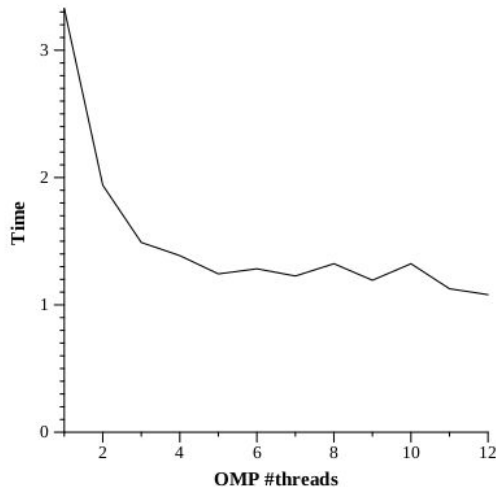
After executing V2 with 1 and 8 threads we got exactly the same output as the serial version one:

```
par2301@boada-1:~/lab3/strong_v2$  
par2301@boada-1:~/lab3/strong_v2$ cmp ../serial.out v2_1.out  
par2301@boada-1:~/lab3/strong_v2$ cmp ../serial.out v2_8.out  
par2301@boada-1:~/lab3/strong_v2$
```

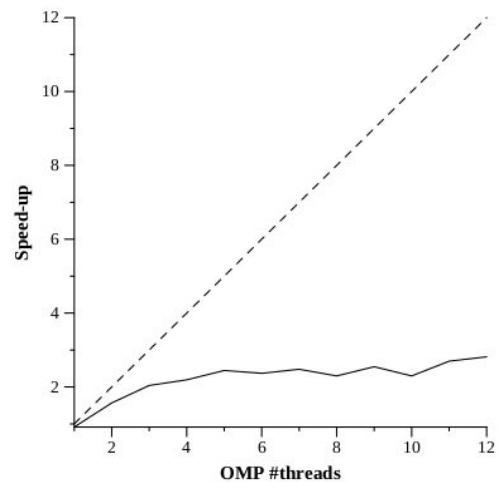
The following table show the execution time (in seconds) for each execution:

SERIAL	V1 (1 Thread)	V1 (8 Thread)	SpeedUp
3.05	3.31	1.36	2.24

The strong scalability plots are very similar to the V1 ones. Once the program uses 5/6 threads, the scaling stops due to task creation and synchronization overheads. We think the taskwait overhead does not allow the threads to explode the mandelbrot's embarrassingly parallelism, as they have to stop and sync at each row.



par2301
Average elapsed execution time
Thu Mar 26 15:44:19 CET 2020



par2301
Speed-up wrt sequential time
Thu Mar 26 15:44:19 CET 2020

TASK DECOMPOSITION ANALYSIS

Now we have pretty similar results as the execution done before but there's a slightly increase on the standard deviation of 2000, which does not mean that the balance between task execution is bad, it's still very good.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,688	640,000
THREAD 1.1.2	72,824	-
THREAD 1.1.3	81,424	-
THREAD 1.1.4	69,964	-
THREAD 1.1.5	83,644	-
THREAD 1.1.6	82,839	-
THREAD 1.1.7	80,498	-
THREAD 1.1.8	80,119	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,688	640,000
Minimum	69,964	640,000
StDev	5,604.98	0
Avg/Max	0.90	1

The granularity has not changed because the total number of tasks instantiation is the same.

As the parallel construct is located before the for construct it is only invoked once. The single worksharing construct it is only invoked 8 times because when the execution reaches the parallel region, a team of 8 threads is created and they all invoke the single construct.

	REGION (open)		SINGLE
THREAD 1.1.1	1	THREAD 1.1.1	1
THREAD 1.1.2	-	THREAD 1.1.2	1
THREAD 1.1.3	-	THREAD 1.1.3	1
THREAD 1.1.4	-	THREAD 1.1.4	1
THREAD 1.1.5	-	THREAD 1.1.5	1
THREAD 1.1.6	-	THREAD 1.1.6	1
THREAD 1.1.7	-	THREAD 1.1.7	1
THREAD 1.1.8	-	THREAD 1.1.8	1
Total	1	Total	8
Average	1	Average	1
Maximum	1	Maximum	1
Minimum	1	Minimum	1
StDev	0	StDev	0
Avg/Max	1	Avg/Max	1

In this version we added the taskwait construct. Because of its localization, inside the first loop but outside the second one; and the single construct, it is executed 800 times (one for each row), by one thread.

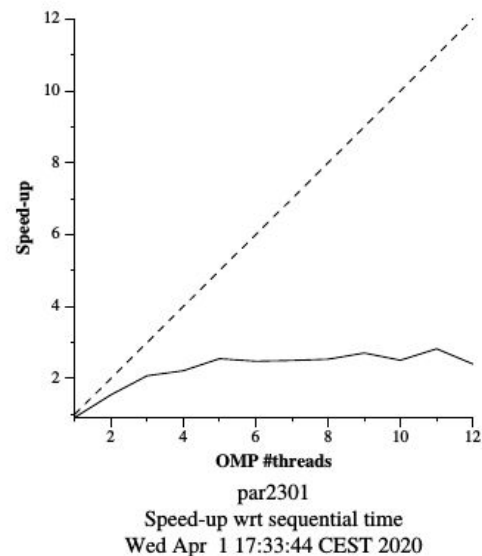
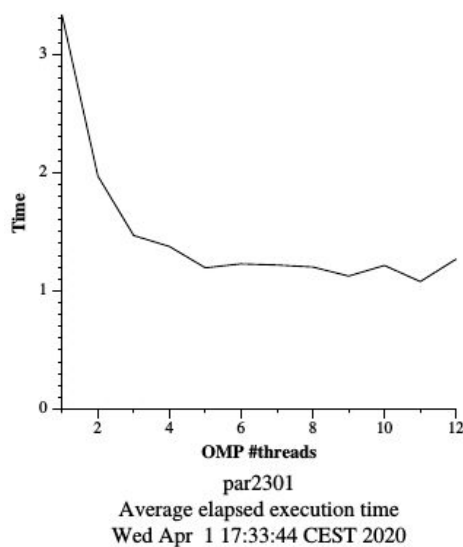
	Begin
THREAD 1.1.1	800
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	800
Average	800
Maximum	800
Minimum	800
StDev	0
Avg/Max	1

TASKWAIT VS TASKGROUP

When a thread reaches a taskwait construct, it is forced to wait for the tasks created at the current execution level, which means that if we had recursively created tasks inside the current one, these would not be waited by the thread. Otherwise, if use the taskgroup construct, all the tasks created inside the delimited zone, regardless of who is the parent task, will be waited.

In this case, as our program is embarrassingly parallel and it does not create tasks recursively, the taskgroup will work as a taskwait.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,832	640,000
THREAD 1.1.2	79,977	-
THREAD 1.1.3	73,655	-
THREAD 1.1.4	79,199	-
THREAD 1.1.5	82,329	-
THREAD 1.1.6	82,564	-
THREAD 1.1.7	71,873	-
THREAD 1.1.8	80,571	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,832	640,000
Minimum	71,873	640,000
StDev	5,195.45	0
Avg/Max	0.89	1



As we can see in the graphs above, there is no difference between the taskgroup version and the V2.

Point Strategy - V3

At this point, we have been asked to remove the `taskwait` clause to avoid waiting for each row to end and let the threads continue with the execution.

```
{  
    /* Calculate points and save/display */  
    #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)  
            {  
                computation code [...]  
            }  
        }  
    }  
}
```

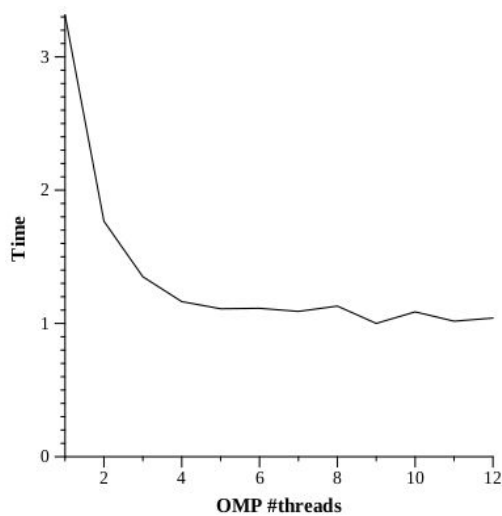
After executing V3 with 1 and 8 threads we got exactly the same output as the serial version one:

```
par2301@boada-1:~/lab3/v3_mandel/src$  
par2301@boada-1:~/lab3/v3_mandel/src$ cmp ../../serial.out v3_1.out  
par2301@boada-1:~/lab3/v3_mandel/src$ cmp ../../serial.out v3_8.out  
par2301@boada-1:~/lab3/v3_mandel/src$
```

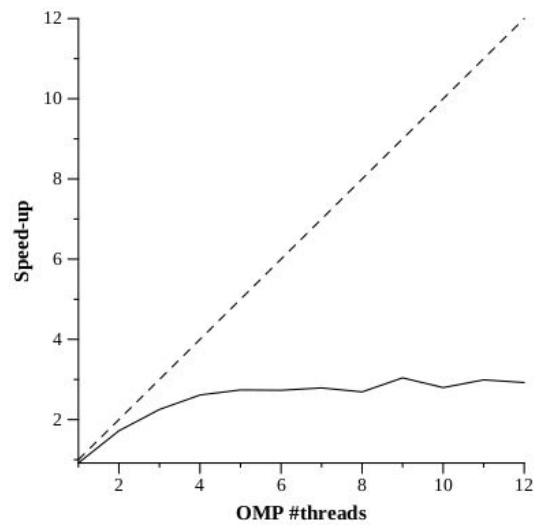
The following table show the execution time (in seconds) for each execution:

SERIAL	V1 (1 Thread)	V1 (8 Thread)	SpeedUp
3.05	3.37	1.17	2.61

Again, talking about strong scalability, we get similar results to the previous code versions. The execution time decreases until the use of 6 threads, where the time stabilizes. On the graphs below we can see that the greatest speed-up was obtained while using 9 threads, but due to task creation overheads, we would not improve this result.



par2301
Average elapsed execution time
Thu Mar 26 15:43:28 CET 2020



par2301
Speed-up wrt sequential time
Thu Mar 26 15:43:28 CET 2020

For this version, as we just removed a barrier, the number of generated and executed tasks must be the same as V2. Sometimes, the task generator thread stops the task creation and starts working into the existing tasks. As memory is limited, generating all the tasks could be sometimes not suitable, so it is better to generate tasks just a little bit faster than they get executed.

2.2. Granularity control with taskloop

- For grainsize(1), which is the equivalent to the second task version that you analyzed, why the new version based on taskloop performs better than the version based on task?

We think that the new version performs better because of the overhead that is produced when creating tasks. On the previous version the program is creating a total of 640,000 tasks meanwhile this new version is creating 12,800, of course the granularity is lower but the tradeoff is positive in this case.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,688	640,000
THREAD 1.1.2	72,824	-
THREAD 1.1.3	81,424	-
THREAD 1.1.4	69,964	-
THREAD 1.1.5	83,644	-
THREAD 1.1.6	82,839	-
THREAD 1.1.7	80,498	-
THREAD 1.1.8	80,119	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,688	640,000
Minimum	69,964	640,000
StDev	5,604.98	0
Avg/Max	0.90	1

Task information V2

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	81,326	4,383
THREAD 1.1.2	83,073	1,169
THREAD 1.1.3	79,925	1,239
THREAD 1.1.4	80,917	1,249
THREAD 1.1.5	82,288	1,066
THREAD 1.1.6	79,640	1,130
THREAD 1.1.7	83,317	1,283
THREAD 1.1.8	81,514	1,281
Total	652,000	12,800
Average	81,500	1,600
Maximum	83,317	4,383
Minimum	79,640	1,066
StDev	1,260.32	1,054.32
Avg/Max	0.98	0.37

Task information V4 Grainsize 1

- Explore how the new version behaves in terms of performance when using 8 threads and for different task granularities, i.e. setting the number of tasks or iterations per task to 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1, for example. Reason about the results that are obtained.

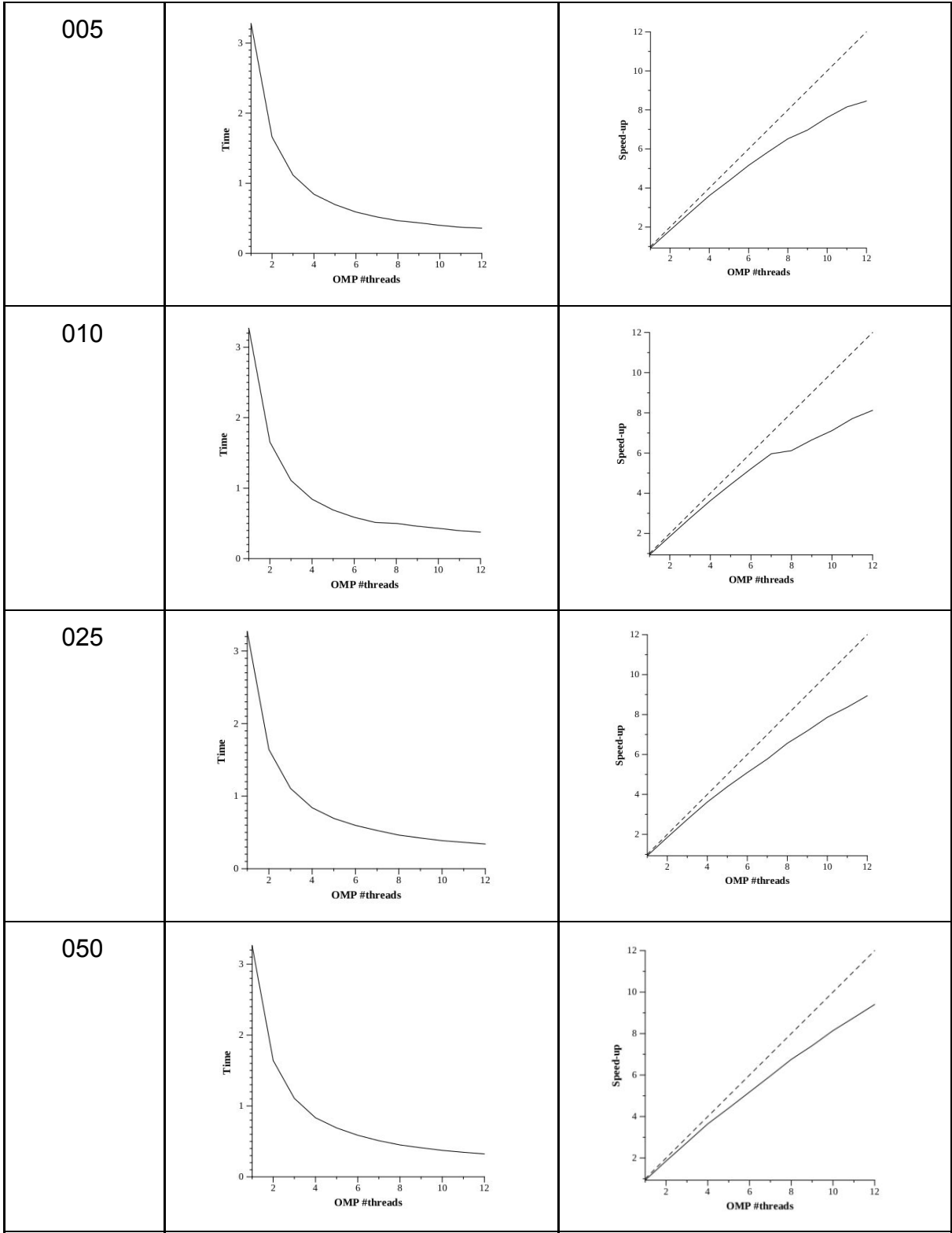
Below we have a sequence of pairs of graphs representing the execution time and the speedup of the different grain sizes.

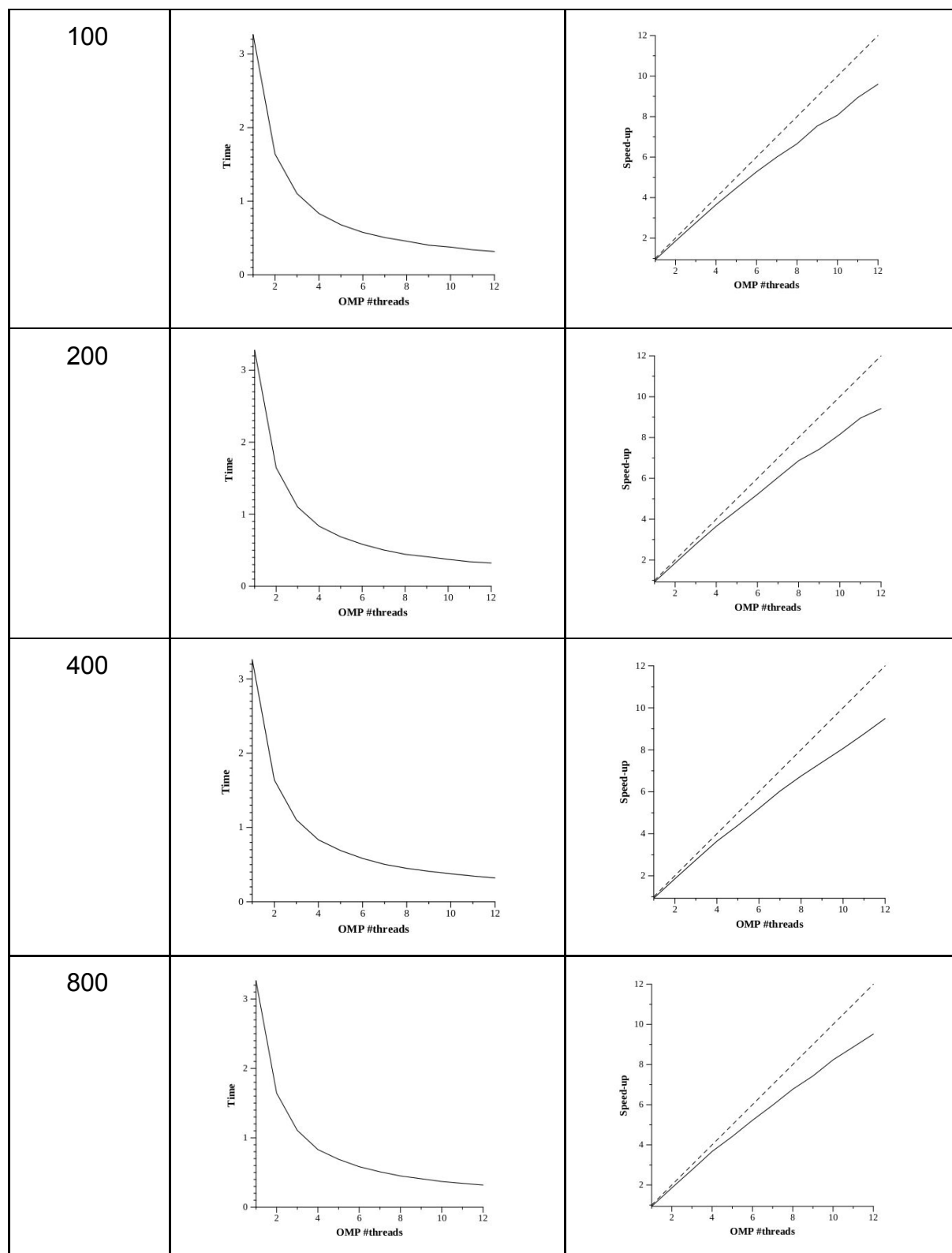
If we take a look at the execution time it's mostly the same on all the different executions, but if we look at the speedup we can appreciate some variation.

From grain size 1 to 5 the speedup rises, but when it hits the 10 grain size at 7 threads it flattens a bit, but afterwards it keeps rising but at a lower pace compared to the previous ones.

Afterwards, all speedups stay more or less the same while being better than the previous ones.

GRAINSIZE	EXE. TIME VS #THREADS	SPEEDUP VS #THREADS
001		
002		





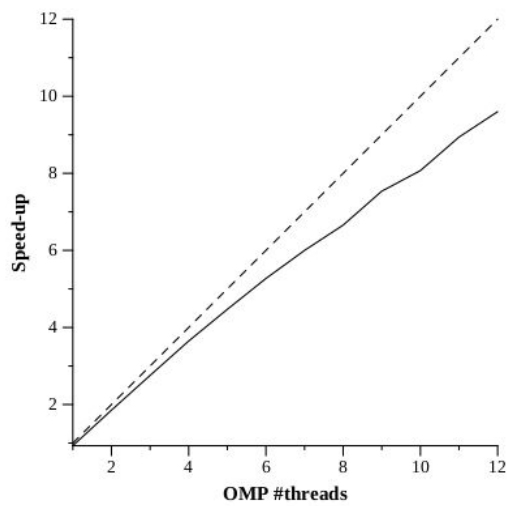
3. Row decomposition

3.1. Taskloop

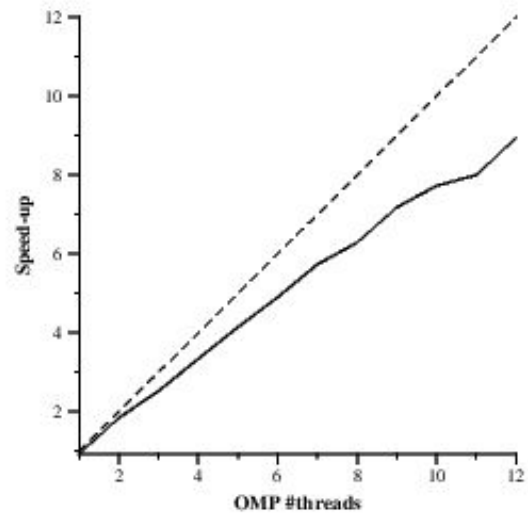
We decided to use the taskloop construct in the most outer loop in order to do the row decomposition scheme. Below we can see the differences between the actual row decomposition and the best point decomposition which was the one with the grain size 100.

#THREADS	Grain Size 100	Row decomposition
1	3.26	3.26
2	1.64	1.65
3	1.1	1.2
4	0.83	0.91
5	0.68	0.73
6	0.58	0.62
7	0.51	0.53
8	0.46	0.48
9	0.4	0.42
10	0.38	0.39
11	0.34	0.38
12	0.32	0.34

If we have a look at the different speedups we will see similar results on the differences between both, being better the grain size 100.



Grain Size 100



Taskloop

With Paraver we obtained the following charts related to the execution:

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	31	1
THREAD 1.1.2	8	-
THREAD 1.1.3	7	-
THREAD 1.1.4	8	-
THREAD 1.1.5	8	-
THREAD 1.1.6	8	-
THREAD 1.1.7	5	-
THREAD 1.1.8	5	-
Total	80	1
Average	10	1
Maximum	31	1
Minimum	5	1
StDev	8.03	0
Avg/Max	0.32	1

This capture show the tasks executed for each thread and how much have been instantiated. In this case the scheduler has decided the distribution of the tasks between threads.

The following charts show how much times the Parallel construct and Single constructs have been invoked:

	REGION (open)
THREAD 1.1.1	1
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	1
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Parallel construct

	SINGLE
THREAD 1.1.1	1
THREAD 1.1.2	1
THREAD 1.1.3	1
THREAD 1.1.4	1
THREAD 1.1.5	1
THREAD 1.1.6	1
THREAD 1.1.7	1
THREAD 1.1.8	1
Total	8
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Single construct

We see that the parallel construct is instantiated just once because the pragma is located outside of the loop. The single construct, as always, is reached by all the threads, but only executed but one of them.

The following chart shows how much time each thread have been running and waiting for synchronization:

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	88.48 %	-	11.27 %	0.25 %	0.00 %	0.00 %
THREAD 1.1.2	69.82 %	30.16 %	0.03 %	0.00 %	0.00 %	-
THREAD 1.1.3	66.88 %	30.16 %	2.97 %	0.00 %	0.00 %	-
THREAD 1.1.4	66.79 %	30.14 %	3.07 %	0.00 %	0.00 %	-
THREAD 1.1.5	62.61 %	30.16 %	7.23 %	0.00 %	0.00 %	-
THREAD 1.1.6	68.55 %	30.17 %	1.28 %	0.00 %	0.00 %	-
THREAD 1.1.7	66.91 %	30.21 %	2.88 %	0.00 %	0.00 %	-
THREAD 1.1.8	62.13 %	30.20 %	7.67 %	0.00 %	0.00 %	-
Total	552.15 %	211.19 %	36.40 %	0.25 %	0.01 %	0.00 %
Average	69.02 %	30.17 %	4.55 %	0.03 %	0.00 %	0.00 %
Maximum	88.48 %	30.21 %	11.27 %	0.25 %	0.00 %	0.00 %
Minimum	62.13 %	30.14 %	0.03 %	0.00 %	0.00 %	0.00 %
StDev	7.76 %	0.02 %	3.55 %	0.08 %	0.00 %	0 %
Avg/Max	0.78	1.00	0.40	0.13	0.40	1

We can see that the first thread is the one who has been creating the tasks, so its running time is near to 90%. The rest of the threads have been running most of the time once they have been created. The synchronization percentage times are little, so we can conclude that the execution has been successfully parallelized with a good load balance and an acceptable overhead.

3.2. for-based Parallelization

In order to do a for-based parallelization we just have to include the according clause: `#pragma omp for` inside of parallel zone (delimited with a `#pragma omp parallel`). For each scheduling mode, we created a code version. The following captures illustrate the previous description:

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp for schedule(static)
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
```

for-based, static scheduling

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp for schedule(dynamic)
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
```

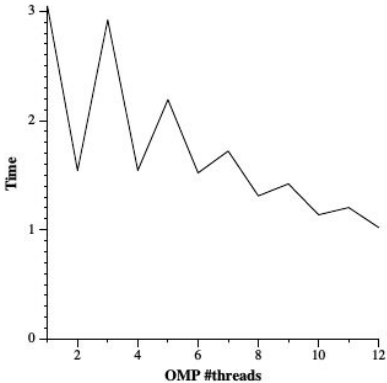
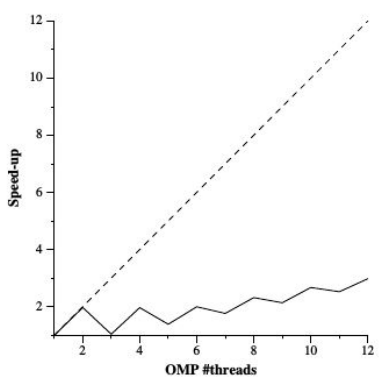
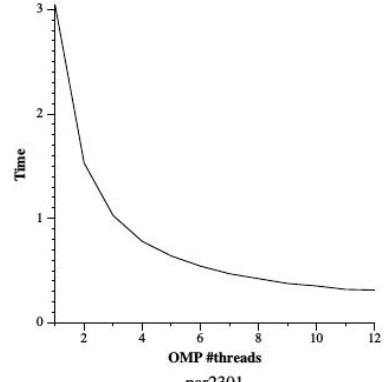
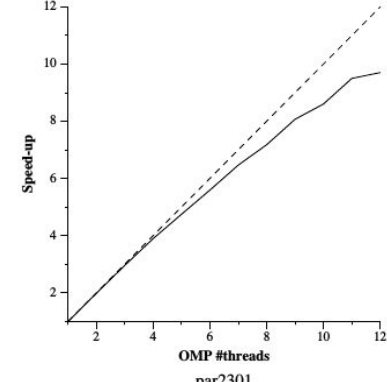
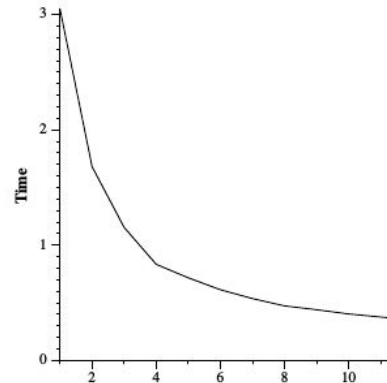
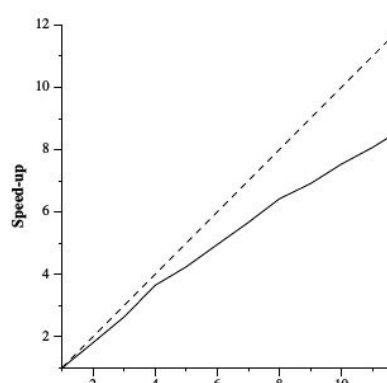
for-based, dynamic scheduling

```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp for schedule(guided)
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
```

for-based, guided scheduling

The static scheduling divides the iterations between all the threads before the execution. The dynamic and guided ones, divide the iterations dynamically, once a thread terminates its work, receives another portion of work. The difference between dynamic and guided is that dynamic scheduling always give the same number of iterations to the free thread meanwhile the guided one gives a chunk of iterations of dynamic size which decreases as the threads grab iterations.

SCHEDULING	PARAVER OUTPUT																																																																																																																							
static	<table><tr><th></th><th>Running</th><th>Not created</th><th>Synchronization</th><th>Scheduling and Fork/Join</th><th>I/O</th><th>Others</th></tr><tr><td>THREAD 1.1.1</td><td>14.58 %</td><td>-</td><td>85.28 %</td><td>0.13 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>THREAD 1.1.2</td><td>0.08 %</td><td>13.71 %</td><td>86.21 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.3</td><td>18.22 %</td><td>13.71 %</td><td>68.06 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.4</td><td>86.28 %</td><td>13.71 %</td><td>0.00 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.5</td><td>84.78 %</td><td>13.71 %</td><td>1.50 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.6</td><td>17.81 %</td><td>13.72 %</td><td>68.48 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.7</td><td>0.07 %</td><td>13.71 %</td><td>86.22 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.8</td><td>0.05 %</td><td>13.71 %</td><td>86.24 %</td><td>-</td><td>0.00 %</td><td>-</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Total</td><td>221.87 %</td><td>95.00 %</td><td>481.00 %</td><td>0.13 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Average</td><td>27.73 %</td><td>13.71 %</td><td>60.25 %</td><td>0.13 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Maximum</td><td>86.28 %</td><td>13.72 %</td><td>86.24 %</td><td>0.13 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Minimum</td><td>0.05 %</td><td>13.71 %</td><td>0.00 %</td><td>0.13 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>StDev</td><td>34.17 %</td><td>0.00 %</td><td>35.11 %</td><td>0 %</td><td>0.00 %</td><td>0 %</td></tr><tr><td>Avg/Max</td><td>0.32</td><td>1.00</td><td>0.70</td><td>1</td><td>0.60</td><td>1</td></tr></table>		Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others	THREAD 1.1.1	14.58 %	-	85.28 %	0.13 %	0.00 %	0.00 %	THREAD 1.1.2	0.08 %	13.71 %	86.21 %	-	0.00 %	-	THREAD 1.1.3	18.22 %	13.71 %	68.06 %	-	0.00 %	-	THREAD 1.1.4	86.28 %	13.71 %	0.00 %	-	0.00 %	-	THREAD 1.1.5	84.78 %	13.71 %	1.50 %	-	0.00 %	-	THREAD 1.1.6	17.81 %	13.72 %	68.48 %	-	0.00 %	-	THREAD 1.1.7	0.07 %	13.71 %	86.22 %	-	0.00 %	-	THREAD 1.1.8	0.05 %	13.71 %	86.24 %	-	0.00 %	-								Total	221.87 %	95.00 %	481.00 %	0.13 %	0.00 %	0.00 %	Average	27.73 %	13.71 %	60.25 %	0.13 %	0.00 %	0.00 %	Maximum	86.28 %	13.72 %	86.24 %	0.13 %	0.00 %	0.00 %	Minimum	0.05 %	13.71 %	0.00 %	0.13 %	0.00 %	0.00 %	StDev	34.17 %	0.00 %	35.11 %	0 %	0.00 %	0 %	Avg/Max	0.32	1.00	0.70	1	0.60	1							
		Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others																																																																																																																	
	THREAD 1.1.1	14.58 %	-	85.28 %	0.13 %	0.00 %	0.00 %																																																																																																																	
	THREAD 1.1.2	0.08 %	13.71 %	86.21 %	-	0.00 %	-																																																																																																																	
	THREAD 1.1.3	18.22 %	13.71 %	68.06 %	-	0.00 %	-																																																																																																																	
	THREAD 1.1.4	86.28 %	13.71 %	0.00 %	-	0.00 %	-																																																																																																																	
	THREAD 1.1.5	84.78 %	13.71 %	1.50 %	-	0.00 %	-																																																																																																																	
	THREAD 1.1.6	17.81 %	13.72 %	68.48 %	-	0.00 %	-																																																																																																																	
	THREAD 1.1.7	0.07 %	13.71 %	86.22 %	-	0.00 %	-																																																																																																																	
	THREAD 1.1.8	0.05 %	13.71 %	86.24 %	-	0.00 %	-																																																																																																																	
	Total	221.87 %	95.00 %	481.00 %	0.13 %	0.00 %	0.00 %																																																																																																																	
	Average	27.73 %	13.71 %	60.25 %	0.13 %	0.00 %	0.00 %																																																																																																																	
	Maximum	86.28 %	13.72 %	86.24 %	0.13 %	0.00 %	0.00 %																																																																																																																	
Minimum	0.05 %	13.71 %	0.00 %	0.13 %	0.00 %	0.00 %																																																																																																																		
StDev	34.17 %	0.00 %	35.11 %	0 %	0.00 %	0 %																																																																																																																		
Avg/Max	0.32	1.00	0.70	1	0.60	1																																																																																																																		
dynamic	<table><tr><td colspan="7"><div>IE ID 3D 🔍 🖨️ 🌈 H ⏮ ⏭ 🇺🇦 ½</div></td></tr><tr><th></th><th>Running</th><th>Not created</th><th>Synchronization</th><th>Scheduling and Fork/Join</th><th>I/O</th><th>Others</th></tr><tr><td>THREAD 1.1.1</td><td>27.78 %</td><td>-</td><td>1.94 %</td><td>70.27 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>THREAD 1.1.2</td><td>0.00 %</td><td>26.09 %</td><td>12.15 %</td><td>61.76 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.3</td><td>0.00 %</td><td>26.09 %</td><td>22.50 %</td><td>51.41 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.4</td><td>0.00 %</td><td>26.09 %</td><td>9.82 %</td><td>64.09 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.5</td><td>0.00 %</td><td>26.08 %</td><td>18.92 %</td><td>55.00 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.6</td><td>0.00 %</td><td>26.08 %</td><td>0.00 %</td><td>73.92 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.7</td><td>0.00 %</td><td>26.09 %</td><td>23.42 %</td><td>50.49 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.8</td><td>0.00 %</td><td>26.39 %</td><td>20.75 %</td><td>52.86 %</td><td>0.00 %</td><td>-</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Total</td><td>27.79 %</td><td>182.90 %</td><td>109.49 %</td><td>479.81 %</td><td>0.01 %</td><td>0.00 %</td></tr><tr><td>Average</td><td>3.47 %</td><td>26.13 %</td><td>13.69 %</td><td>59.98 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Maximum</td><td>27.78 %</td><td>26.39 %</td><td>23.42 %</td><td>73.92 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Minimum</td><td>0.00 %</td><td>26.08 %</td><td>0.00 %</td><td>50.49 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>StDev</td><td>9.19 %</td><td>0.11 %</td><td>8.60 %</td><td>8.36 %</td><td>0.00 %</td><td>0 %</td></tr><tr><td>Avg/Max</td><td>0.13</td><td>0.99</td><td>0.58</td><td>0.81</td><td>0.49</td><td>1</td></tr></table>	<div>IE ID 3D 🔍 🖨️ 🌈 H ⏮ ⏭ 🇺🇦 ½</div>								Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others	THREAD 1.1.1	27.78 %	-	1.94 %	70.27 %	0.00 %	0.00 %	THREAD 1.1.2	0.00 %	26.09 %	12.15 %	61.76 %	0.00 %	-	THREAD 1.1.3	0.00 %	26.09 %	22.50 %	51.41 %	0.00 %	-	THREAD 1.1.4	0.00 %	26.09 %	9.82 %	64.09 %	0.00 %	-	THREAD 1.1.5	0.00 %	26.08 %	18.92 %	55.00 %	0.00 %	-	THREAD 1.1.6	0.00 %	26.08 %	0.00 %	73.92 %	0.00 %	-	THREAD 1.1.7	0.00 %	26.09 %	23.42 %	50.49 %	0.00 %	-	THREAD 1.1.8	0.00 %	26.39 %	20.75 %	52.86 %	0.00 %	-								Total	27.79 %	182.90 %	109.49 %	479.81 %	0.01 %	0.00 %	Average	3.47 %	26.13 %	13.69 %	59.98 %	0.00 %	0.00 %	Maximum	27.78 %	26.39 %	23.42 %	73.92 %	0.00 %	0.00 %	Minimum	0.00 %	26.08 %	0.00 %	50.49 %	0.00 %	0.00 %	StDev	9.19 %	0.11 %	8.60 %	8.36 %	0.00 %	0 %	Avg/Max	0.13	0.99	0.58	0.81	0.49	1
	<div>IE ID 3D 🔍 🖨️ 🌈 H ⏮ ⏭ 🇺🇦 ½</div>																																																																																																																							
		Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others																																																																																																																	
	THREAD 1.1.1	27.78 %	-	1.94 %	70.27 %	0.00 %	0.00 %																																																																																																																	
	THREAD 1.1.2	0.00 %	26.09 %	12.15 %	61.76 %	0.00 %	-																																																																																																																	
	THREAD 1.1.3	0.00 %	26.09 %	22.50 %	51.41 %	0.00 %	-																																																																																																																	
	THREAD 1.1.4	0.00 %	26.09 %	9.82 %	64.09 %	0.00 %	-																																																																																																																	
	THREAD 1.1.5	0.00 %	26.08 %	18.92 %	55.00 %	0.00 %	-																																																																																																																	
	THREAD 1.1.6	0.00 %	26.08 %	0.00 %	73.92 %	0.00 %	-																																																																																																																	
	THREAD 1.1.7	0.00 %	26.09 %	23.42 %	50.49 %	0.00 %	-																																																																																																																	
	THREAD 1.1.8	0.00 %	26.39 %	20.75 %	52.86 %	0.00 %	-																																																																																																																	
	Total	27.79 %	182.90 %	109.49 %	479.81 %	0.01 %	0.00 %																																																																																																																	
	Average	3.47 %	26.13 %	13.69 %	59.98 %	0.00 %	0.00 %																																																																																																																	
Maximum	27.78 %	26.39 %	23.42 %	73.92 %	0.00 %	0.00 %																																																																																																																		
Minimum	0.00 %	26.08 %	0.00 %	50.49 %	0.00 %	0.00 %																																																																																																																		
StDev	9.19 %	0.11 %	8.60 %	8.36 %	0.00 %	0 %																																																																																																																		
Avg/Max	0.13	0.99	0.58	0.81	0.49	1																																																																																																																		
guided	<table><tr><td colspan="7"><div>IE ID 3D 🔍 🖨️ 🌈 H ⏮ ⏭ 🇺🇦 ½</div></td></tr><tr><th></th><th>Running</th><th>Not created</th><th>Synchronization</th><th>Scheduling and Fork/Join</th><th>I/O</th><th>Others</th></tr><tr><td>THREAD 1.1.1</td><td>34.50 %</td><td>-</td><td>0.00 %</td><td>65.50 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>THREAD 1.1.2</td><td>0.00 %</td><td>33.24 %</td><td>0.00 %</td><td>66.75 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.3</td><td>0.00 %</td><td>33.25 %</td><td>0.00 %</td><td>66.74 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.4</td><td>0.00 %</td><td>33.25 %</td><td>0.00 %</td><td>66.74 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.5</td><td>0.00 %</td><td>33.24 %</td><td>0.00 %</td><td>66.75 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.6</td><td>0.00 %</td><td>33.26 %</td><td>0.00 %</td><td>66.74 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.7</td><td>0.00 %</td><td>33.25 %</td><td>0.00 %</td><td>66.75 %</td><td>0.00 %</td><td>-</td></tr><tr><td>THREAD 1.1.8</td><td>0.00 %</td><td>33.25 %</td><td>0.00 %</td><td>66.74 %</td><td>0.00 %</td><td>-</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Total</td><td>34.51 %</td><td>232.74 %</td><td>0.01 %</td><td>532.72 %</td><td>0.03 %</td><td>0.00 %</td></tr><tr><td>Average</td><td>4.31 %</td><td>33.25 %</td><td>0.00 %</td><td>66.59 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Maximum</td><td>34.50 %</td><td>33.26 %</td><td>0.00 %</td><td>66.75 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>Minimum</td><td>0.00 %</td><td>33.24 %</td><td>0.00 %</td><td>65.50 %</td><td>0.00 %</td><td>0.00 %</td></tr><tr><td>StDev</td><td>11.41 %</td><td>0.00 %</td><td>0.00 %</td><td>0.41 %</td><td>0.00 %</td><td>0 %</td></tr><tr><td>Avg/Max</td><td>0.13</td><td>1.00</td><td>0.95</td><td>1.00</td><td>0.69</td><td>1</td></tr></table>	<div>IE ID 3D 🔍 🖨️ 🌈 H ⏮ ⏭ 🇺🇦 ½</div>								Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others	THREAD 1.1.1	34.50 %	-	0.00 %	65.50 %	0.00 %	0.00 %	THREAD 1.1.2	0.00 %	33.24 %	0.00 %	66.75 %	0.00 %	-	THREAD 1.1.3	0.00 %	33.25 %	0.00 %	66.74 %	0.00 %	-	THREAD 1.1.4	0.00 %	33.25 %	0.00 %	66.74 %	0.00 %	-	THREAD 1.1.5	0.00 %	33.24 %	0.00 %	66.75 %	0.00 %	-	THREAD 1.1.6	0.00 %	33.26 %	0.00 %	66.74 %	0.00 %	-	THREAD 1.1.7	0.00 %	33.25 %	0.00 %	66.75 %	0.00 %	-	THREAD 1.1.8	0.00 %	33.25 %	0.00 %	66.74 %	0.00 %	-								Total	34.51 %	232.74 %	0.01 %	532.72 %	0.03 %	0.00 %	Average	4.31 %	33.25 %	0.00 %	66.59 %	0.00 %	0.00 %	Maximum	34.50 %	33.26 %	0.00 %	66.75 %	0.00 %	0.00 %	Minimum	0.00 %	33.24 %	0.00 %	65.50 %	0.00 %	0.00 %	StDev	11.41 %	0.00 %	0.00 %	0.41 %	0.00 %	0 %	Avg/Max	0.13	1.00	0.95	1.00	0.69	1
	<div>IE ID 3D 🔍 🖨️ 🌈 H ⏮ ⏭ 🇺🇦 ½</div>																																																																																																																							
		Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others																																																																																																																	
	THREAD 1.1.1	34.50 %	-	0.00 %	65.50 %	0.00 %	0.00 %																																																																																																																	
	THREAD 1.1.2	0.00 %	33.24 %	0.00 %	66.75 %	0.00 %	-																																																																																																																	
	THREAD 1.1.3	0.00 %	33.25 %	0.00 %	66.74 %	0.00 %	-																																																																																																																	
	THREAD 1.1.4	0.00 %	33.25 %	0.00 %	66.74 %	0.00 %	-																																																																																																																	
	THREAD 1.1.5	0.00 %	33.24 %	0.00 %	66.75 %	0.00 %	-																																																																																																																	
	THREAD 1.1.6	0.00 %	33.26 %	0.00 %	66.74 %	0.00 %	-																																																																																																																	
	THREAD 1.1.7	0.00 %	33.25 %	0.00 %	66.75 %	0.00 %	-																																																																																																																	
	THREAD 1.1.8	0.00 %	33.25 %	0.00 %	66.74 %	0.00 %	-																																																																																																																	
	Total	34.51 %	232.74 %	0.01 %	532.72 %	0.03 %	0.00 %																																																																																																																	
	Average	4.31 %	33.25 %	0.00 %	66.59 %	0.00 %	0.00 %																																																																																																																	
Maximum	34.50 %	33.26 %	0.00 %	66.75 %	0.00 %	0.00 %																																																																																																																		
Minimum	0.00 %	33.24 %	0.00 %	65.50 %	0.00 %	0.00 %																																																																																																																		
StDev	11.41 %	0.00 %	0.00 %	0.41 %	0.00 %	0 %																																																																																																																		
Avg/Max	0.13	1.00	0.95	1.00	0.69	1																																																																																																																		

SCHEDULING	EXECUTION TIME	SPEEDUP
static	 <p>par2301 Average elapsed execution time Sat Apr 11 14:57:23 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time Sat Apr 11 14:57:23 CEST 2020</p>
dynamic	 <p>par2301 Average elapsed execution time Sat Apr 11 14:56:51 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time Sat Apr 11 14:56:51 CEST 2020</p>
guided	 <p>par2301 Average elapsed execution time Sat Apr 11 14:56:39 CEST 2020</p>	 <p>par2301 Speed-up wrt sequential time Sat Apr 11 14:56:39 CEST 2020</p>

The worst of all executions has been the static one, we can appreciate huge differences when odd thread numbers are used because the thread that receives the middle set of the executions has to deal with the most intensive computation part.

When we compare the dynamic to the guided there are differences but not as big as the previous one.

The winner is the dynamic one. Probably due to the way of working that the guided has, which is to give every thread a group of tasks and the harder it gets to compute lesser tasks are given to the threads, so before the middle part it can give big chunks of tasks but when the middle part is selected it has to reduce the number of tasks given, but it has already given a big chunk of heavy computing to some threads, so the speedup is a bit worse.

CONCLUSION

With this report we have seen the importance of understanding the code we are parallelizing and the chosen granularity in order to minimize as much as possible the overheads.

The number of threads is usually way smaller than the number of potential tasks, so intuitively a very fine grain would generate a too big quantity because the associated overheads would be also too big.

Focusing into the mandelbrot set, as the different rows of the matrix can have even belonging points (fast to calculate) or non-belonging points (requiere reaching the maximum iterations), the execution time of each row can differ a lot, so a dynamic scheduling versus an static one can make the difference .