PARALLELISM

Laboratory 3:

Iterative task decomposition with OpenMP:

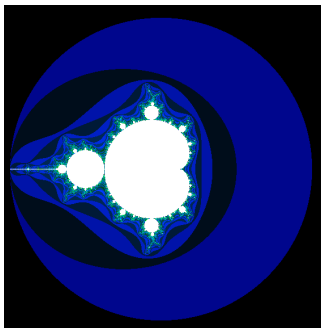
the computation of the Mandelbrot set

Daniel Muñoz Arroyo, par3110
10/11/2021
3rd, Spring semester

Index

1. Introduction

In this laboratory assignment the idea is to explore the tasking model in OpenMP of the code to compute the Mandelbrot set.



The Mandelbrot set is a particular set of points whose boundary generates a two-dimensional fractal shape. This image will be shown with the execution of the next command: “OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000”.
and the display shown by the option -d is the picture on the left.

The exploration will be with two different strategies:

- Point strategy: This technique consists of creating a task for every point.
- Row strategy: This technique consists of creating a task for every row.

Firstly, to explore the tasking model in OpenMP I had to do an analysis of the dependencies and if they have, protect it from some database problems like data race.
Secondly, we have to parallelize the code. It will be done with 3 methots.

- Method 1: use the #pragma omp task. I will call version 2 to this method
- Method 2: use #pragma omp taskloop. I will call version 2 to this method.
- Method 3: use #pragma omp taskloop nogroup. I will call version 3 to this method.

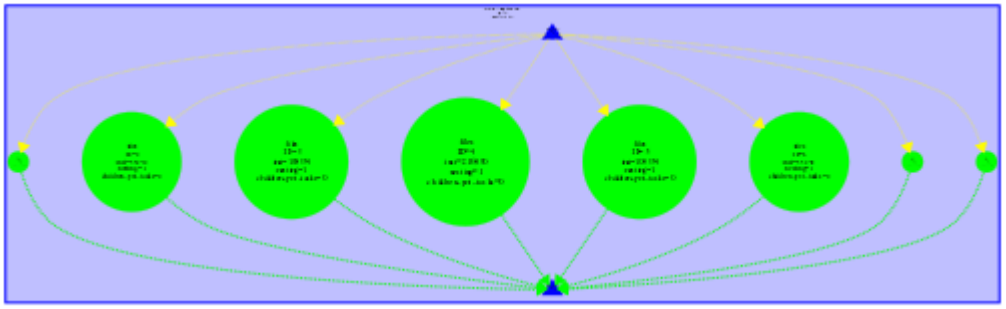
Finally, when I parallelize the two strategies with the 3 methods I will compare the performance of all the versions and strategies. And then I will conclude my deliverable.

2 Task decomposition and granularity analysis

2.1 Analysis row strategy

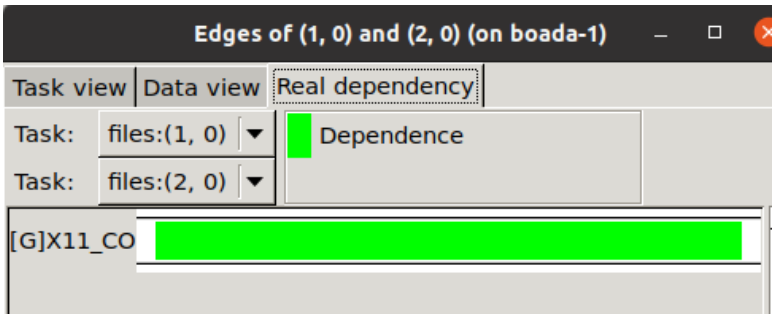
With this strategy, the idea is to create a task in every row computing all the points for all the columns in the row. To do it, I created a tareador task in every iteration of the row loop. This code is attached in the zip, with the name mandel-ompRowv1.c.

Now, if we compile the program with make and execute it with no options (I used the next command "./run-tareador.sh mandel-tar"). It execute the program in tareador and the task dependence graph is the next:



As we can see, we earn a lot of parallelization, because all the tasks are in the same level (no dependencies between them) except two of them. But the size of the tasks are not regular. The tasks don't have the same size because not all the points have to be calculated it depends on the row. I think in the do while is computing the color in every point of the row and not all the rows will compute all the points. That happens in all of the next TDG with the -d or -h option or without options.

If we execute "./run-tareador.sh mandel-tar -d" it will show in the tareador the TDG of the program with -d option and we can see it in the left image.
As we can see, the TDG is sequential. We can see it clicking with the right mouse button in a dependency line, dataview and edge, we can see on the left



As we can see in the picture, we can appreciate a dependence between the task in row 1 and the task in row 2. If I put the cursor in the name (I couldn't make a screenshot of it) the variable that makes the dependence and the path.
The path shows me that it is Xlib who is causing the dependencies. If we take a look at the code, with the -d option, an if is executing. It scales the color and display point, calling the Xlib and creating the dependencies. It will execute in every row, so the dependencies will be in all the tasks. For this reason, the TDG is sequential. We may protect it if we use a critical in the call of the functions that use the Xlib (XSetForeground and XDrawPoint).

if we execute with the -h option, the TDG is the next:

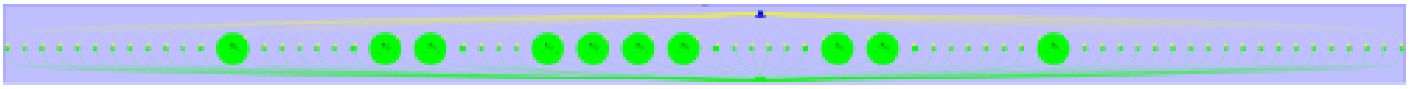
As we can see, it is sequential. If we take a look in the dependencies, it shows me that the dependencies are made in the heap when executes the next line code in main: histogram = malloc(maxiter*sizeof(int));

We may protect it using private variables of histogram.

2.2 Analysis point strategy

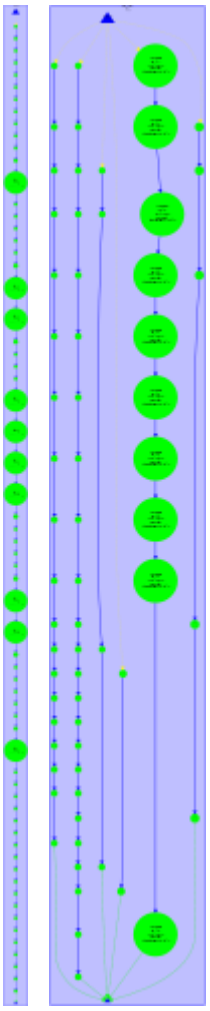
With this strategy, the idea is to create a task in every point. To do it, I created a tareador task in every iteration of the col loop. This code is attached in the zip, with the name mandel-tar_point.c.

Now, if we compile the program with make and execute it with no options (I used the next command "./run-tareador.sh mandel-tar"). It execute the program in tareador and the task dependence graph is the next:



As we can see, we earn a lot of parallelization, because all the tasks are in the same level (no dependencies between them). But the size of the tasks are not regular. The tasks don't have the same size because not all the points have to be calculated it depends on where are them. I think in the do while is computing the color in every point. That happens in all of the next TDG with the -d or -h option or without options.

if we execute with the -d option, the TDG on the right, the first one.
The dependencies are like in row strategy. In the code, the dependences are made by the use of the functions XSetForeground and XDrawPoint that use Xlib.



if we execute with the -h option, the TDG on the right, the second one.
The dependencies are like in row strategy. the dependencies are made in the heap when executes the next line code in main: histogram = malloc(maxiter*sizeof(int));
This strategy has more parallelisation than in the row strategy with -h option because we can parallelize something.
We may protect it when I try to parallelize the code with private histogram.

3. Parallelisation Strategy with OpenMP

The parallelisation that I will use is the point and row strategy, but in both cases I need to respect the dependencies. I did it like it's shown in the next picture.

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

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

The picture above is the common part of all versions to guarantee the dependencies. The atomic is to protect the dependencies in the histogram (with -h option) and the critical is to protect the calls to the functions XSetForegrond and XDrawPoint in the display (with -d option). With that, the dependencies are respected.

3.1 Point decomposition with OpenMP

In this strategy, to ensure that the display is correct and it works well, I executed the command "OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000" that shows the execution of the code with 1 thread and I obtain the same image (shown with -d option) than with 2 threads with the command "OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000".
It happens for all the versions, so it means that the parallelization of the code is right.

3.1.1 Point decomposition with OpenMP (v1)

In this version, we have to parallelize the code with #pragma omp task.
With this strategy, there is an implicit barrier that is in the single construct and a thread master will create the tasks. Then the other threads wait for synchronization to execute the tasks.

3.1.2 Point decomposition with OpenMP (v2)

In this version, we have to parallelize the code with taskloop instead of task. If I don't put the grainsize or numtasks, the taskloop will divide the for in num_threads (in this case in 8 taks and the grainsize will be 10000/8 = 1250 points by task) it will get less synchronization time in the scalability.

To do this version I only add a #pragma omp taskloop firstprivate (row) between the two for. This code is attached with the name “mandel-ompPointv2.c”

3.1.3 Point decomposition with OpenMP (v3)

In this version, the teacher said to add a nogroup to get out the implicit barrier in the taskloop. I did the modification adding the nogroup clause in taskloop firstpirvate(row) like this taskloop firstpirvate(row) nogroup the code is attached with the name mandelPointv3.c.

3.2 Row decomposition

In this strategy, to ensure that the display is correct and it works well in all the versions parallelizing the mandelbrot set, I executed the command “OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000” that shows the execution of the code with 1 thread and I obtain the same image (shown with -d option) than with 2 threads with the command “OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000”.
It happens for all the versions, so it means that the parallelization of all the versions are right.

3.2.1 Row decomposition with OpenMP (v1)

In this version, we have to parallelize the code with a row strategy by adding #pragma omp task. So I did it adding #pragma omp task between the 2 loops, creating a task for every iteration of the external loop (one task by row). For more information, you can see the mandelRowv1.c attached with all the code.

3.2.2 Row decomposition with OpenMP (v2)

In this version, we have to parallelize the code with a row strategy by adding #pragma omp taskloop. So I did it adding #pragma omp task up the external loop, creating a task for every iteration of the external loop (one task by row). For more information, you can see the mandelRowv2.c attached with all the code.

3.2.3 Row decomposition with OpenMP (v3)

In this version, the teacher said to add a nogroup to get out the implicit barrier in the taskloop. I did the modification adding the nogroup clause in taskloop firstprivate(row) like this taskloop firstprivate(row) nogroup the code is attached with the name mandelRowv3.c.

4. Performance analysis

To understand the performance, I will take some screenshots.

In both strategies, the plots that I will show were made executting the command “sbatch submit-strong-omp.sh” and then the command “gs nameFileCreated.ps”.

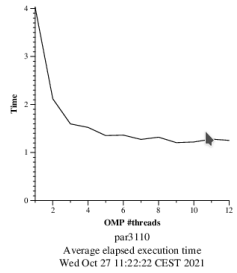
The screenshots that I will show for both strategies were done with paraver.

One type of the screenshots I did went to hints -> OpenMp task -> Profile of explicit task creation and execution. It creates a table that shows the tasks that create the taskloop.

The second type of screenshots I did went to hint -> thread state profile (efficiency). It creates a table showing the % of time that expends the execution with three columns (one for the % of running time, the next column for the % of synchronization time and the last column the % of time spent for the creation of the taks).

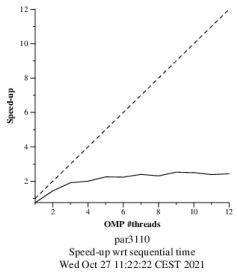
4.1 Point strategy

4.1.1 Point performance analysis v1



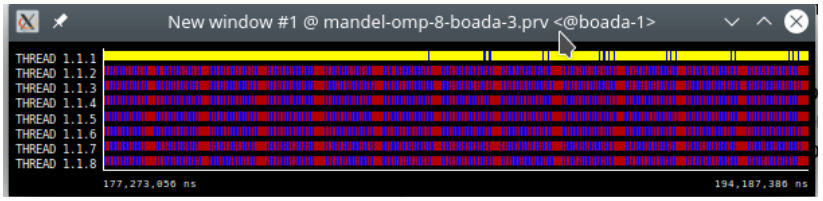
With “sbatch submit-strong-omp.sh” and “gs mandel-omp-10000-strong-omp.ps” we can see the next picture that is the strong scalability plot:

As we can see, the scalability is not so good with 3 or more processors than with 2 because if we increment the number of threads, the synchronization time increases too.

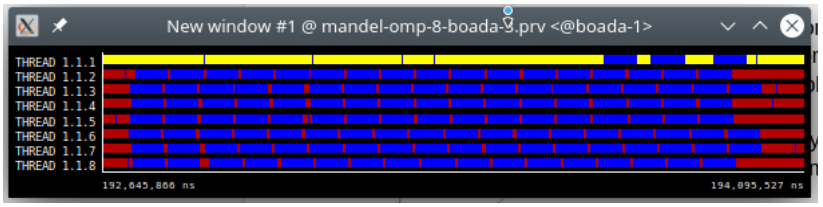


With “sbatch submit-extrae.sh mandel-omp 8” and “mandel-omp-8-boada-3.prv” we can see it with paraver.

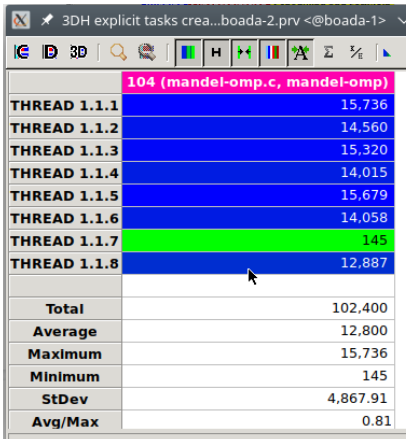
If we zoom in the timetable, we will see the next picture:



As we can see, it shows the execution of some iterations of the internal loop.



We can see that in every iteration (the picture above), it needs a bit of synchronization time and a big synchronization time between the iterations. It is caused by the dependencies. For that reason, the scalability is not so good with 3 or more processors than with 2, but is better than sequential.



	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	15,736
THREAD 1.1.2	14,560
THREAD 1.1.3	15,320
THREAD 1.1.4	14,015
THREAD 1.1.5	15,679
THREAD 1.1.6	14,058
THREAD 1.1.7	145
THREAD 1.1.8	12,887
Total	102,400
Average	12,800
Maximum	15,736
Minimum	145
StDev	4,867.91
Avg/Max	0.81

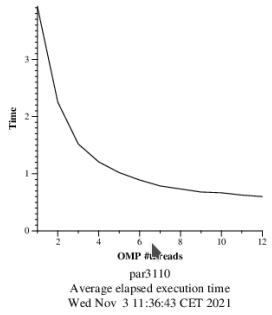
As we can see in the left picture, the number of tasks created in this version are 102400

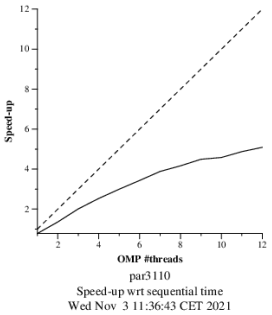
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	47.24 %	0.00 %	52.76 %
THREAD 1.1.2	30.35 %	69.64 %	0.00 %
THREAD 1.1.3	30.26 %	69.74 %	0.00 %
THREAD 1.1.4	30.18 %	69.81 %	0.00 %
THREAD 1.1.5	30.61 %	69.39 %	0.00 %
THREAD 1.1.6	30.36 %	69.64 %	0.00 %
THREAD 1.1.7	30.47 %	69.52 %	0.00 %
THREAD 1.1.8	30.31 %	69.69 %	0.00 %
Total	259.79 %	487.44 %	52.77 %
Average	32.47 %	60.93 %	6.60 %
Maximum	47.24 %	69.81 %	52.76 %
Minimum	30.18 %	0.00 %	0.00 %
StDev	5.58 %	23.03 %	17.45 %
Avg/Max	0.69	0.87	0.13

To quantify the execution time and the synchronization time, I made a screenshot of the table shown on the left.

As we can see, the average of the execution time is 32,47 % the other part is synchronization time (60,93%) and creation of tasks (6,6%).

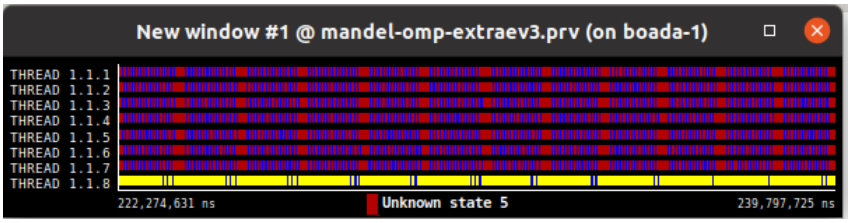
4.1.2 Point performance analysis v2





As we can see, the scalability is increasing when we increase tooo the number of threads. If we compare it with the scalability plot of the version 1, this is better.

To understand why it is better, I made a screenshot of some executions of the internal loop as we can see below.



If we compare the execution of the version 1 with the 2 is more or less the same, but it can be better compared.

3DH explicit tasks cre...oada-2.prv <@boada-1>

	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	4,187
THREAD 1.1.2	2,982
THREAD 1.1.3	3,065
THREAD 1.1.4	3,016
THREAD 1.1.5	3,172
THREAD 1.1.6	2,990
THREAD 1.1.7	3,102
THREAD 1.1.8	3,086
Total	25,600
Average	3,200
Maximum	4,187
Minimum	2,982
StDev	377.70
Avg/Max	0.76

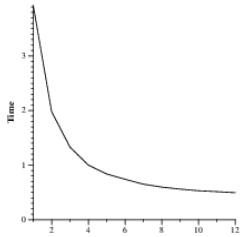
The task creation is less than in the version 1 because there are less tasks.
In the version 1 there were a task for every iteration of the internal for (is the same as one task by point), but now, there are in a task more than one point. To justify it, we can compare the number of tasks in boths versions. So there are 25600 tasks in this version and 102400 in the last version, so now, the taskloop did 102400/25600 = 4 points by taks.

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

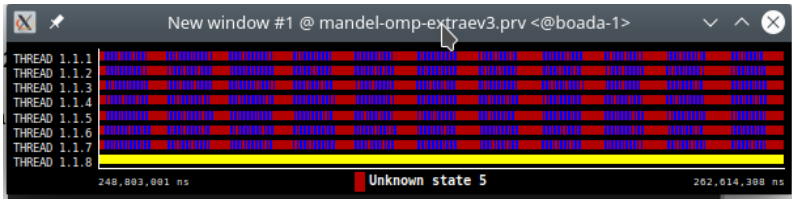
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	54.60 %	25.38 %	20.02 %
THREAD 1.1.2	67.87 %	32.12 %	0.01 %
THREAD 1.1.3	69.50 %	30.49 %	0.01 %
THREAD 1.1.4	69.46 %	30.54 %	0.01 %
THREAD 1.1.5	69.71 %	30.29 %	0.01 %
THREAD 1.1.6	68.71 %	31.28 %	0.01 %
THREAD 1.1.7	68.58 %	31.42 %	0.01 %
THREAD 1.1.8	68.37 %	31.63 %	0.01 %
Total	536.80 %	243.14 %	20.06 %
Average	67.10 %	30.39 %	2.51 %
Maximum	69.71 %	32.12 %	20.02 %
Minimum	54.60 %	25.38 %	0.01 %
StDev	4.76 %	1.99 %	6.62 %
Avg/Max	0.96	0.95	0.13

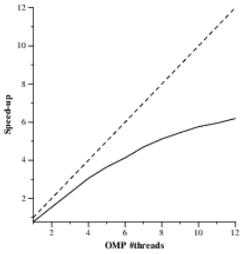
As we can see in the left picture, the percentage of the execution when running the computation is 67,10% that is better than the version 1, the % of synchronization time is 30,39% and the tasks creation time is 2,51%.
It is better than the last version because if we have less tasks, we will need less time to synchronize them and less task creation.

4.1.3 Point performance analysis v3



We can see the picture on the left that is the strong scalability plot. As we can see, the scalability plot is better than the last version.





The picture above is the execution of some iterations. As we can see, is more or less the same as the version 2, but it can be better compared.

3DH explicit tasks cre...oada-2.prv <@boada-1>

	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	3,165
THREAD 1.1.2	3,204
THREAD 1.1.3	3,319
THREAD 1.1.4	3,228
THREAD 1.1.5	3,249
THREAD 1.1.6	3,174
THREAD 1.1.7	3,094
THREAD 1.1.8	3,167
Total	25,600
Average	3,200
Maximum	3,319
Minimum	3,094
StDev	62.84
Avg/Max	0.96

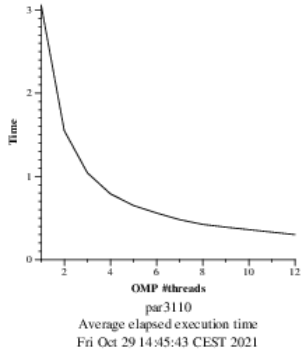
The number of tasks will be the same because the taskloop will compute 4 points by tasks as it did in the version 2. But if we have a look in the left screenshot of the tasks that execute every thread, it is better balanced because the minimum tasks of one thread is 3094 and the maximum 3249 so a difference of 3249-3094 = 155 tasks. And in the last version, there is a difference of 4187 - 2982 = 1205 tasks. It doesn't say nothing because one task can have more cost than others.

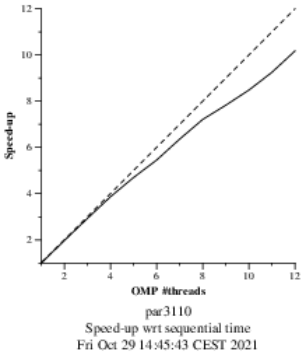
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	89.73 %	10.25 %	0.02 %
THREAD 1.1.2	89.11 %	10.88 %	0.01 %
THREAD 1.1.3	89.24 %	10.75 %	0.01 %
THREAD 1.1.4	89.16 %	10.84 %	0.01 %
THREAD 1.1.5	89.54 %	10.46 %	0.01 %
THREAD 1.1.6	89.29 %	10.70 %	0.01 %
THREAD 1.1.7	80.41 %	0.02 %	19.57 %
THREAD 1.1.8	89.38 %	10.61 %	0.01 %
Total	705.85 %	74.51 %	19.64 %
Average	88.23 %	9.31 %	2.45 %
Maximum	89.73 %	10.88 %	19.57 %
Minimum	80.41 %	0.02 %	0.01 %
StDev	2.96 %	3.52 %	6.47 %
Avg/Max	0.98	0.86	0.13

As we can see in the left picture, all the threads are executing more or less the same part of the (89% instead thread 7 with 80 %). It is a 7% of difference. In the version 1, the percentage is 69%-54% = 15%. For that reason. this version is better than the last one because with the nogroup directive is better balanced than if we don't use it.

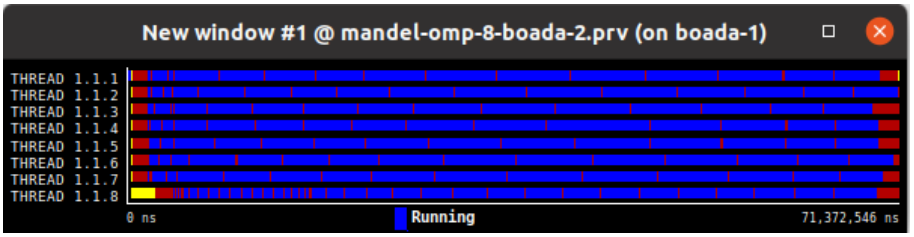
4.2 Row strategy

4.2.1 Row performance analysis v1





As we can see in the left picture of the scalability plot, this version is better than the point strategy. I will try to justify why is it.



We can see in above the screenshot of the paraver execution. With this picture we can see that there are less synchronization than in the previous strategy because if I don't do any zoom, we can see the execution of the mandelbrot set instead of all red by the synchronization time. So now we have less synchronization time.

	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	31
THREAD 1.1.2	25
THREAD 1.1.3	30
THREAD 1.1.4	25
THREAD 1.1.5	30
THREAD 1.1.6	26
THREAD 1.1.7	126
THREAD 1.1.8	27
Total	320
Average	40
Maximum	126
Minimum	25
StDev	32.58
Avg/Max	0.32

As we can see in the left picture that shows the table of the tasks created by every thread, the total of the tasks created are 40. This implies that there are 40 rows because every task is the computation of all the points in a row.

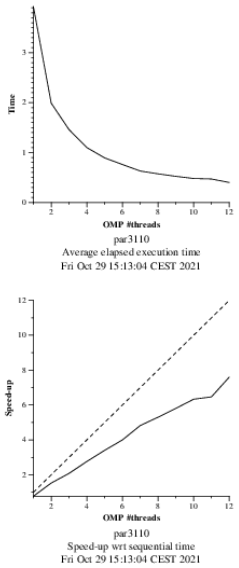
2D thread state profile @ mandel-omp-8-boada-2.prv <@boada-1>

Default

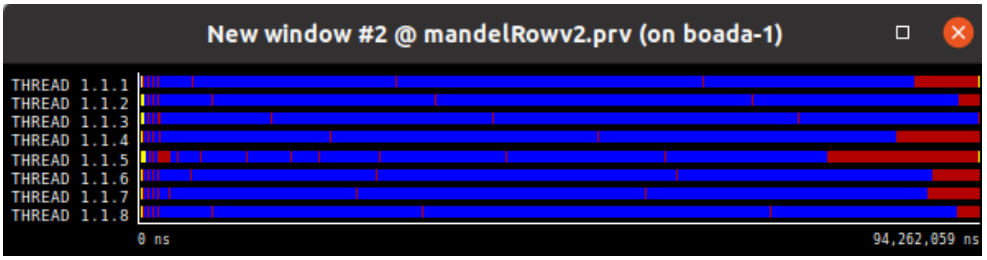
	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	96.04 %	3.93 %	0.04 %
THREAD 1.1.2	96.64 %	3.34 %	0.02 %
THREAD 1.1.3	96.20 %	3.78 %	0.02 %
THREAD 1.1.4	96.91 %	3.08 %	0.01 %
THREAD 1.1.5	98.22 %	1.77 %	0.01 %
THREAD 1.1.6	97.02 %	2.97 %	0.01 %
THREAD 1.1.7	94.80 %	3.07 %	2.13 %
THREAD 1.1.8	99.64 %	0.35 %	0.01 %
Total	775.47 %	22.29 %	2.23 %
Average	96.93 %	2.79 %	0.28 %
Maximum	99.64 %	3.93 %	2.13 %
Minimum	94.80 %	0.35 %	0.01 %
StDev	1.37 %	1.11 %	0.70 %
Avg/Max	0.97	0.71	0.13

As we can see in the left picture that shows the table of the % of the time of the 3 things that intervent on it (the things are in the column: % of time that are executing the tasks, the % of time executing the synchronization time and the time spent in the creation of the tasks) it has very less synchronization time only 2,79%, the 96,93 % are executing the tasks and only the 0,28% to create tasks.

4.2.2 Row performance analysis v2



The left picture shows the elapsed time and scalability plot of this version. As we can see, it has less scalability than the last version.
As we can see above, if we have a look at the timetable made in paraver, we can't conclude anything.



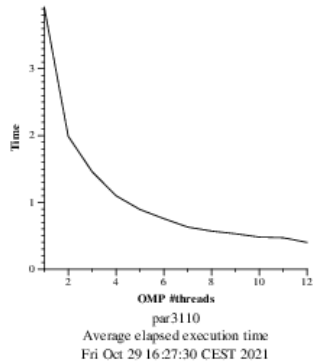
	102 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	31
THREAD 1.1.2	7
THREAD 1.1.3	7
THREAD 1.1.4	7
THREAD 1.1.5	7
THREAD 1.1.6	7
THREAD 1.1.7	7
THREAD 1.1.8	7
Total	80
Average	10
Maximum	31
Minimum	7
StDev	7.94
Avg/Max	0.32

As we can see in the left picture that shows the table of the tasks created by every thread, the total of the tasks created are 80. Now, the taskloop is creating 1 task by the computation of all the points in a half row.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	81.03 %	18.63 %	0.33 %
THREAD 1.1.2	92.02 %	7.97 %	0.01 %
THREAD 1.1.3	93.37 %	6.62 %	0.01 %
THREAD 1.1.4	97.21 %	2.78 %	0.01 %
THREAD 1.1.5	93.47 %	6.52 %	0.01 %
THREAD 1.1.6	99.84 %	0.15 %	0.01 %
THREAD 1.1.7	96.64 %	3.35 %	0.01 %
THREAD 1.1.8	89.70 %	10.29 %	0.01 %
Total	743.29 %	56.32 %	0.39 %
Average	92.91 %	7.04 %	0.05 %
Maximum	99.84 %	18.63 %	0.33 %
Minimum	81.03 %	0.15 %	0.01 %

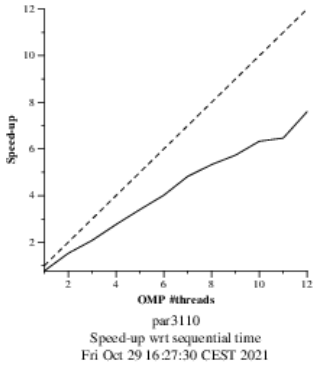
As we can see in the left picture that shows the table of the % of the time and the % of synchronization time and the % of the time creation of the tasks increased.
It is because we have more tasks than before and for this reason we will have to create more tasks, so the % of time spend creating tasks and the synchronization time will increase so the % of synchronization time increases too.

4.2.3 Row performance analysis v3



To see the gain, we can see the picture on the left that shows the time and speed up plot. As we can see, the speed up plot is the same as the last version.

The execution in the timeline is the same as before, so I can't conclude anything. For the table of the tasks created, there are the same tasks as in the version 2.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	93.78 %	6.19 %	0.03 %
THREAD 1.1.2	97.70 %	2.29 %	0.01 %
THREAD 1.1.3	99.86 %	0.13 %	0.01 %
THREAD 1.1.4	90.48 %	9.52 %	0.01 %
THREAD 1.1.5	94.07 %	5.92 %	0.01 %
THREAD 1.1.6	97.69 %	2.30 %	0.01 %
THREAD 1.1.7	81.57 %	18.19 %	0.24 %
THREAD 1.1.8	92.86 %	7.13 %	0.01 %
Total	748.01 %	51.67 %	0.32 %
Average	93.50 %	6.46 %	0.04 %
Maximum	99.86 %	18.19 %	0.24 %
Minimum	81.57 %	0.13 %	0.01 %
StDev	5.34 %	5.28 %	0.07 %
Avg/Max	0.94	0.36	0.17

We can see that the table is practically the same percentage of the last table. The percentage can change a bit executing in paraver.

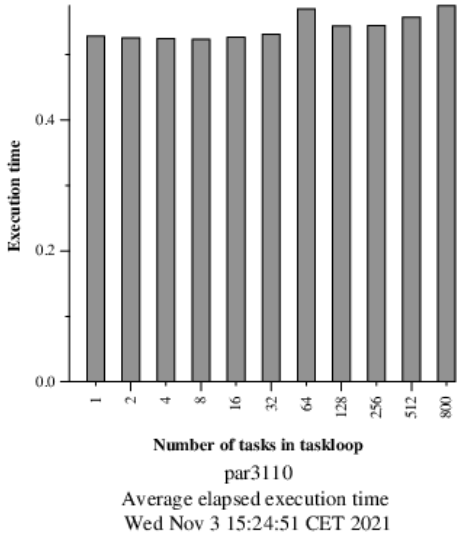
Finally, the execution with nogroup and without it is the same for this strategy by the dependencies.

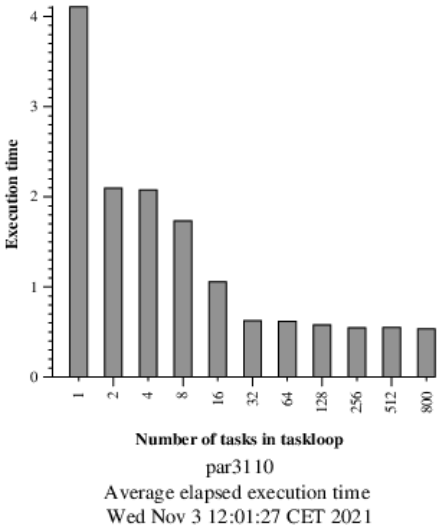
5. Conclusion

If we have to compute the mandelbrot set, the best way to do it is with a grain size of 1 task by row. It can be possible using the version 1 of row strategy that consists tof do a task in every iteration of the external loop, computing in every task 1 row.
This is the best way of all the versions in this laboratory because it has the best scalability plot and it implies that it has the best performance.
It is possible because it has the smallest number of tasks and it is balancet. With that, it can reduce the synchronization time and the creation time because it has less tasks to synchronize and less task to create and consequently with this way will spend 96% of the time to execute the mandel-omp.c.

6. Optional

The script submit-numtasks-omp.sh gives to mandel-omp.c the number of tasks with the -u option and in the mandel-omp.c, it keep the value in a variable named user_param. The only thing that I did was add a num_task(user_param) in the taskloop directive like this #pragma omp taskloop num_task(user_param) firstprivate(row) nogroup for the point strategy for the Point strategy and #pragma omp taskloop num_task(user_param) nogroup for the row strategy. For more information, the codes are attached with the names optionalRownogroup.c and optionalPointnoroup.c.
With those modification, we can sent to boada the script indicating the number of threads with the command “sbatch submit-numtasks-omp.sh 8” (I sent with 8 threads) and the respective graphics diagram are the nexts:





On the left we have the Point strategy and on the right we have the Row strategy.
As we can see for the Row, if we increment the tasks number, we will have more parallelism. If every point takes the same time to compute, the ideal number of tasks that we should have is 8, the same as threads. But it is not happening because the time to compute a point can be different than if we take another point. For this reason, with 8 tasks is not the best because the distribution of tasks will be unbalanced. To balance it, we have to create more tasks and for that reason, the time is decreasing. But if we create so many tasks, as we saw in the Point strategy in version 1 (102400 tasks), the synchronization time will take a lot of the % of the execution (in that case was 60%) and the tasks creation time too (was 6,6%).
For the row strategy, the best grain size is one that has the minimum tasks to have parallelism to execute the program between all the threads and can be balanced between them.

in the right picture above shows the execution time in Point strategy of with one to 800 tasks in the internal taskloop, so it implies that is not executing 1 to 800 tasks, but in real it will executes 1 to 800 tasks by iteration As we can see, when we increment the tasks to 8, the time execution is decreasing but is practically insignificant. If we create more than 8 tasks, the execution time is increasing because there are a lot of tasks and by consequence, there will be more synchronization and tasks creation time. This strategy is not so good to execute with a lot of tasks because the best size is a task for all the internal for and it is the same as compute a task by row in Row strategy.
For the Point strategy, the best grain size is the coarsest grain size because it generates a lot of tasks and it implies a lot of synchronization and task creation time

Finally, I can conclude that the best strategy to compute the mandelbrot set is the row strategy because it has the minimum tasks to have the threads executing approximately Total time/N threads in every thread.