PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

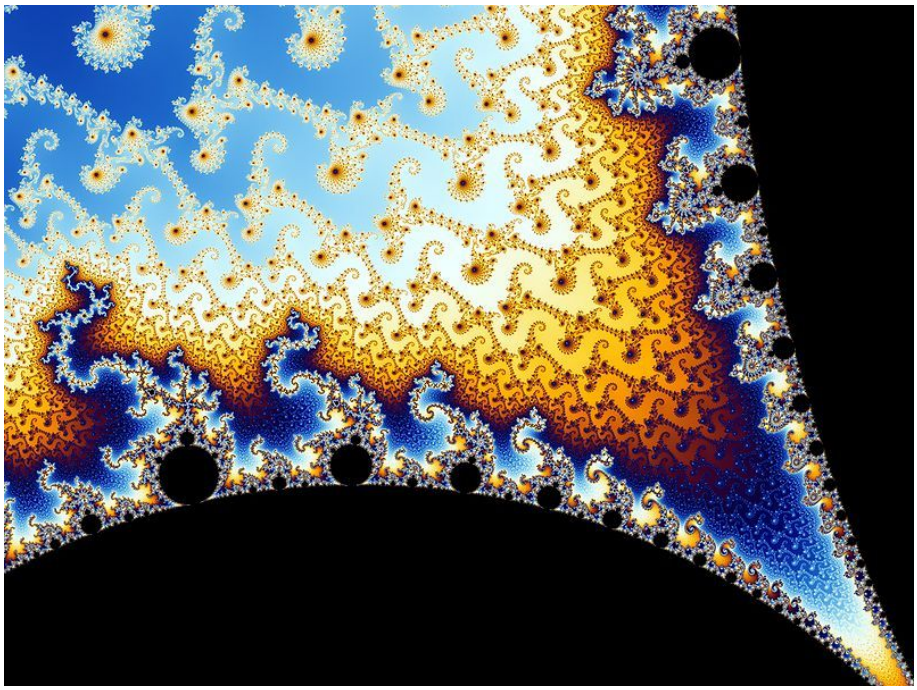
PAR 2312
Ana Mestre Borges
Fèlix Arribas Pardo

1. Introduction

In this laboratory session we are going to practice and observe different parallelization strategies on a matrix: Row and Point.

Not only the size of the parallelized element. We also tested different Open-MP ways to parallelize a for-loop: `task`, `taskloop` and `for`.

In addition, the loop that we are executing have different execution time depending on the **Mandelbot**.



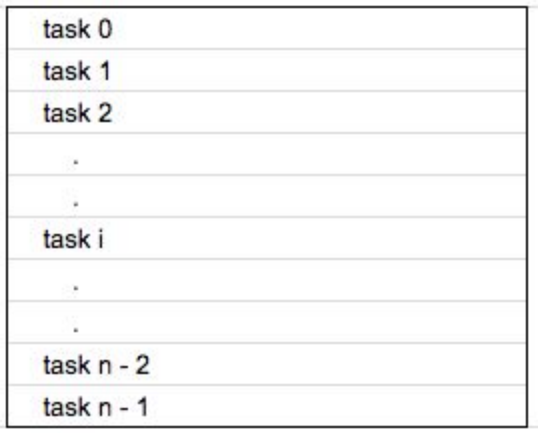
2. Parallelization strategies

Row decomposition strategy

When we use Row strategy, we are creating one task for each row of the matrix.

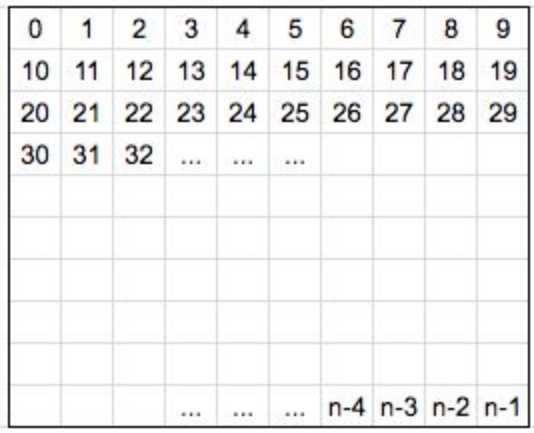
On the one hand, this makes bigger tasks. This, sometimes may be a problem: Assuming that the tasks have different work to do, if we have few rows we could not have enough parallelization. One row (thread) doing a lot of work and the rest waiting that to finish.

On the other hand the schedule, fork and join overhead is smaller. In comparison with the point strategy, we have way much less tasks to manage.



Point decomposition strategy

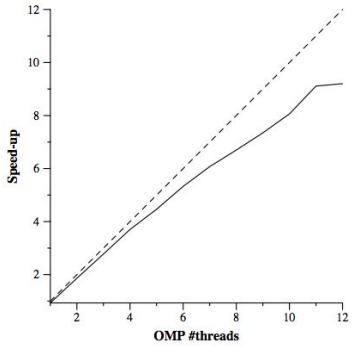
With Point strategy, we are creating one task for each element of the matrix. Tasks are smaller, so there is no much difference in each thread's execution time.



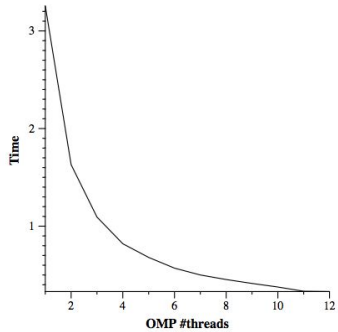
The problem that we might find in this strategy is that, if the matrix is so big, we will have a lot of tasks to manage. There will be a lot of overhead scheduling those tasks, forking them into the different threads and joining them back.

3. Preformance evaluation

To evaluate both strategies with `task` and `taskloop` clauses we used the `submit-strong-omp` script which basically executes three times the code sequentially and then does twelve OpenMP executions with 1, 2, 3, ... until 12 threads, obtaining the elapsed time of all executions and the speedup in a text document. It also creates two plots, one for the elapsed time and another for the speedup for different number of threads.

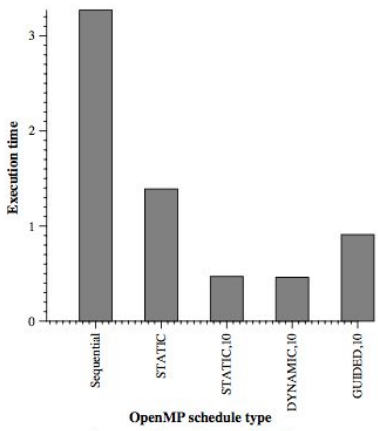


Speed-up wrt sequential time

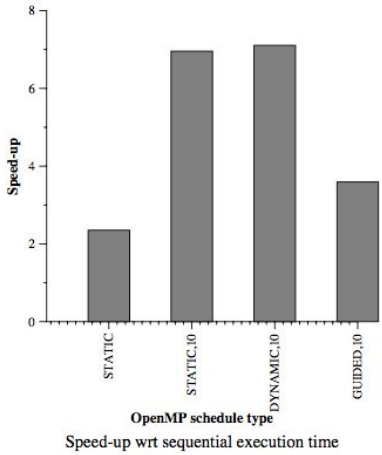


Average elapsed execution time

When using the for clause, we used the `submit-schedule-omp` script. This one executes the program with a different schedule options: `STATIC`, `STATIC 10`, `DYNAMIC 10` and `GUIDED 10`. Instead of creating the plots of the elapsed time and the speedup from the sequential version with different number of threads, this script generate the plot of the execution time and the speedup of the sequential version and the four different schedule options to compare them properly.



Average elapsed execution time



Speed-up wrt sequential execution time

4. Conclusions

Before all the executions and all the different results of the deliverable we have seen that the best way to execute this code in parallel with **8 threads** is `task` or `taskloop`-based parallelization with **row** strategy. We thought, at the beginning of the session that the point strategy would work better.

We also concluded a very important thing for the future when parallelizing code:

Every time we have to parallelize some code we find different ways to do that: *Where do we create tasks? Tree or leaf strategy? Row or point strategy? Row or column? task, taskloop or for? static, dynamic or guided?*

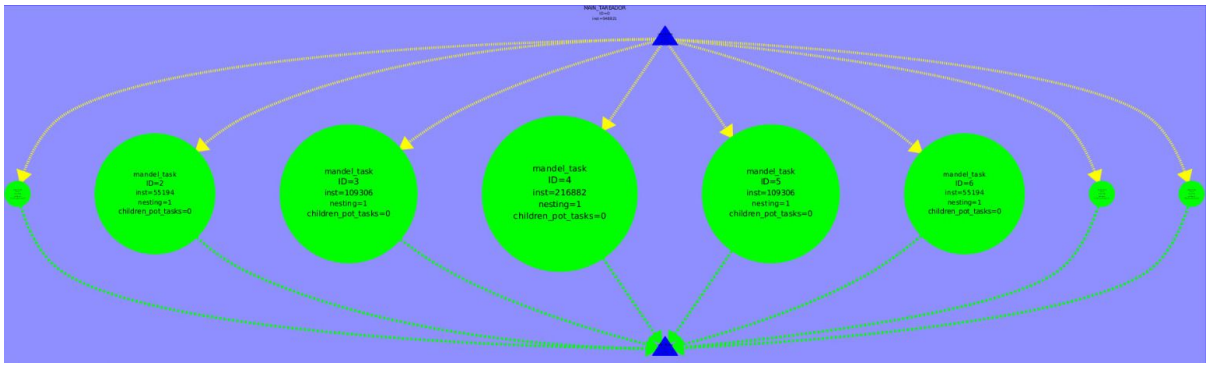
All programs are different and depending on the code you are trying to parallelize you are using one strategy and a specific clause or another. The best way to find the best parallel performance is trying a lot of different ways.

5. Deliverable

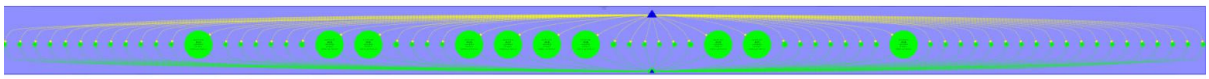
5.1 Task granularity analysis

1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for the non-graphical version of mandel-tareador? Obtain the task graphs that are generated in both cases for -w 8.

Neither row nor point granularities have dependencies in the main loop of **Mandelbrot**.
Also, each task that executes the same code, have different amount of work to do. You can observe this in the tasks graphs. The for-loop iterations are the same, but the instructions executed are not. In fact, we observe a huge difference.



mandel-tareador non-graphical version with ROW grnularity

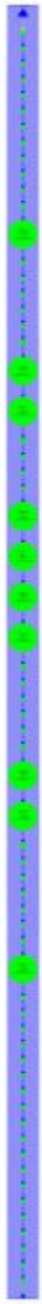


mandel-tareador non-graphical version with POINT grnularity

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How do you plan to protect this section of code in the parallel OpenMP code?

When we want to draw it (`_DISPLAY_`) the win (`Window`) and gc (`GC`) variables generate dependencies. This code fragment can be protected with a `#pragma omp critical` clause.

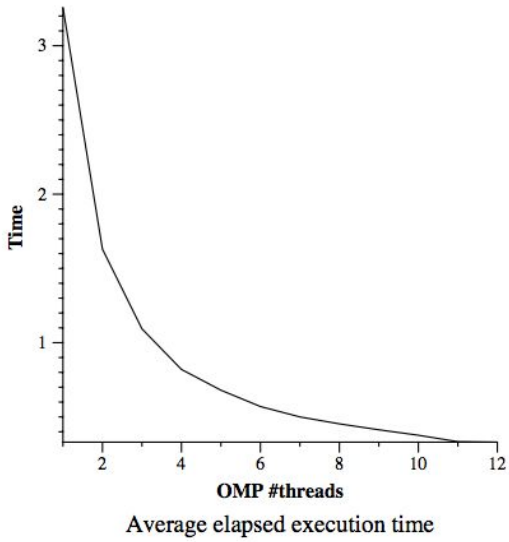
graphical version with **row** grnularity



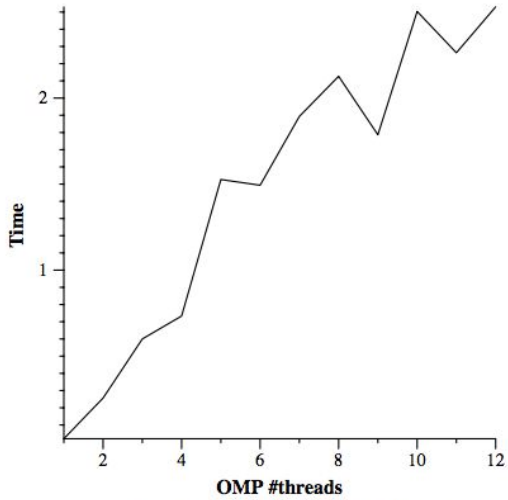
graphical version with **point** grnularity

5.2 OpenMP `task`–based parallelization

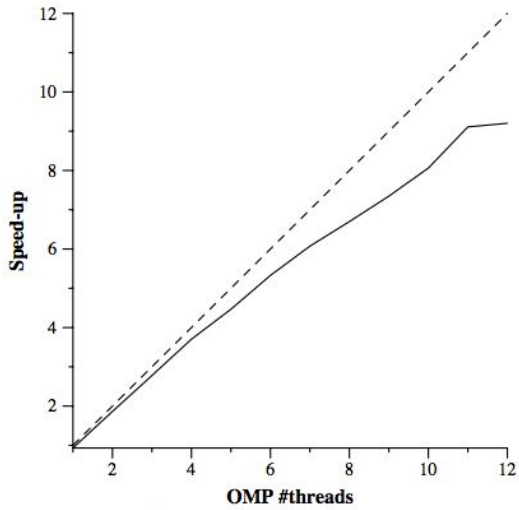
1. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.



Average elapsed execution time

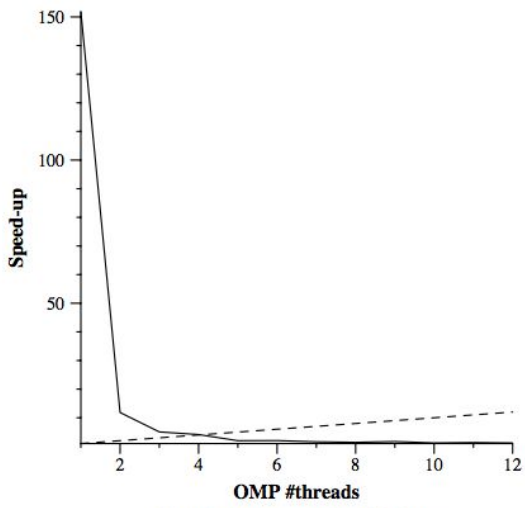


Average elapsed execution time



Speed-up wrt sequential time

`task`-based row



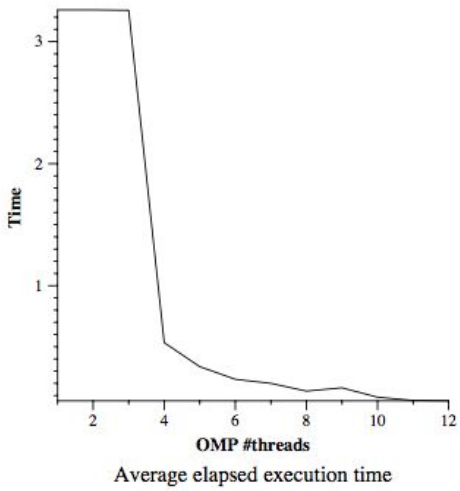
Speed-up wrt sequential time

`task`-based point

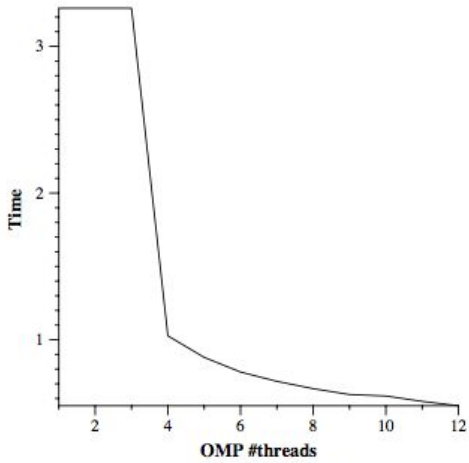
We can see that with a few threads the **point** strategy works better than **row**. But when we increment the OMP #threads, splitting task in **rows** has a great performance. That is because when we are using **points** the `critical` part needs to sync their threads.

5.3 OpenMP `taskloop`–based parallelization

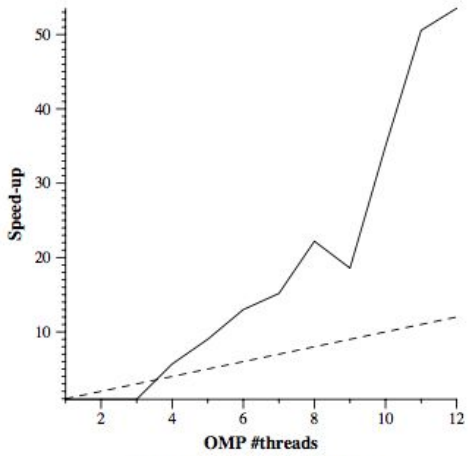
1. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.



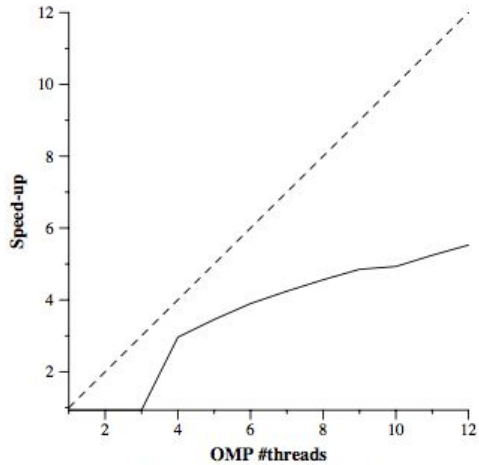
Average elapsed execution time



Average elapsed execution time



Speed-up wrt sequential time



Speed-up wrt sequential time

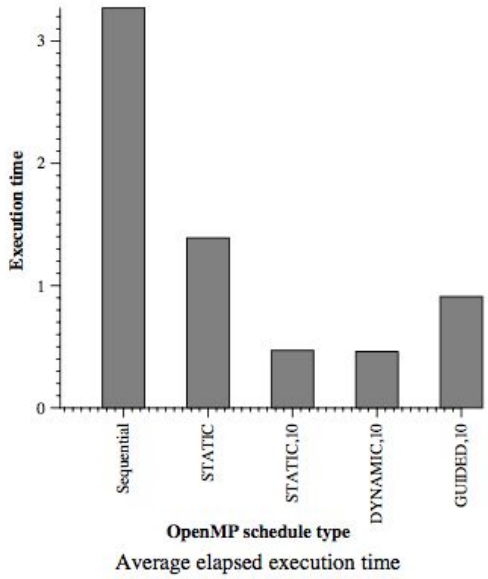
`taskloop`-based row

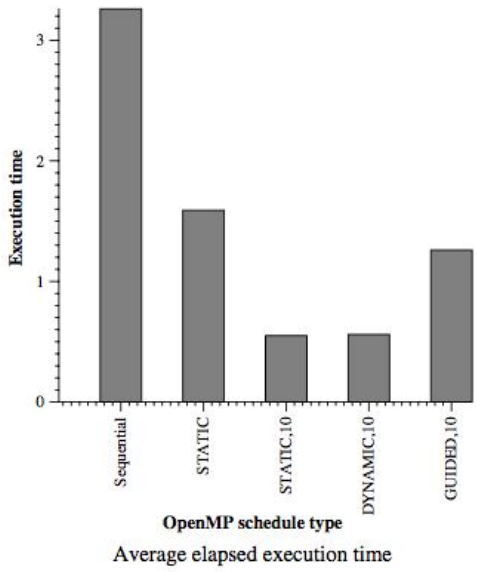
`taskloop`-based point

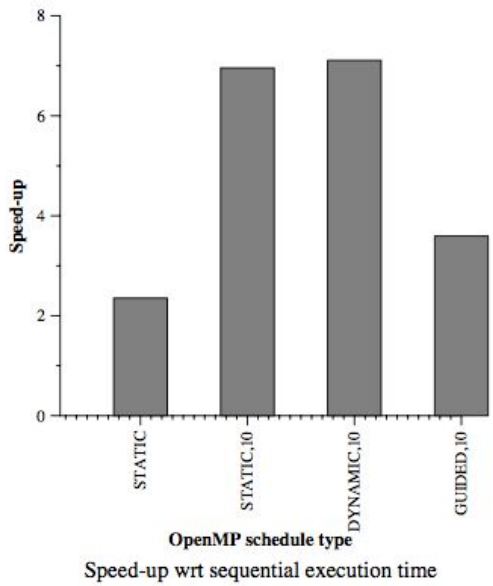
If we compare `taskloop` with `task` we can se that `taskloop` starts getting better when we have a lot of threads (more than 6). If we compare **row** and **point** decomposition we observe the same problem that we had before: **point** decomposition need to sync its threads much more than **row** decomposition.

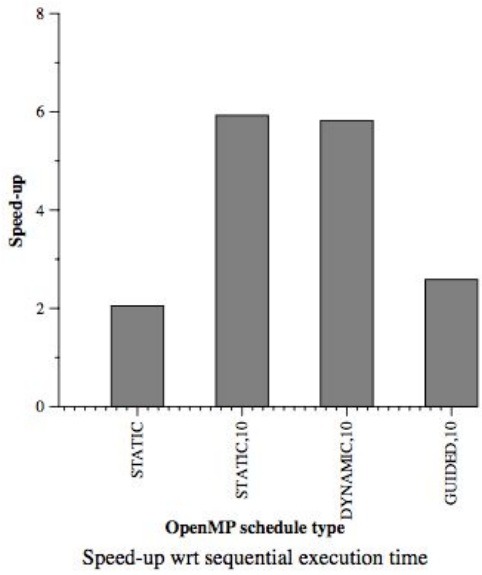
5.4 OpenMP `for`–based parallelization

1. For the the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots that have been obtained for the 4 different loop schedules when using 8 threads (with -i 10000). Reason about the performance that is observed.









`for`-based row

`for`-based point

In this case we don't see much difference between **row** and **point** decomposition, that is because the strategy does not force the synchronization and it does not lose so much time then.

2. For the Row parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions (with 8 threads and -i 10000) and analysis with Paraver, reasoning about the results that are obtained.

	static	*static, 10*	*dynamic, 10*	*guided, 10*
Running average time per thread	1.411,75 µs	1.862,7 µs	802,80 µs	1.085,6 µs
Execution unbalance (average time divided by maximum time)	0,45	0,85	0,88	0,84
SchedForkJoin (average time per thread or time if only one does)	1.429.266 µs	27.799 µs	7.728 µs	833.195 µs

To get the running average per thread we used `OMP parallel functions duration histogram` configuration, to get the execution unbalance we used the `OMP in scheduleforkjoin` configuration and then created a new Histogram.

We can see that the dynamic-10 version is faster than rest thanks to the Schedule, Fork and Join overhead time. It is way much lower than the rest of the strategies. Also, it is interesting that static-10 is faster than static and guided-10 in SchedForkJoin but way much slower in the average execution time per thread. But thanks that SchedForJoin performance it is better than the other two, we can see this in the plots of the exercise 5.4.1.