PAR — Deliverable for Laboratory Session 3

Víctor Jiménez Arador, Miguel Moreno Gómez

Group 12_{02} — Fall semester 2017/2018

Contents

1 Introduction **3**

2 Parallelization strategies **3**

3 Performance evaluation **3**

4 Task granularity analysis **4**

5 `OpenMP task`-based parallelization **8**

6 `OpenMP taskloop`-based parallelization **10**

7 `OpenMP for`-based parallelization **12**

8 Conclusions **17**

1 Introduction

In this laboratory session we have observed how to use practically OpenMP to parallelize an embarrassingly parallel program (that is, one that has little to no dependencies with the rest of the code), The **Mandelbrot set**.

To do so, we have paralellized using two strategies: *Row*-based and *Point*-based, each using several OpenMP directives: `task`, `taskloop` and `for`.

2 Parallelization strategies

There are two of them in this deliverable:

- **Row-based:** Tasks are composed of rows of a matrix; there are fewer tasks, but they are quite big! This strategy is ideal when using not too many cores and when we want to minimize forking-and-joining.
- **Row-based:** Tasks are composed of each element of the matrix: there are lots and lots of tasks, but the size per task is smaller, so the overhead per each task created will be greater than in the Row-based parallelization. This strategy is ideal when using lots of cores.

3 Performance evaluation

Performance is evaluated in two ways:

- **Strong scalability:** All problems are executed between 1..12 cores to evaluate the speed-up compared to the sequential time and the average time elapsed per core execution. It's a standard benchmark we can use for all strategies. This generates a drawn plot.
- **Scheduling:** Using the for directive we evaluate the scheduling of both strategies using four different options: STATIC, STATIC-10, DYNAMIC, GUIDED-10. Unlike the scalabity plot, this one is line-based.

4 Task granularity analysis

4.1 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`.

The two most important characteristics in common of the task graphs are:

- There is no independence between rows and columns: the only dependences found are the one needed to finish the execution of the program (in other words, task joint).
- Tasks do not have a fixed size. Even though each task has the same amount of code, the work per task is not. A reason can be found in the recurrence of the Mandelbrot set: $Z_{n+1} = Z_n^2 + c$ has quadratic growth, so for some ns it will be bigger than for others (yet the value of n is capped to prevent the generation of a huge set).

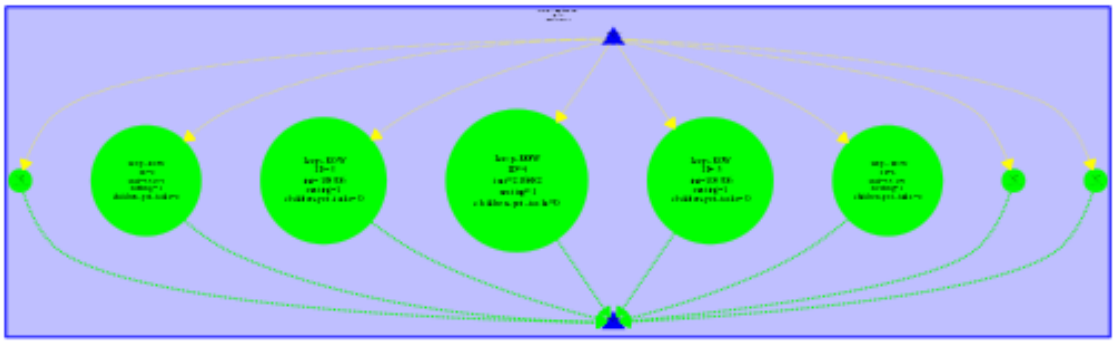


Figure 1: Task Dependency Graph for Row-based parallelization for the program that prints the output on the console.

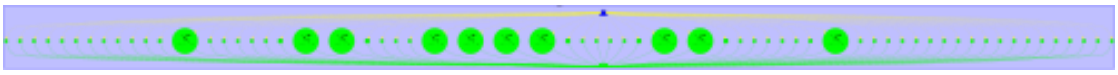


Figure 2: Task Dependency Graph for Point-based parallelization for the program that prints the output on the console.

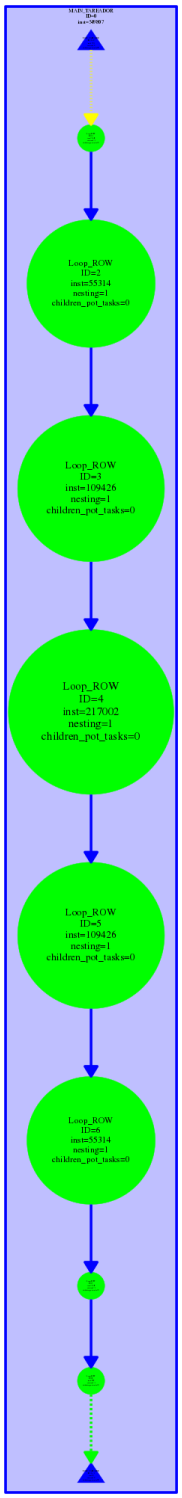


Figure 3: Task Dependency Graph for Row-based parallelization using OpenGL to render the Mandelbrot set.

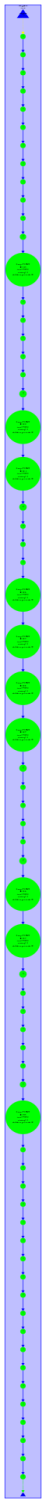


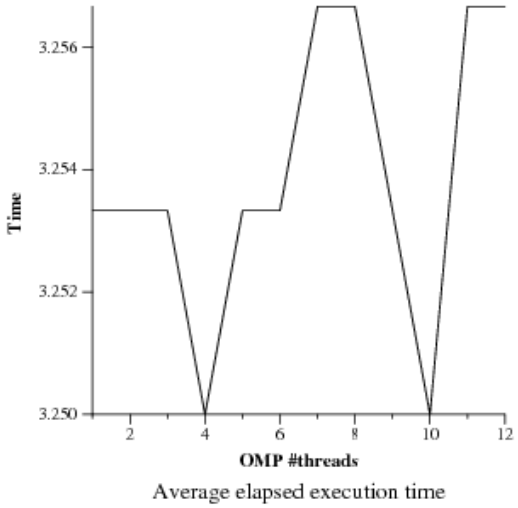
Figure 4: Task Dependency Graph for Point-based parallelization using OpenGL to render the Mandelbrot set.

4.2 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?

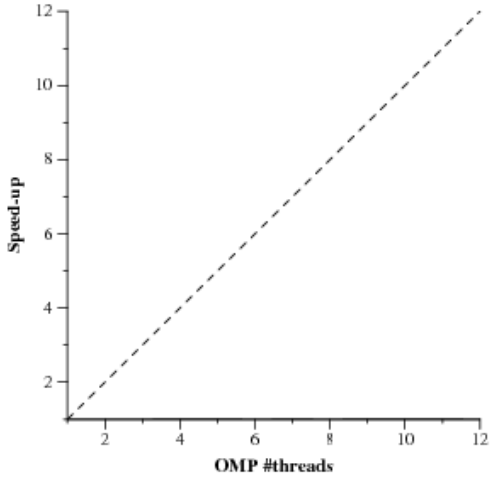
The code that is causing the serialization of the `mandeld-tareador` tasks is the function `XDrawPoint(...)`. We plan to protect this section of code using the `#pragma omp critical` directive before said function.

5 OpenMP task-based parallelization

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

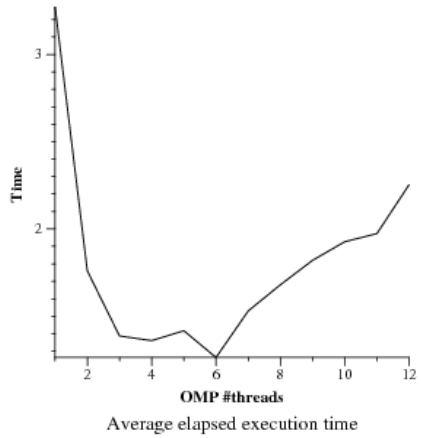


Average elapsed execution time

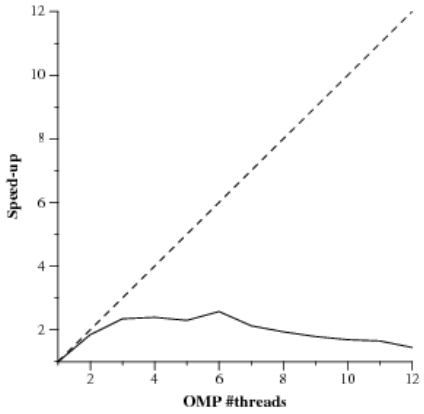


Speed-up wrt sequential time

Figure 5: Speed-up and average elapsed time for Row-based parallelization using the task directive.



Average elapsed execution time



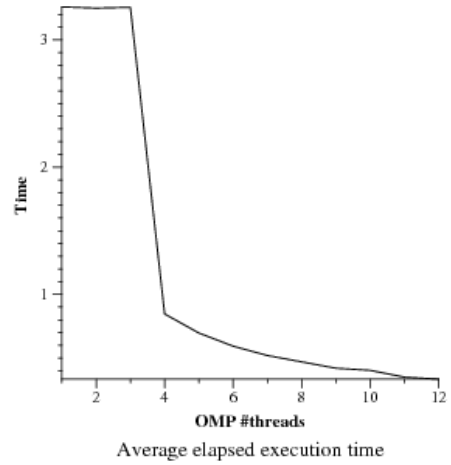
Speed-up wrt sequential time

Figure 6: Speed-up and average elapsed time for Point-based parallelization using the task directive.

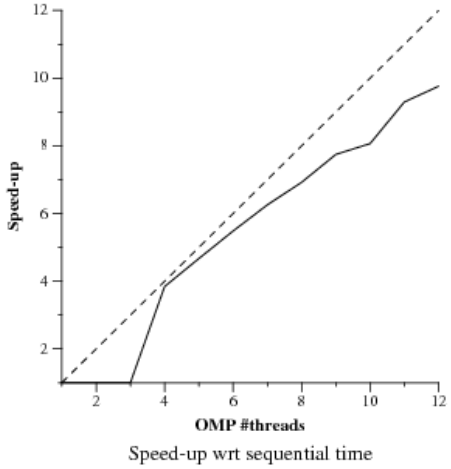
Row and Point run erratically depending on the number of cores: the times in Row are quite big, unless there are 4 or 10 cores in use (which run in greatly lower times), and Point has a logarithmic decay in times, but using 6 or more threads causes the execution time to increase greatly! For Point we suspect that the use of the critical for the display version generates these issues.

6 OpenMP taskloop-based parallelization

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

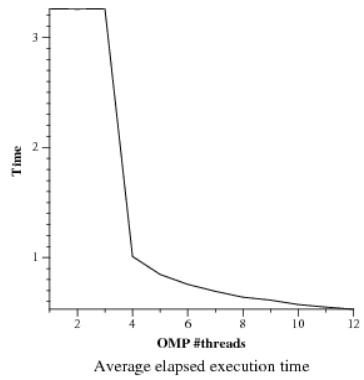


Average elapsed execution time



Speed-up wrt sequential time

Figure 7: Speed-up and average elapsed time for Row-based parallelization using the taskloop directive.



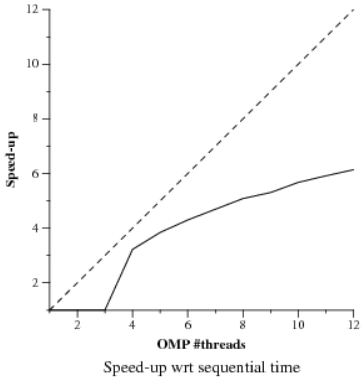


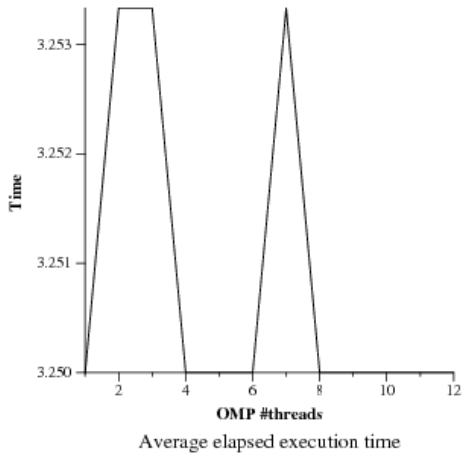
Figure 8: Speed-up and average elapsed time for Point-based parallelization using the taskloop directive.

- **Row decomposition Performance analysis:** It's close to being perfectly(?) parallel, the reason it's not is due to the overhead of the N.
- **Point decomposition Performance analysis:** While it improves compared to the serialized code, the end result is that the Point decomposition is not as fast as the Row one by almost 3 and a half times.

Both strategies are quite similar: using 1 to 4 threads is quite expensive, but using more than 4 the performance of the program increases greatly. The only difference is in speed-up: Point has to sync each thread because there are lots of tasks, while Row doesn't have to because of the fewer number of tasks.

7 OpenMP for-based parallelization

7.1 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.



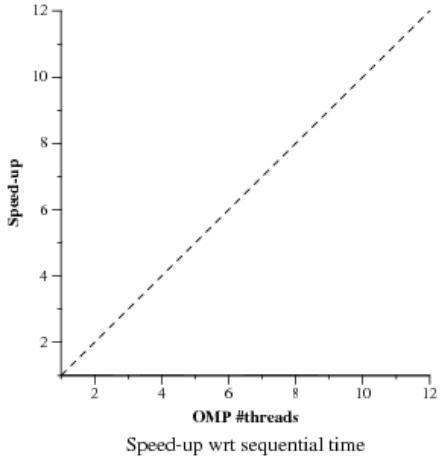
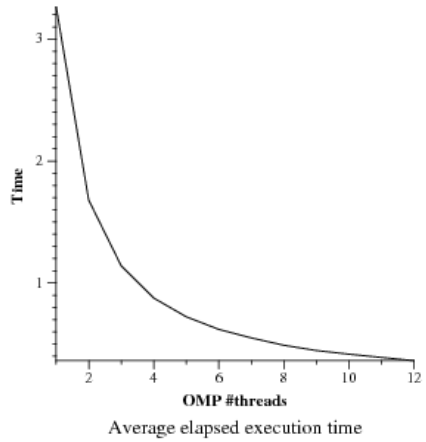


Figure 9: Speed-up and average elapsed time for Row-based parallelization using the for directive.



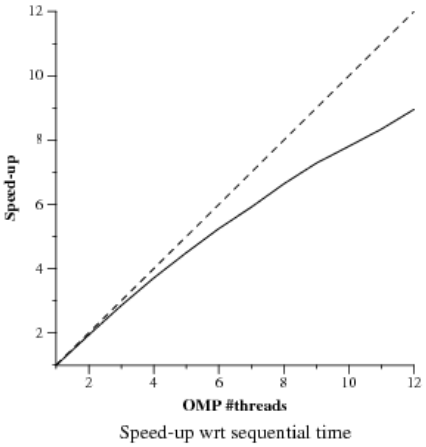


Figure 10: Speed-up and average elapsed time for Point-based parallelization using the for directive.

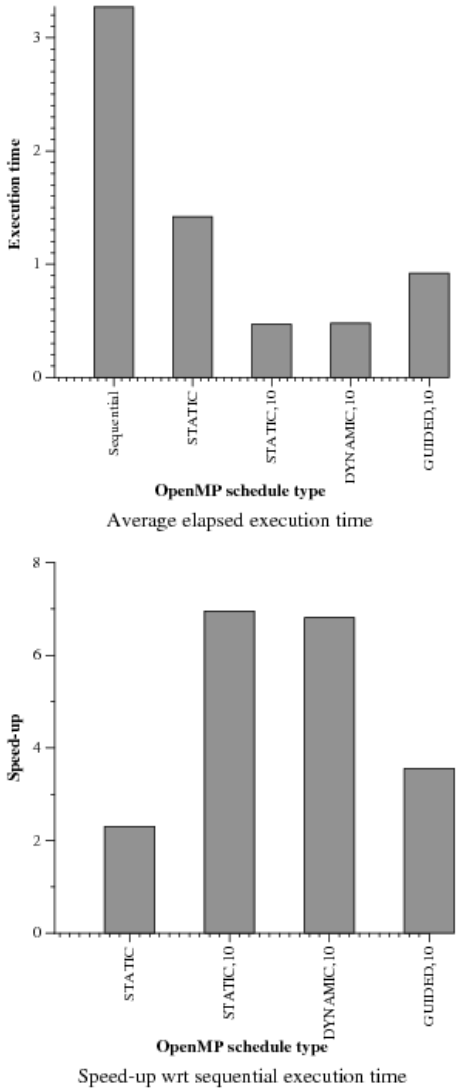


Figure 11: Graph showing the scheduling performance for Row-based parallelization.

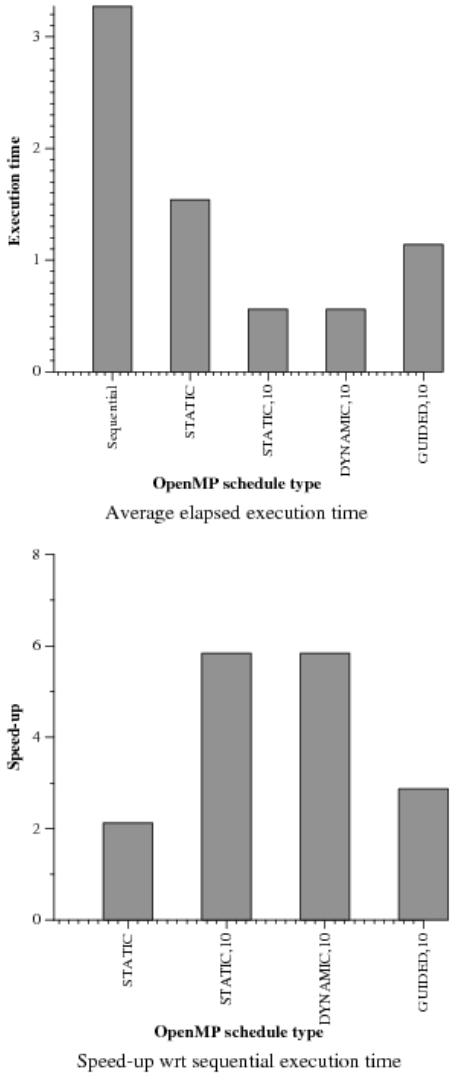


Figure 12: Graph showing the scheduling performance for Point-based parallelization.

- **Row decomposition Performance analysis:** There are strange spikes when using few cores (with the exception of four to six cores, and eight to infinity), which means that this decomposition is slow using few cores due to the size per task. That said, when using 8 or more cores, the execution of the program is quite fast.
- **Point decomposition Performance analysis:** Unlike the previous decomposition, this one has a strictly logarithmic decay in time execution, and

an close-to-linear speedup increase.

- **Scheduling:** There is not too much of a difference in scheduling of bot Row-based and Point-bases tasks, because each strategy does not force synchronization (due to the problem being embarrassingly parallel, synchronization is not an issue) for any of the scheduling types.

7.2 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 avg time/thread	1,780.18	462,057	440,858	350.578
Execution unbalance	0.78	0.89	0.97	0.392
SchedForkJoin	1,483.63	30.461	3.735	6.577

(All times displayed in microseconds)

8 Conclusions

As we have seen, parallelizing a program can have a significant improvement on the performance of embarrassingly parallel applications. Also, there are not too many modifications to the code to produce this versions: in fact, we have seen that there are several ways to parallelize this code and each approach may give different results, in time and scalability of the problem, which has its pros and cons.