Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

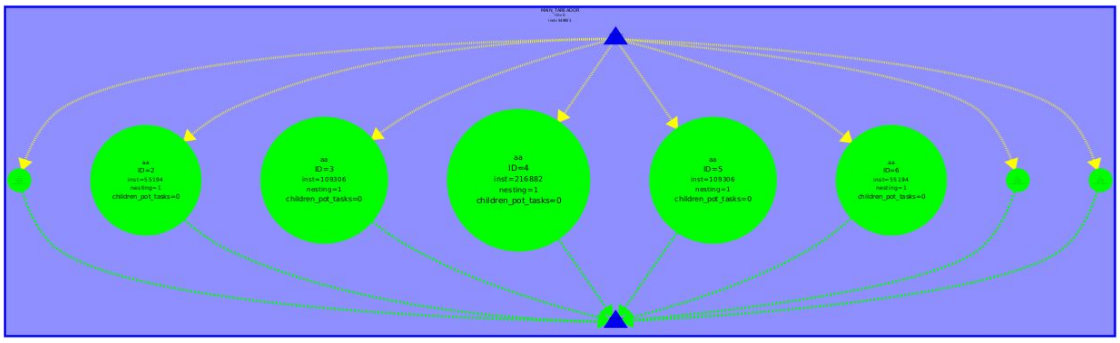
Pablo Cebollada Hernández
Iván Díaz Ortega
Par1104
Q1 2019/2020
7/11/19

INTRODUCTION

In this assignment we are going to look at the possible parallelism of the code for the calculation of the Mandelbrot set. We will take different approaches, since, first, we are going to try a point decomposition, test different granularities for the task decomposition and analyse the results that we get. Afterwards we are going to do the same with a row decomposition.

4.1 Task decomposition and granularity analysis

When we compile and create the two graphics about the two different granularities, per row, creating a task every time the code changes the variable Row. This makes a small number of nodes. However, some nodes have a lot of iterations and that creates a significant amount of unbalance between the different tasks. Per Point, creating a task every time the code makes an iteration in the for Col, creates many task but with a small number of iterations. We can see many more tasks than with the row decomposition but with much less iterations per node than the first decomposition.



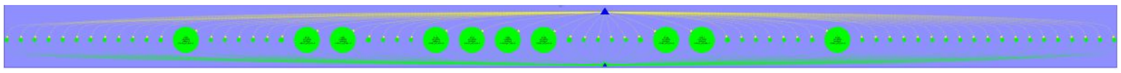
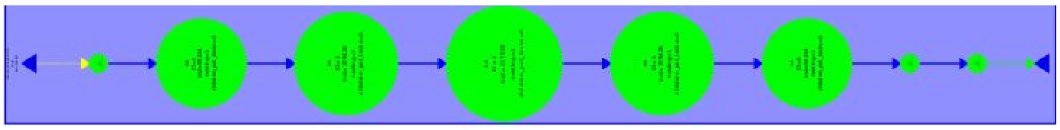
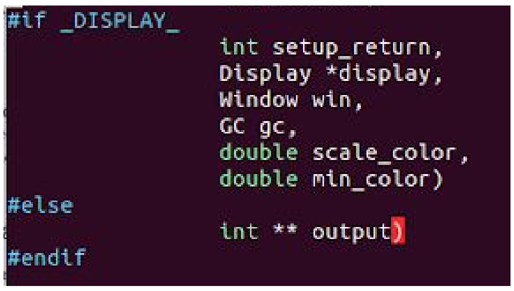


Image (up) Dependencies per Row Image (down) Dependencies per Point



Dependencies when mandeld_tar is executed

If we look at the dependency graph of the execution of the program with the graphical interface we can observe that the dependencies don't allow us to parallelize anything. After observing the code, we have concluded that the section which causes the serialization when you execute ./mandeld-tar is the part of the code that generate the display (Image 4), that could be because the program prints every pixel one by one and following an order, thus creating a dependency between the different tasks.



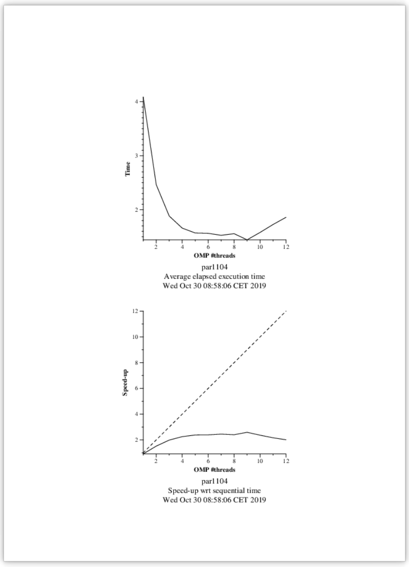
```
#if _DISPLAY_
                int setup_return,
                Display *display,
                Window win,
                GC gc,
                double scale_color,
                double min_color)
#else
                int ** output)
#endif
```

Display code

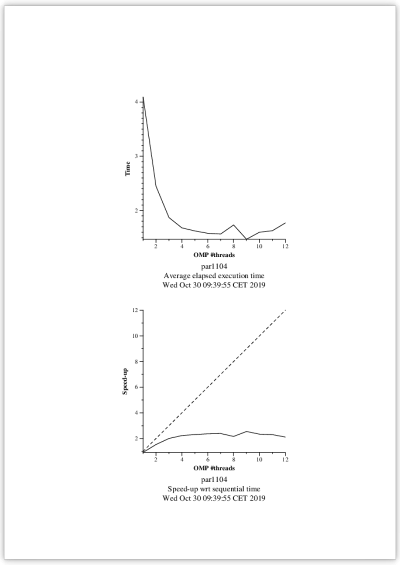
Now, looking at the different strategies and the different granularities that they imply we can say that the row strategy, though having a lower granularity than the point, it would be more helpful when having a lower amount of threads, since, in this case, having a lower granularity would be better for controlling overheads. On the other hand, if we have a larger amount of threads, a higher granularity would be better, since the potential parallelism is greater and the overhead wouldn’t be a problem. However, if the amount of tasks generated highly surpasses the amount of available threads, the granularity should be lower, since the overhead problem appears again, thus a row strategy could be, again, a better option.

4.2 Point decomposition in OpenMP

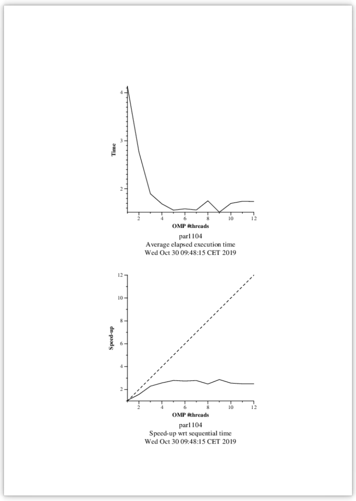
We tried three different versions of the point decomposition strategy: one without any synchronization mechanism, other using taskwait or taskgroup and a third one that uses the taskloop clause. Focusing first on the first strategy that we analysed and looking at the execution time and speed-up we can see how fast we arrive at the limit of the possible parallelisation since, as we can see in the graphic for the execution time and speed-up without synchronization mechanism, the execution time stabilizes at around five threads, just as the speed-up does. This might be due to the great amount of overhead and also because of the strategy used in this case, since there are a few threads that create tasks, not only one. The strange valley that’s in the 9 threads line might be due to the CPU turboing up the frequency and going faster for a small period of time, causing that strange behaviour. If we now look at the performance that we get by allowing some kind of synchronization mechanism we can see, by looking at the different plots, the one with taskwait and the other with taskgroup, we observe that the potential gain is even lower, since, by limiting the threads that create tasks to wait for their children to finish to create more tasks. Although with taskgroup a higher speed-up can be observed. This can be due to a lower amount of overhead, however, the performance is quite the same, since the lower execution time is got at around the same amount of threads, 5.



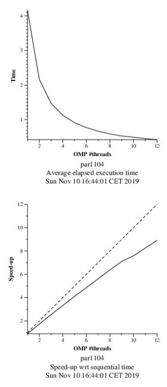
Execution time and speed-up without synchronisation mechanism



Execution time and speed-up with taskwait

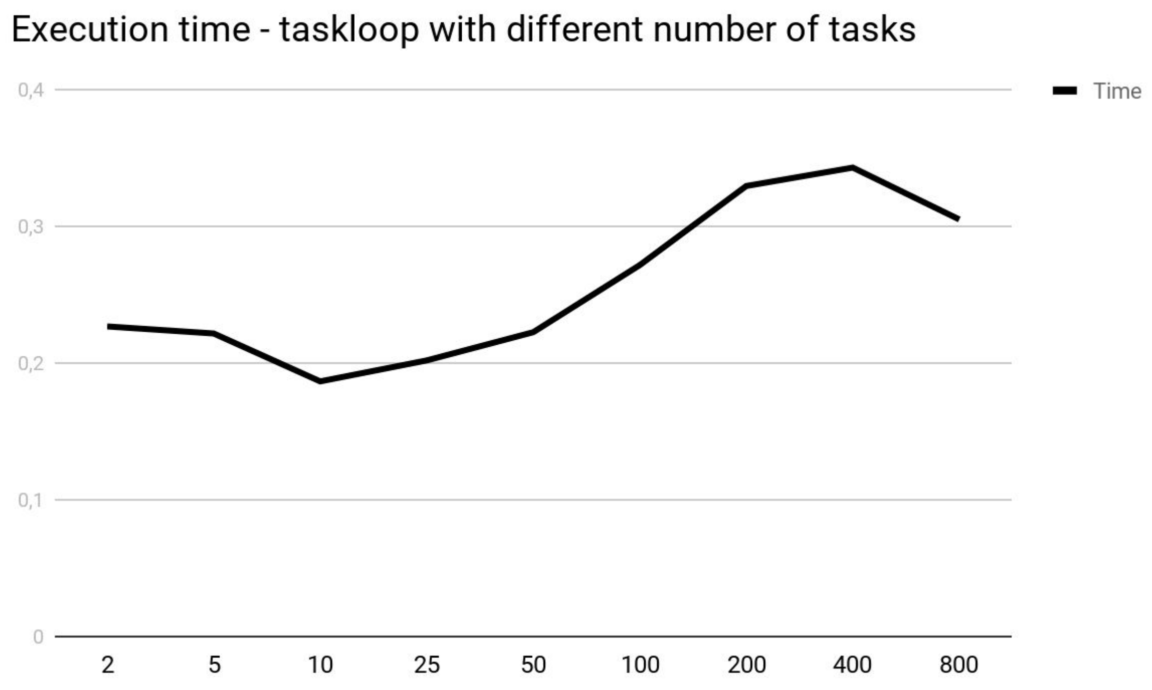


Execution time and speed-up with taskgroup



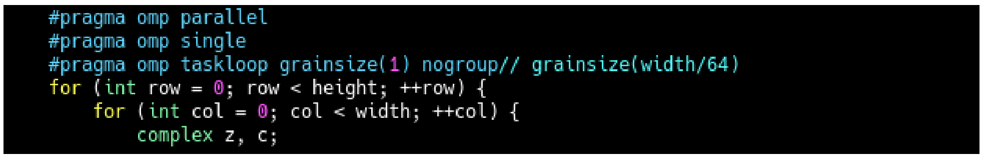
Execution time and speed-up with taskloop

Focusing now on the taskloop construct and the impact the granularity has on the performance, and looking at the graphic we have here that shows us the execution time for the different number of tasks created, we can clearly see how this number impacts on the execution time. If we start analysing the graph from the left, lower number of tasks, we observe that with a small number of tasks the execution time is lower than with a larger amount. This is due to, again, the overhead created by the creation of a great amount of tasks. Since the number of threads is the same across all these tests the overhead will be greater as we increase the number of tasks as synchronization work has to be done more often.



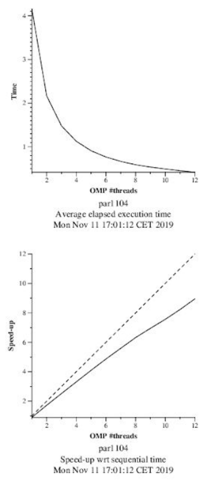
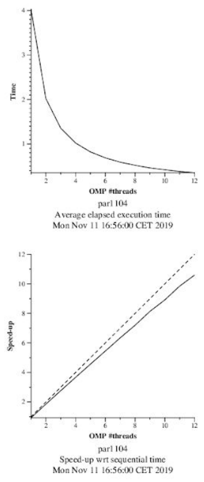
4.3 Row decomposition in OpenMP

To successfully achieve the Row strategy we implemented a taskloop strategy with the parameters of grainsize(1) and nogroup. We chose taskloop against a simple task because taskloop is much better in overhead time, this factor and the massive number of iterations in the two loops makes this one the better option. The reason why we chose grainsize(1) is because we wanted to observe the performance of the taskloop clause with the same granularity as the other clauses, although we could change grainsize(1) for grainsize(width/64) and obtain the same result. This way we can compare the difference in overhead that can be introduced or reduced by using this clause and not another one. The last parameter nogroup eliminates the taskgroup associated with the construct of taskloop.



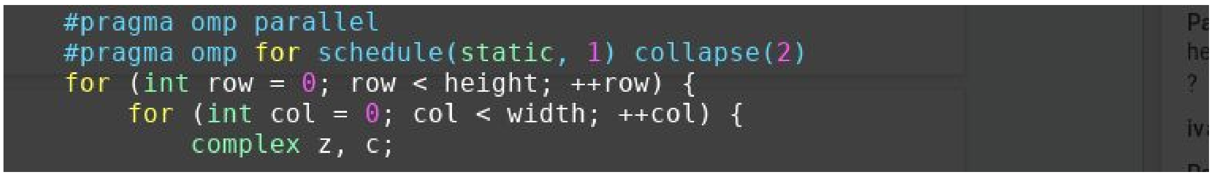
```
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop grainsize(1) nogroup// grainsize(width/64)
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            complex z, c;
```

As you can see in the graphics, left, granularity Point and right, granularity Row, the time graphic may seem that they are the same, but granularity Row is a little bit faster in the interval [2, 4] than granularity Point, in terms of speed up, granularity Row wins against granularity Point because the penalty for overhead in Point is much bigger than in Row, making Row the best option to choose.

	
Granularity Point	Granularity Row

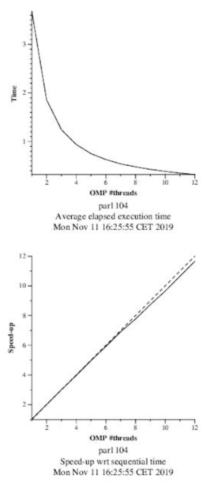
4.4. for-based parallelisation

To implement the code parallelisation with the for clause we need to add it to the code the way it is done in the picture below.

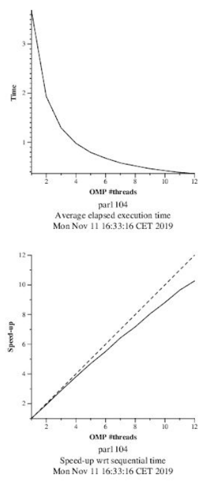


```
    #pragma omp parallel
    #pragma omp for schedule(static, 1) collapse(2)
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            complex z, c;
```

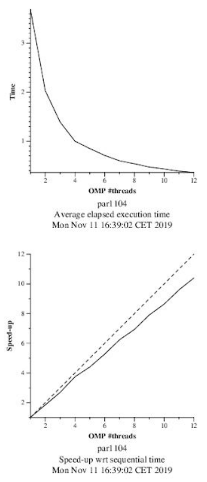
Here we decided to use the collapse() clause because, this way, we would have a serious advantage deciding the chunk size, since we are not limited by the size of a row but by the size of the entire matrix. First let's take a look at the static schedule policy. Using the minimum chunk size we can see how the spee-up line follows almost a linear pattern, meaning that the overhead added by this clause is very low and that using this policy with this granularity scales very well with the amount of threads. The other two policies behave very similar, since using a chunk size of 1 with guided is the same that using dynamic. But comparing them with the static policy a worse over-head line is observable with these two, since using these policies causes more overhead due to that they assign each task dynamically during execution.



for - static schedule
chunk size 1

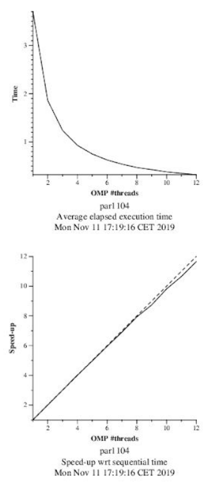
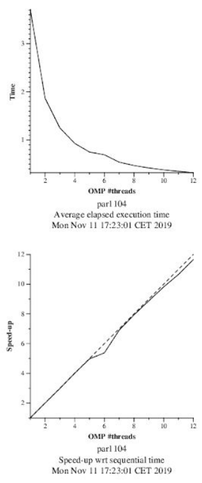
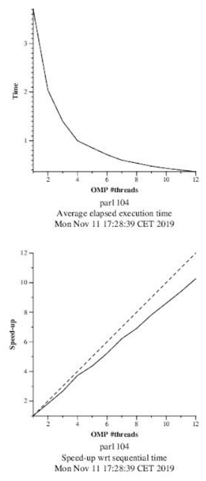


for - dynamic schedule
chunk size 1



for - guided schedule
chunk size 1

Now, if we take a look at the same policies using a bigger chunk, we see that the speed-up is almost perfect, meaning that the overhead generated is barely noticeable, and that this program scales really well with the number of threads. Although the guided policy is the one that gains the least profit out of increasing the chunk size, since the guided policy reduces it through the execution.

		
for - static schedule chunk size width	for - dynamic schedule chunk size width	for - guided schedule chunk size width