PAR Laboratory Assignment

Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Albert Bernal (2205) and Sergi Bertran (2206)

12/11/2020

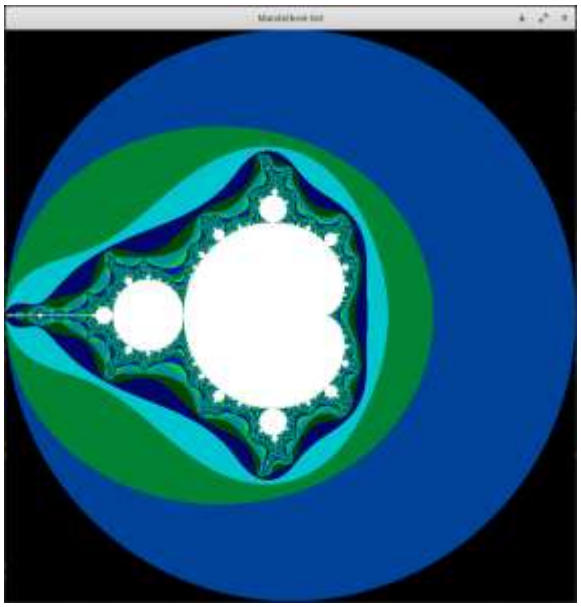
Introduction

Parallel computing is, most of the time, the best way of speeding up the code we write. Computer hardware can't evolve as fast as we want, and therefore taking advantage of using multiple computing units to solve a problem is where we can find its solution.

First, let's define our case of study. The Mandelbrot set is a very famous mathematical object defined by a simple rule:

$$z_{n+1} = {z_n}^2 + c$$

By giving a different colour to each point of the set regarding the iterations it took to compute, we can obtain this interesting image:



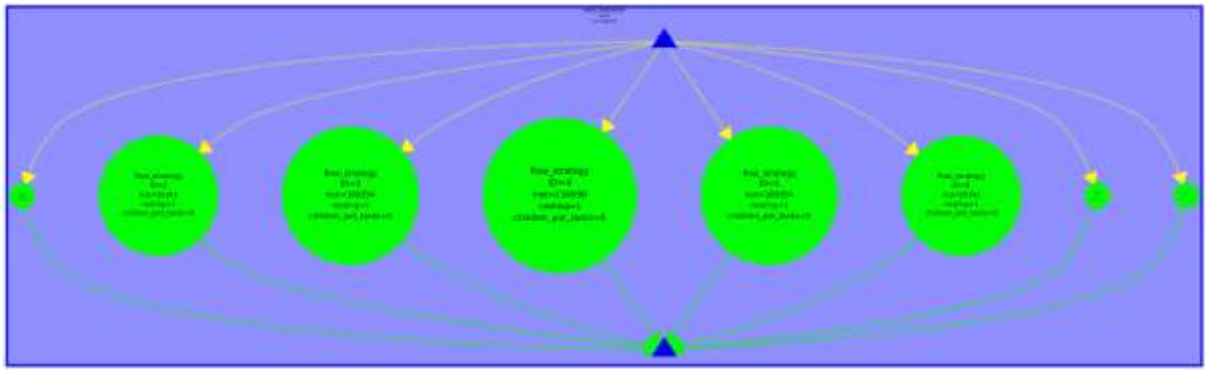
In this laboratory session we will be understanding, improving and evaluating different methods of defining and dividing tasks within our programs by taking a computing approach to the Mandelbrot set calculus.

4.1 Task decomposition and granularity analysis

The aim of this section is to test different parallelisation strategies and see which obtains the best outcome. Therefore, the first step we took was to analyse the code so as to understand what it does. The two main strategies that are going to be used are point and row decomposition.

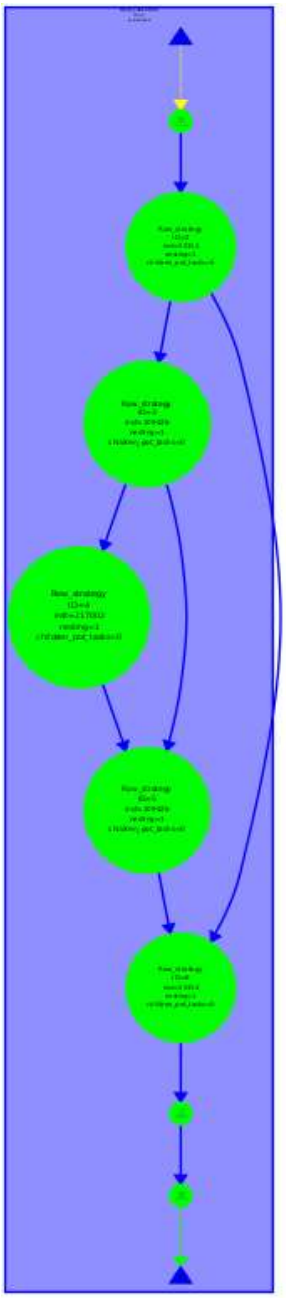
We executed row first, through Tareador and without any additional parameters.

As can be observed in the image below, there are no task dependencies generated between the tasks that compute the Mandelbrot set. In this case, since every task can be executed in parallel, we can refer to it as an embarrassingly parallel program. Its load balance is not the desired, though, some threads will end their work far before others, limiting the efficient use of resources.

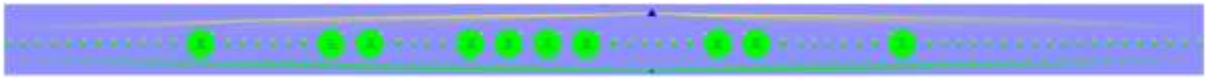


We then run the binary interactively with the additional parameter *-d*. This option is used to display the Mandelbrot set. Using this parameter we observe that the program runs completely sequentially. No part of the program can be parallelized.

Finally, we ran the binary interactively with the additional parameter *-h*, which is used to obtain the references of the sequential execution. In this graph we see that there are some more dependencies between tasks. However, it shows us that the dependencies that exist between tasks are not effective and do not allow code parallelization.

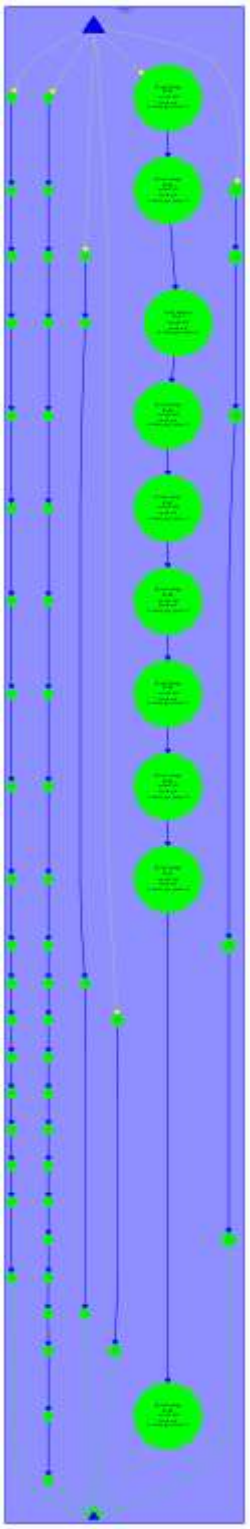


As for the point decomposition strategy, when we execute the binary without any additional parameters with Tareador we observe that a graph is shown again where we see the program is embarrassingly parallel. All tasks can be run in parallel, although this time there is a much higher number of tasks. However, we also note that there are still a number of tasks of considerable weight, and the load is unevenly balanced between the threads.



Next we executed the binary with the point decomposition strategy interactively with the additional parameter *-d*. In this case we observe that the program is executed again completely sequentially with the only difference that the number of tasks has increased and that consequently the execution time due to the overheads will be higher.

Finally, we interactively execute the binary with the additional parameter *-h*. In this task dependency graph obtained through Tareador we observe that several tasks have been parallelized. However, the improvement is vague, since the bigger tasks are completely sequential, with no possibility of parallelizing them due to dependencies.

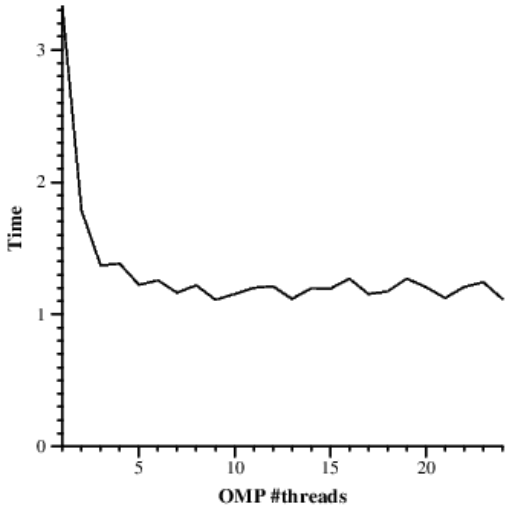


The best strategy to use would be column decomposition, because it is well balanced and with a number of reasonable tasks. In contrast, the points strategy creates many tasks, which results in a much higher overhead that limits the scalability of the program.

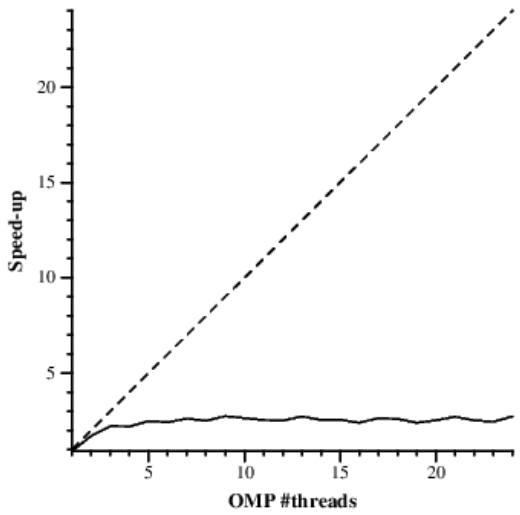
4.2 Point decomposition in OpenMP

In this section we aim to describe, understand and evaluate the performance obtained by using different iterations of the same point decomposition strategy.

The first implementation was given in the original model-omp.c file, which we compiled and executed after reviewing its content. These are the elapsed time and speedup charts obtained by



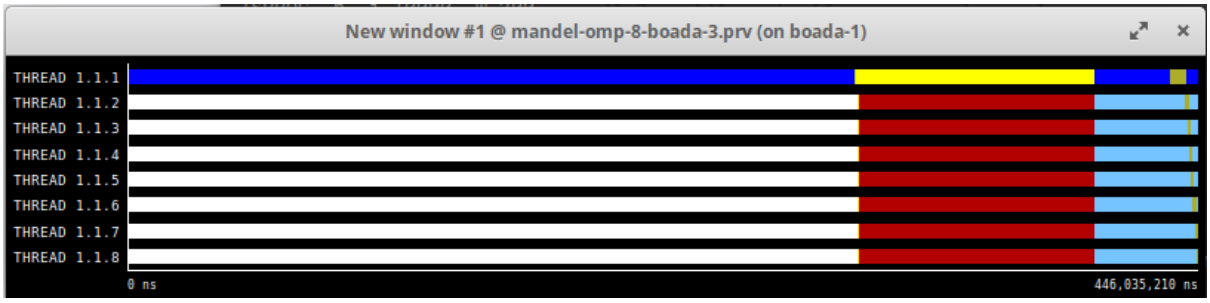
par2205
Average elapsed execution time
Wed Nov 11 19:39:06 CET 2020



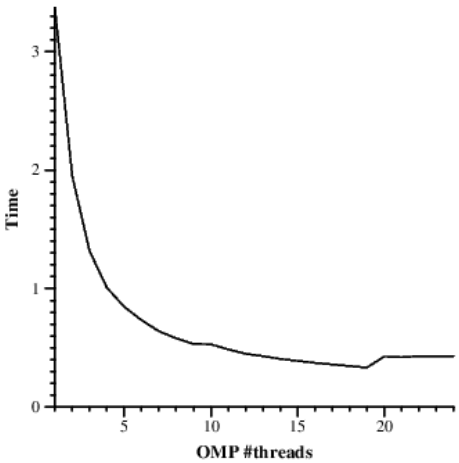
par2205
Speed-up wrt sequential time
Wed Nov 11 19:39:06 CET 2020

submitting the executable to boada:

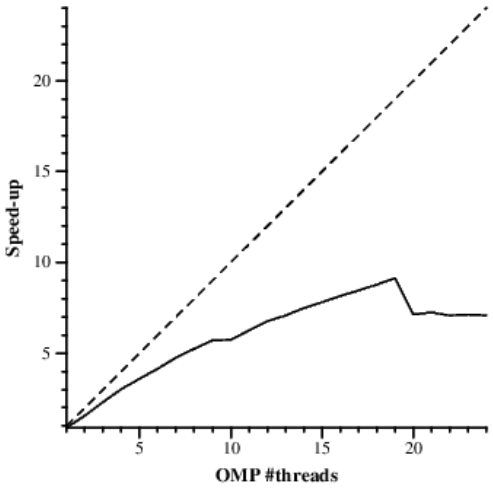
As can be observed, its performance is not even close to desirable, with a non-increasing speedup, a non-decreasing elapsed time and, consequently, a deceiving scalability. This is expected, since the amount of tasks created and the overhead attached to it will prevent us from achieving any kind of progress by adding threads. To back this reasoning, we attach the task distribution obtained through Paraver, in which we can see only the first thread executing code and the others waiting for it.



Afterwards, we are asked to reimplement model-omp using taskloop. Our approach was to apply this taskloop directive to the inner loop in the mandelbrot function. We introduced "#pragma omp taskloop firstprivate(row) grainsize(1)" between the two loops (a file called taskloop1.c is attached in the deliverable) and recompiled. These are the charts obtained:



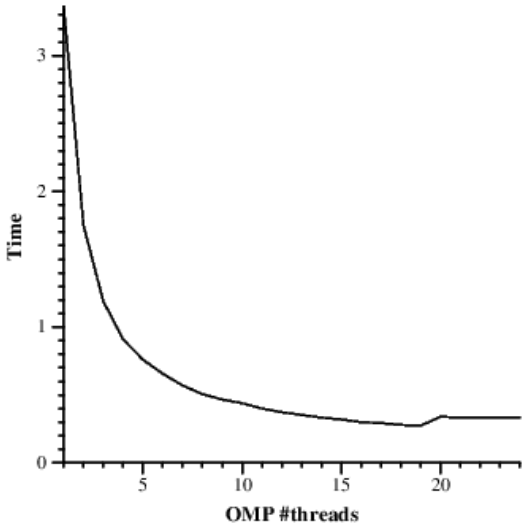
par2205
Average elapsed execution time
Wed Nov 11 19:01:20 CET 2020

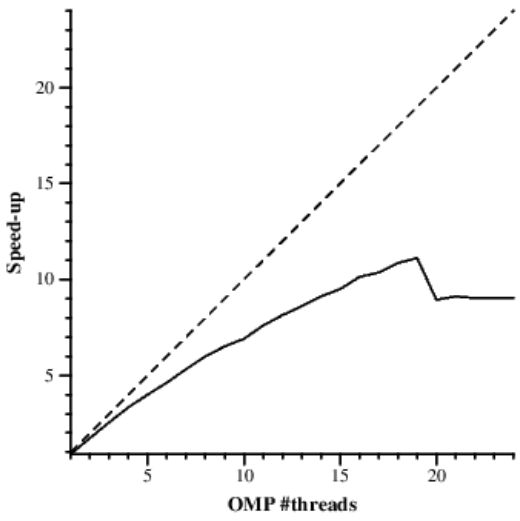


par2205
Speed-up wrt sequential time
Wed Nov 11 19:01:20 CET 2020

We expected a better result due to taskloop doing a much better job creating and assigning tasks, and it was indeed better. The elapsed time reaches its threshold at a much lower value, nearly halving the one of the previous execution. Speedup, although presenting a strange deviation around the 18 threads mark, is much more promising, and is far closer to the ideal 1:1 ratio the dashed line represents. We think this strategy can be considered a valid option, since the program is strongly scalable.

The last optimization suggested was to add the nogroup clause to the taskloop, which removes the implicit taskgroup region that encloses the taskloop construct. By adding it, the implicit barrier from the taskgroup disappears with it, and threads spend less time waiting. These are the last charts:



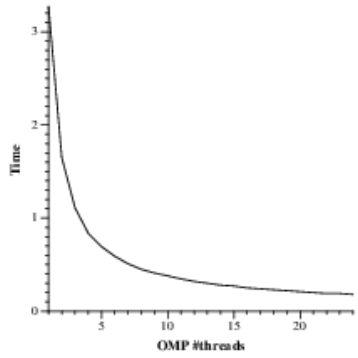


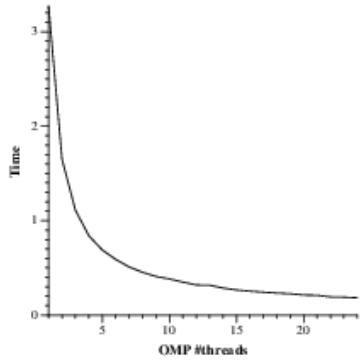
A minor improvement, but an improvement nonetheless, is obtained thanks to the nogroup clause. We can easily compare the speedup's slope to the previous one by comparing its maximum value, which was around 9x, to this one's, around 11x. This also helps improve the scalability we discussed earlier, and we conclude this parallelisation strategy is viable.

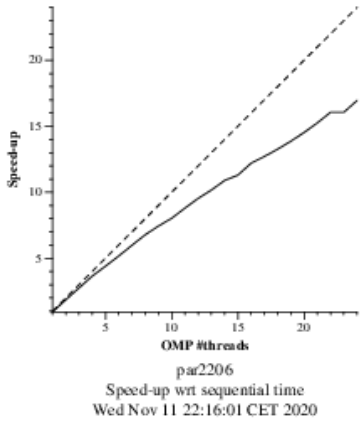
4.3 Row decomposition in OpenMP

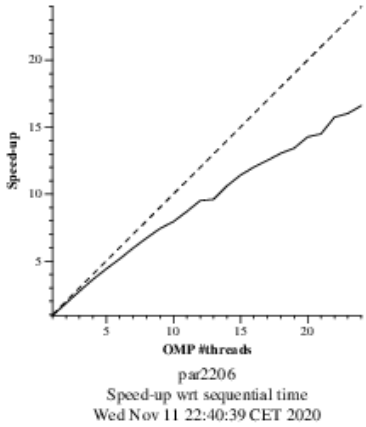
The other approach suggested was the analysis based on row decomposition. To do so, the change we applied in the code was moving the task definition one line above its initial position, affecting the loop instead of each iteration, as it can be seen in the files attached.

These are the charts obtained with both taskloop versions of the row decomposition strategy:









We cannot appreciate much differences between the task and taskloop versions (task being the one on the left, taskloop on the right). Both of them have similar performances although the performance of the task version is a bit better. However, we can conclude observing these plots is that there is an important improvement in the performance and efficiency compared to the point strategy. Moreover, it has a satisfactory scalability. The more threads the more speed-up it has nearing the ideal scalability which would be that per each processor we add the speed-up increase by one unit as we can see in the plot. Therefore, we think this is the best strategy and the one that should be implemented.