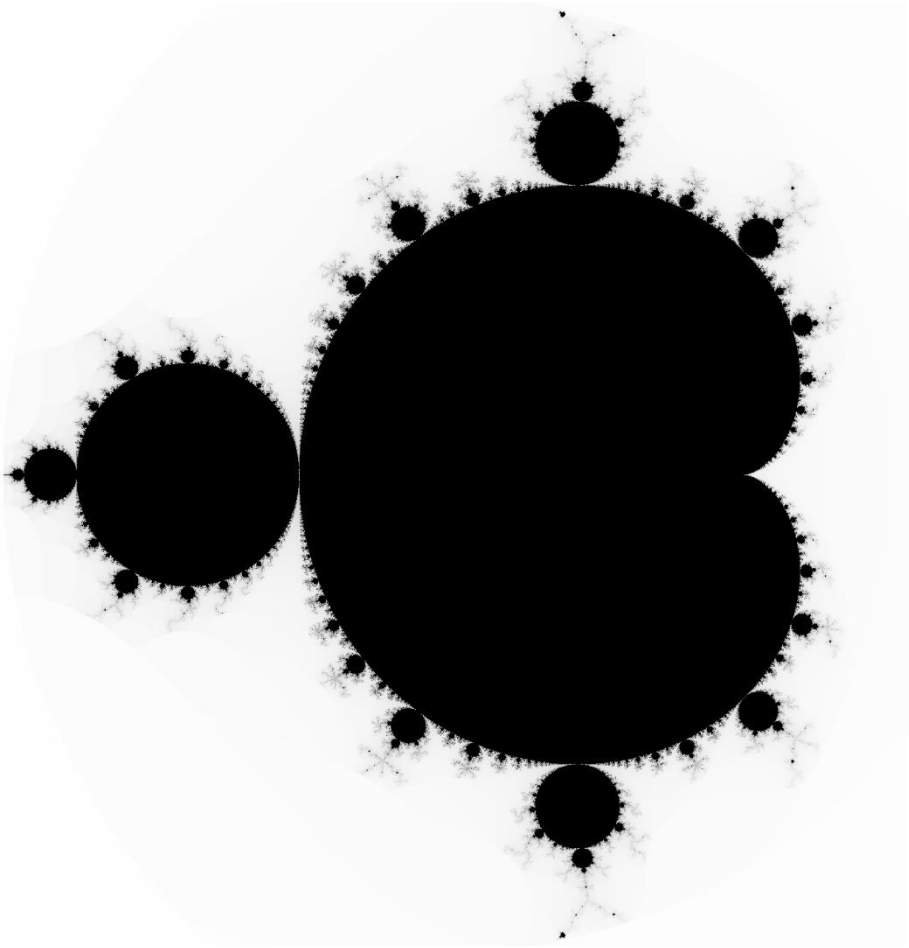
Third Assignment Report

Exercise 1

Computation of the Mandelbrot set

The assignment was to compute the Mandelbrot set, represented in the figure below.

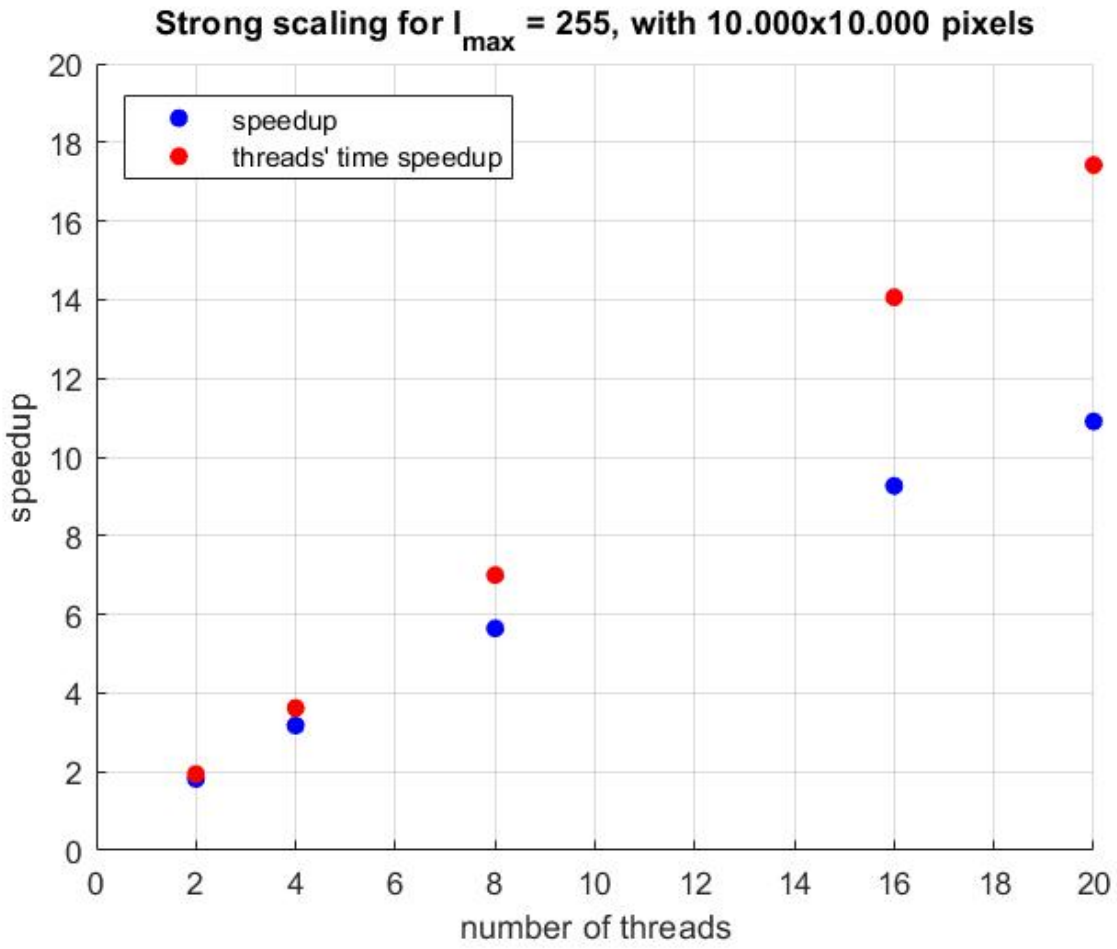


All tests were conducted by considering the square region with vertices (-1.5, -1.5),

(-1.5, 1.5), (1.5, 1.5), (1.5, -1.5), in order to compute the whole Mandelbrot set. In order to achieve a balanced workload a dynamic scheduling was necessary for the loop. Moreover, on Ulysses, the standard setting is OMP_PROC_BIND=FALSE and OMP_PLACES='', which in

some cases resulted in the system migrating threads, therefore causing degraded performances. Hence it was necessary to set OMP_PLACES=cores to avoid such problems. Finally, the tests were done on the normal version of the exercise, not the one compiled with the -DBALANCED flag, which for I_max=255 behaves poorly and for I_max=65535 behaves slightly worse than the normal one.

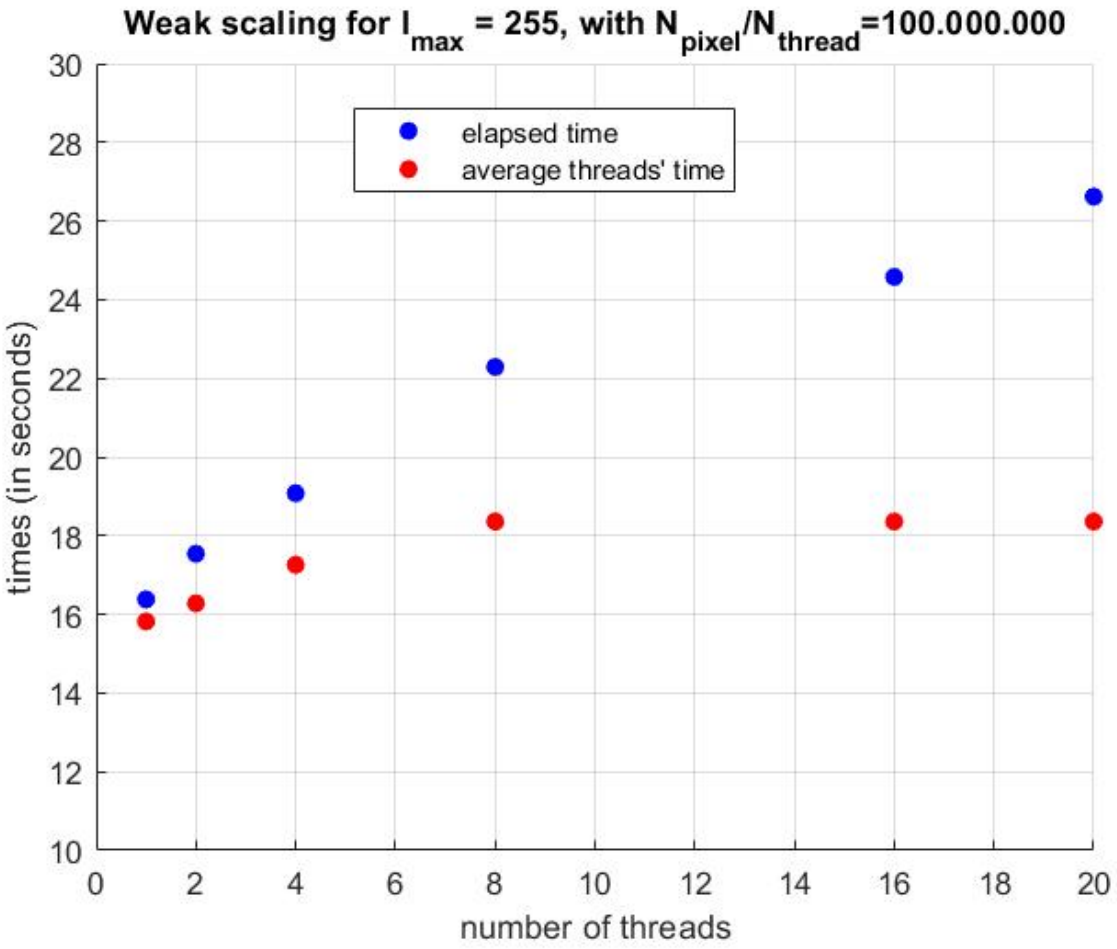
In the picture below we can see a strong scaling test, done with a problem size of 10.000x10.000 pixels and a maximum of 255 iterations per pixels.



Blue dots represent the speedup calculated on the elapsed times, whereas the red ones were done considering only the average walltime of the threads. As a result the whole algorithm doesn't scale too well, probably due to the fact that the creation of the image is not done in parallel. On the other hand the threads time, being the problem embarrassingly parallel, should have linear speedup. This doesn't exactly happen, even though the speedup is still in the worst case 85% of the theoretical one, due to the parallel overhead caused by

the dynamic scheduling of the OpenMP for loop. Such scheduling is needed in order to ensure a balanced load distribution among the threads, as with a static scheduling the results were horrendous, with some threads taking up to 40 seconds and other taking 0.002 seconds in my test. Guided scheduling, albeit better than the static one, still delivered worse performance than dynamic scheduling.

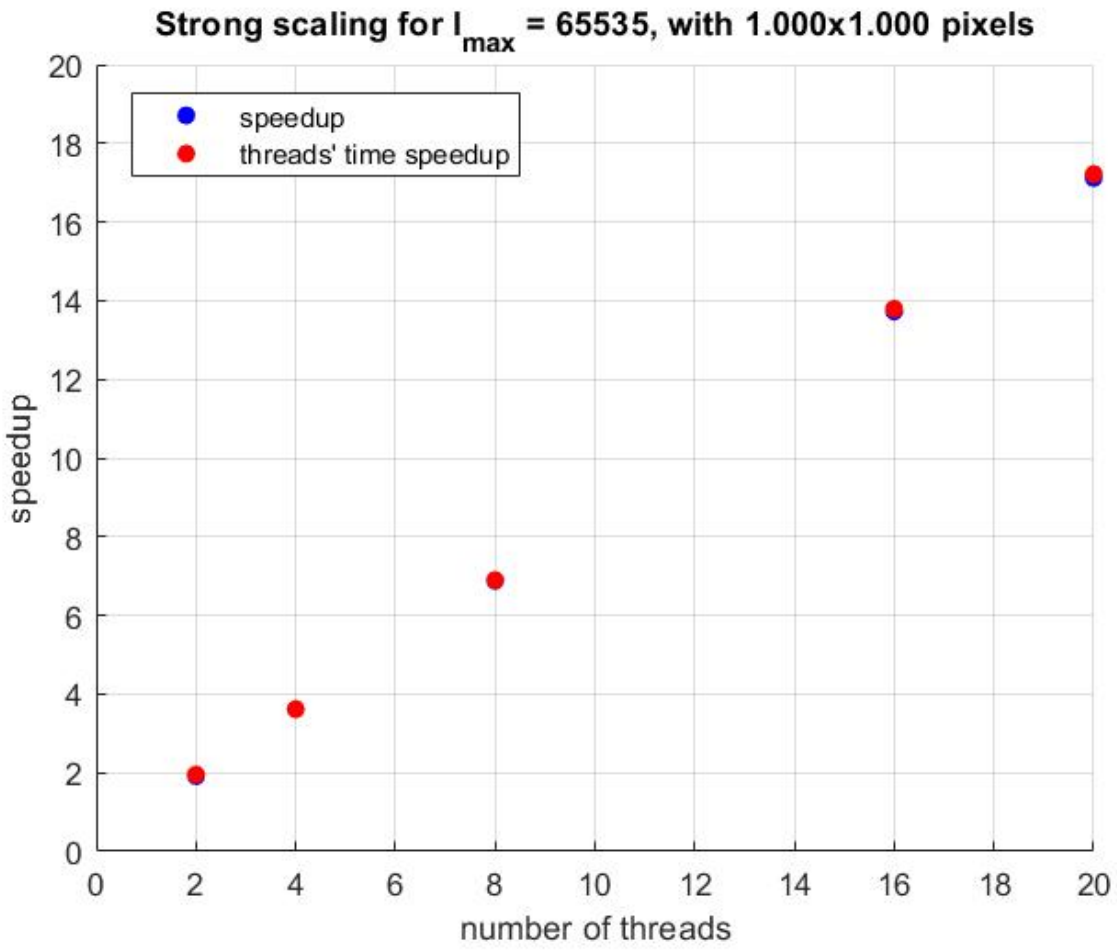
In the figure below we can see a weak scaling test, done by fixing a problem size of 100.000.000 pixels per thread and a maximum of 255 iterations per pixels.



For the algorithm to weakly scale well the execution times should be more or less the same, which clearly is not the case with this application. The threads' times scale decently, whereas the elapsed time don't really scale. This is definitely due to the fact that by increasing the problem size also the serial part increases, and in particular writing to the file requires more time. Of course by increasing the number of pixels also the overhead increases, as the

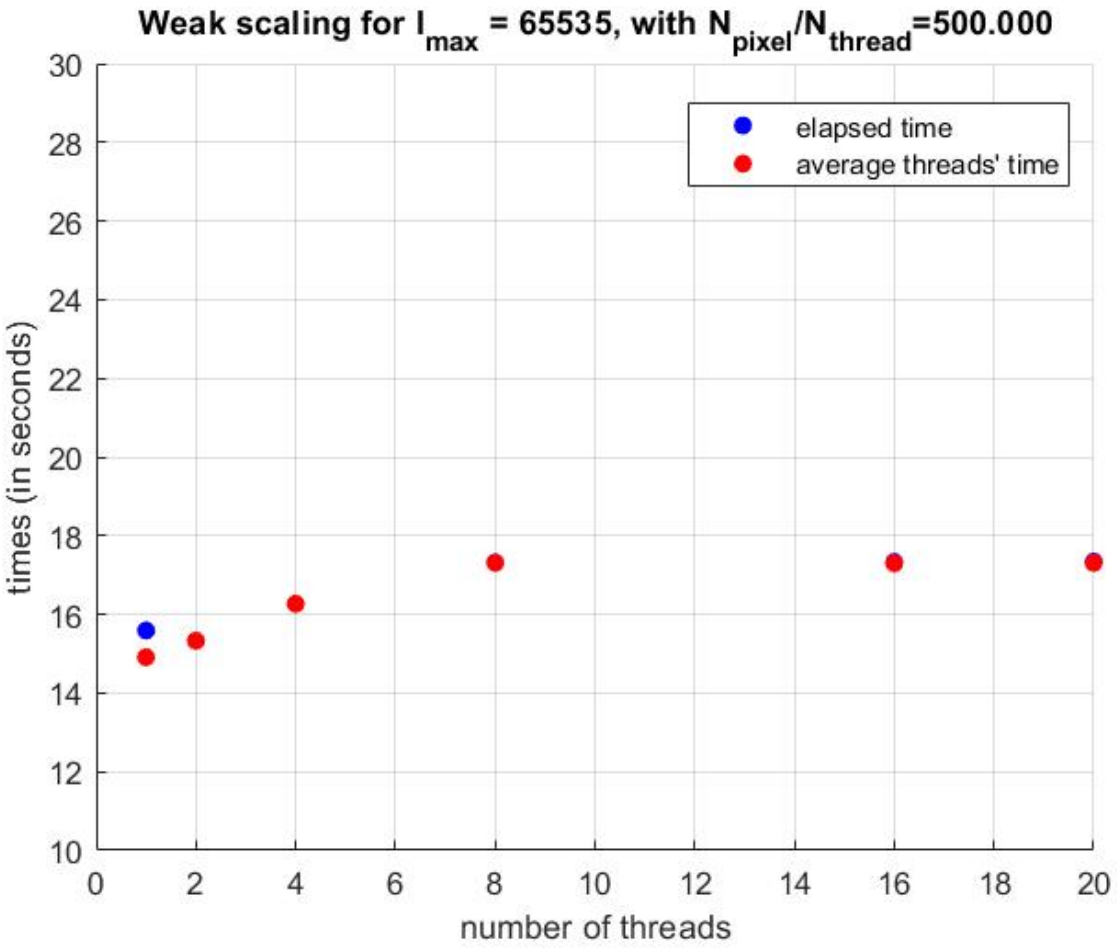
OpenMP scheduler has more work to do, and that contributes as well to the bad scaling of the code.

The next tests were done with 65535 as the maximum number of iterations. The algorithm in general scaled better than before, definitely due to the computation being the main part of the execution.



In this case the speedup of the whole algorithm is basically equal to the one of the threads.

In the next picture a weak scaling test is plotted. The scaling for the elapsed times again equals the one for the threads' time, but it still is not perfect. This is due to, as before, the increase in writing time and the overhead of the scheduler.



Hybrid code

Concerning the hybrid MPI+OpenMP code, first of all when running it the flag –bind-to none must be specified, otherwise the MPI processes will be bound to a single core each and the threads will spawn on such core, making OpenMP useless.

Concerning performance, when changing the number of threads and using the same argument for the -np flag the speedup is slightly worse than before. This is due to the overhead caused by the MPI machinery, which as seen during assignment 1 amounts to approximately 1.5 seconds. Moreover there is also the extra communication times between processes that degrade performance. It is interesting in fact to observe how, while keeping fixed the product (number of processes used)*(number of threads), the performance changes. In the test case, up to 20 processes where started, with a corresponding 20/(number of processes) threads per each process. The times are reported in the following

table. The test was done on a grid of 1000x1000 pixels, with vertices (-1.5, -1.5), (-1.5, 1.5), (1.5, 1.5), (1.5, -1.5) and 65535 as the maximum number of iterations.

Number of Processes	1	2	5	10	20
Elapsed Times (seconds)	3.29	3.40	4.49	6.87	8.79

It is evident that there is a strong increase in the communication times between processes, due to the fact that each process has to send to the master process its results, so as to enable the master process to create the final image.