TDDC78 - Report Lab3 - loibi806 - goubh275

Stationary heat conduction

To perform the heat simulation in parallel, we decided to split the grid into groups of columns. Each thread will work on a different group of column. However, we want to make sure that the correct values are used on each iteration of the process, i.e. each thread will use values from the previous iteration, and not some values potentially already updated by another thread during the same iteration.

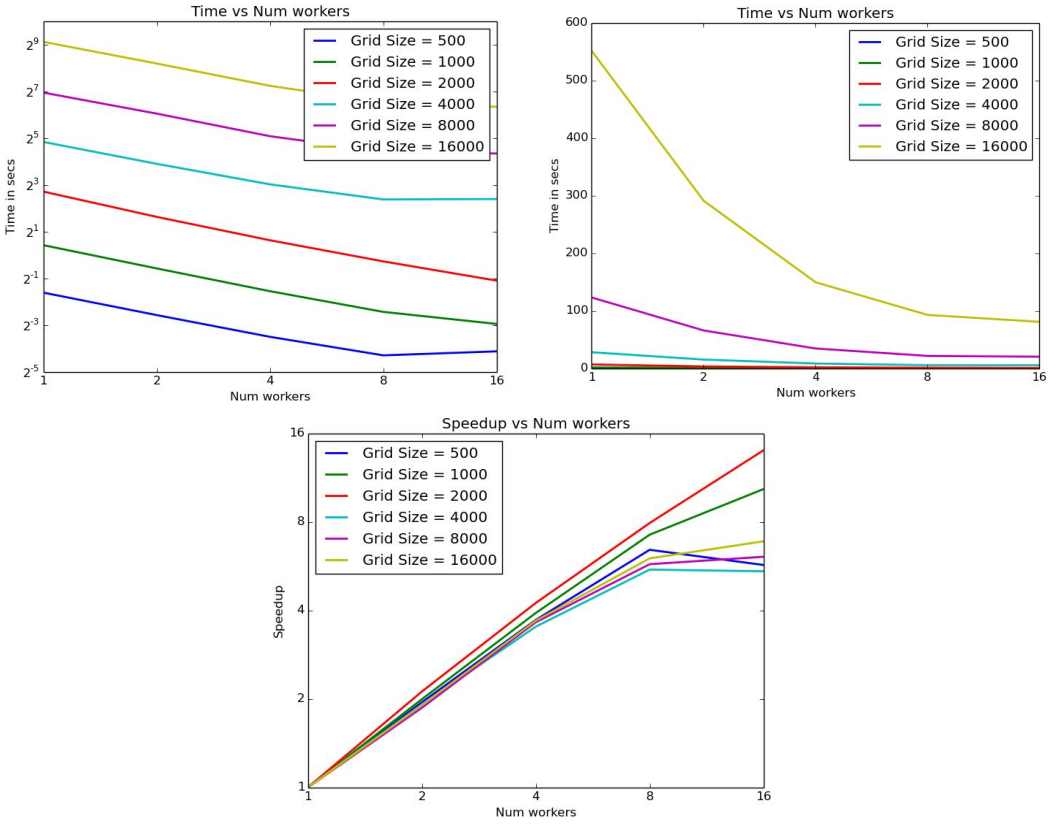
The outside loop (for the number of iterations) won't change, and isn't put in parallel. After the declaration of the original error, we start the parallelism using *!$omp parallel.* We declare *T*, the array, and *k*, the iteration variable as shared values, but all the other variables are private. Using the number of threads and the size of the grid, we can now give each thread a start and end column. The first and last column of the grid are not included, since they represent the border and have a fixed value.

Each thread will now save the first column outside of its group on the left and the right. These will be used to make sure that the correct values are used on each iteration, and will allow update the gris while still keeping the old values somewhere. We now iterate on all the columns from the group. This is similar the the sequential version, where we use the previously saved left column inside the computation to have the correct, non-updated values. We have a special case if we are the end of the group, where the column on the right outside of the group is used.

A local error is computed by computing the max difference between an element from the previous values of the column and the new values. It will be automatically put together with the errors from the other threads because we are using *reduction(max : error)* inside the omp header, which will find the max value of all local errors and send it to everyone.

A few other changes were made to the code to allow us to give the size as a parameter, but it doesn't interfere with the parallel implementation.

Results



We represented the same values but with two different scales we did some testing up to 16 threads, since it is the maximum available on one triolith node. We globally got some coherent results. A bigger grid size yield a greater running time, i.e. doubling the side of the grid by two increases the running time by four, and doubling the number of workers reduce by a factor of two the running time (approximately of course).

However, after more than 8 threads, we see a strange behaviour. For some sizes, the execution still decreases, but not for all of them. For small grids, e.g. 500 x 500, we can explain it by the fact that the cost needed for creating the threads and parallelising the code is higher than the improvement resulting from parallel execution. For bigger sizes (1000 and 2000), we observe a normal behaviour. However, if we keep increasing the grid size, the speedup decreases. It is hard to explain, but it might be due to some cache effects, were more rows have to be copied from main memory on each iterations, making the process slower.