Numerical Simulation and Scientific Computing

Exercice 3: Task 4

Elsayed Saleh Abrishq Bassam : 12302324

Ravelet Thomas : 12302809

Saudin Abdic : 11730114

Christoforos Eseroglou : e12331470

In the code, there are 4 different “for”-loops:

1) The main loop in the main function cannot be parallelized, as it iterates several times over the same vector. Therefore, it is impossible to calculate subsequent iterations until the results of the first iterations have been calculated.

2) The second loop is the nested loop that iterates all over the grid. Since these loops act on certain points of the "xnew" vector, it is possible to parallelize the execution of these 2 loops. Each thread must have its own indices, so i and j must be private. In addition, each thread must have access to the following variables: xold, xnew, h2, c, N. These variables must therefore be shared between all threads.

3) The third loop calculates north and south boundaries. Since it only calculates certain points on the grid, it can also be parallelized.

4) Same thing as the third loop but for the north boundary.

To greatly reduce computation time, the simulation is run with 60 cores at 2048 resolution and 10,000 iterations. Although it is asked to use 100,000 iterations, using only 10,000 iterations means that all simulations (with some error at the end, given that the number of iterations is insufficient to have an error close to 0) can be run in less than 10 minutes. What's more, this is sufficient time to evaluate the efficiency and scalability of the parallelization.

It's also important to note that the compiler used on the cluster is different from the one installed on my computer. To avoid any problems with the cluster compiler, I had to remove the N variable from the "shared" clause. This is because the value of N is caught by the solver's lambda function and therefore doesn't need to be specified among the shared variables.

The result of relative speedup is as follows:

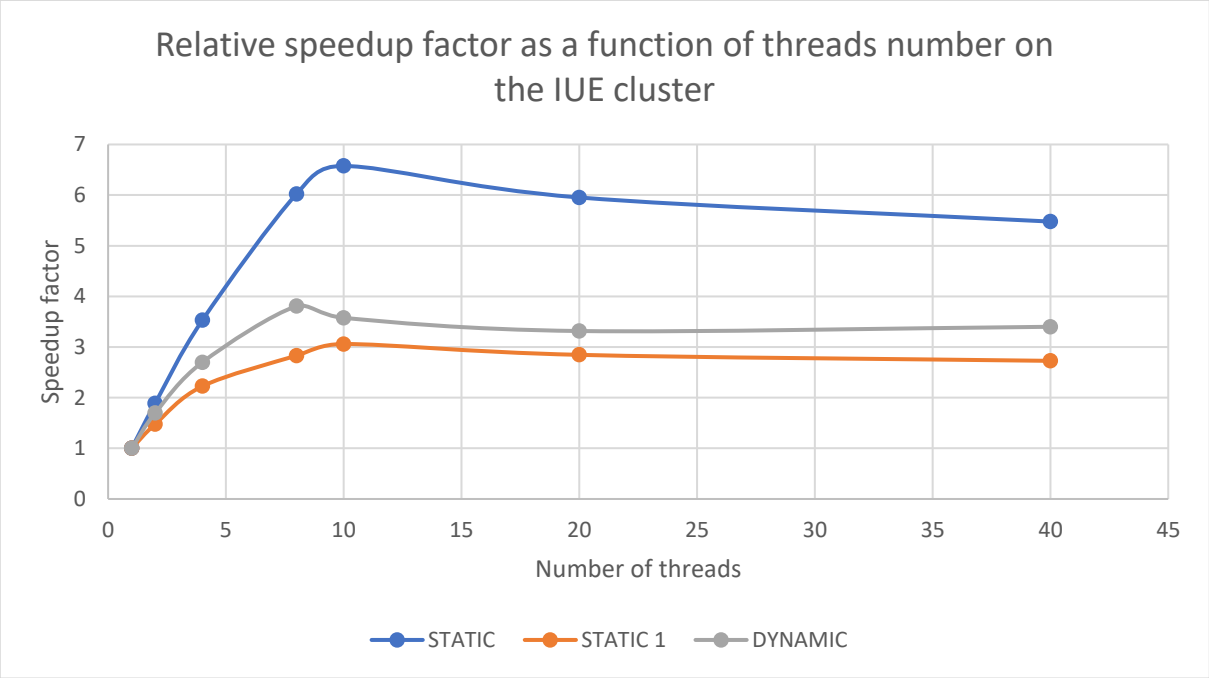

Figure 1: relative speedup of each type of scheduling on the IUE cluster

We immediately notice 2 things: the optimal number of threads is around 10, and using many more doesn't improve performance much. The second thing is that "static" scheduling is by far the one that benefits most from parallelization. We can see that with 10 threads, the static version sees its performance multiplied by 6.5.

To correctly evaluate the scalability of our parallelization, we need to look at the Time / Number of threads diagram:

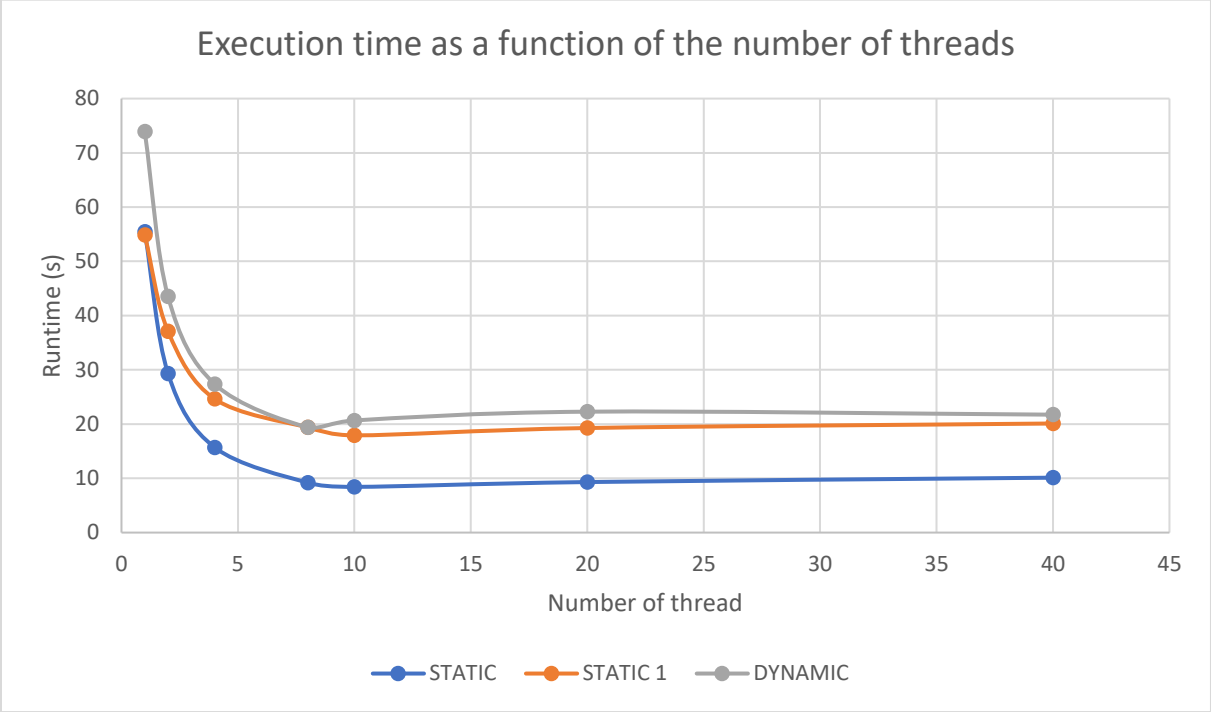

Figure 2: Execution time for each scheduling type as a function of the number of threads

Scalability is poor, since after 10 threads, execution times don't really decrease. Execution time decreases linearly with the number of threads, if the number of threads is between 1 and 5.

Static scheduling is the most efficient. There are several reasons for this:

- Dynamic scheduling requires to assign several times several threads to different chunks. If several threads finish at the same time, some of them must wait for others to be assigned to new chunks before they themselves can be assigned to a new chunk. This may explain why, when chunks are divided equally among all threads, dynamic scheduling is the least efficient.
- Static,1 scheduling does not require threads to be dynamically assigned, but chunk sizes may be too small, so threads need to stop and start working very regularly. The frequent stopping and restarting of threads, which must skip several memory chunks on a regular basis, is probably the cause of this scheduling's lack of performance.
- Finally, static scheduling is the most efficient, as it doesn't require regular re-allocation of threads to new memory chunks, and the memory chunks to be processed are well matched to the threads, thus optimizing overall program execution time.