Report

Assignment 2

1- Explain how you parallelized and optimized « simulate ». List the optimizations you applied and the reason you chose to apply them.

A. : The code is pretty simple. After setting our number of threads with the « omp_set_num_threads » function, we set an integer variable named « k » to the required number of iterations.

Then we enter in the parallelized part of the program. To avoid doing serveral fork-joins, we chose to join each thread only after doing all the required number of iterations, and so only at this moment, the final result of the output array is computed. Excess fork-joins lower performances, so we decided to use a « while loop » with condition that the variable k must be 0 (or less) to reach the end of the parallelized part of the program (and go back to the sequential one, which is not anymore in the « simulate » function).

In this « while loop », we use several tools from the OpenMP library : one « single », two « for », and one « barrier » directives.

The « barrier » directive will ensure that every threads wait each other after evaluating the condition, so that at this point we ensure that no thread reads a different value k, which could result in a final output array with incorrect values, because some threads would finish earlier and would not have performed some required computations.

The « single » directive is used to update our variable « k » which holds the current remaining number of iterations to do. This variable is shared by all threads so we only want one thread to update it, that's why we use this directive. The real optimization for this part is the use of the « nowait » clause which allows the other threads to continue their execution without waiting for the variable « k » to be updated. This will remove a useless implicit barrier and let the whole execution being faster.

We then have our two « for » directives with « schedule(static) » clause which will let the threads compute on different rows (only the rows of the array to compute are distributed between the threads because the directives are focused on the outer loop). These directives will optimize the computation of each element of the array (the first « for » is for saving the current values in a second array, and the second one is for computing the next values of each element of the array, thanks to the current ones). Dividing the work among several threads working at the same time participates in reducing the execution time of the program, especially when it concerns a huge array to compute. Because « for » directives are implicit barriers, to get a faster execution, we add the « nowait » clause at the second « for » to omit its useless barrier (because we already have one right after the « while » condition).

2- Report how much each optimization improves performance and explain the result obtained.

A. : To experiment our optimizations, we decided to run our program locally several times for each of them (2 processors available) with 4 threads on a 2500 by 2500 array and 500 iterations, and take the best execution time we could obtain.

We used a first version of our code which only executes sequentially (no parallel region). This took 16.53 seconds. From this point we are able to check how our optimizations improved our execution.

When adding the parallel region, and only the two « for » directives (no « nowait » clause), our execution run in 13.26 seconds. This is due to the fact that we divide the computation work among our threads with the « for » directives.

Our next optimization is the use of the « while loop ». We have to add our « barrier » directive just after the condition checking to ensure that every threads read the same number of remaining iterations, and the « single » directive to decrement this number by one each time only once. The execution at this time run in 13.33 seconds, a little bit higher than before. The main point of this optimization here is to remove the excess fork-joins we had at the previous step. The fact that the execution time is higher is because we have too much barriers in our parallel execution part.

We can safely remove two useless barriers by adding the « nowait » clause on the « single » and the second « for » directives. The program was able to fully execute in 12.84 seconds, which is better than the previous execution times. The « nowait » clause avoids threads waiting for each other to finish their current computation when it's not needed.

The last optimization we made is the « schedule(static) » clause associated to the two « for » directives. By doing this, our program reached an execution time of 12.43 seconds. The static scheduling type is appropriate when all iterations have the same computational cost, which is the case in our program.

3- Present the execution time you measured for the following set of thread counts {1,2,4,8,16} running for 500 iterations and side length of 10000.

A. : Table of results for assignment2.c program with all the optimizations previously mentioned using a 10000 by 10000 array and 500 iterations :

Number of threads	Execution time (in seconds)	Speedup over single threaded version
1	205.6	1
2	106.4	1.932
4	64.11	3.206
8	38.16	5.387
16	30.29	6.787