INSTITUTO SUPERIOR TÉCNICO

Wolves and squirrel : Part 1

Parallel and Distributed Computing CPD

Group 30 :
Cappart Quentin (77827)

Decomposition done

To understand our decomposition, we must explain before how our serial version works. The main work are done in the huge worldLoop :

```
for(i = 0 ; i < 4 * noOfGenerations ; i++){...}
```

What is particular is that we iterate to 4 times the number of generations. This structure was done to do the following things :

1. We proceed the red generation and we keep all the conflicting movement.
2. We deal with the conflicting movement of the red generation.
3. We proceed the black generation and we keep all the conflicting movement.
4. We deal with the conflicting movement of the black generation.

Each part are done in one particular iteration, the correct action are chosen thanks to modulo and conditional operation. With this structure we don't want to parallelize this main loop, but only his content. And after each iteration we need to synchronize. So, our parallelization are done like this

```
#pragma omp parallel for private(x,y,cell)
for(y = 0 ; y < worldSideLen ; y++){
      for(x = 0 ; x < worldSideLen ; x++){
         //proceeding the cells
      }
)
```

Let's notice that we only parallelize on the row and not on the column because of the cache and to keep the adjacent portion of memory together.

Conflict resolution

The conflict resolution is done by the architecture of the serial version. Indeed, all the conflicting movement are not made directly but their are kept in memory and proceeded after each subgeneration with the `update` method. So, with the parallelization, given that we have an implicit synchronization after each iteration of the main loop (generation loop), the conflict are implicitly resolved. We made sure after that our outputs with severals threads are always the same than the serial version.

Load Balancing

Performances analyze

The following array recaps the execution time of the OMP version with severals numbers of threads.

We did these experiments on a computer with 4 cores with the input `instance 10 9 8 100`. We let to the "iteration print" in the code [1]. First, we notice that for the smallest instance, `ex3.in`,

1. If we remove it, the execution time is to fast to do an useful analyze.

Instance	1 thread	2 threads	4 threads	8 threads
ex3.in	0.008	0.008	0.017	0.020
world_10.in	0.011	0.006	0.021	0.035
world_100.in	0.111	0.070	0.068	0.1278
world_1000.in	12.885	7.712	6.534	6.522

Table 1 – Performances (in sec) for the different instances.

more we have threads, more the program is slow. This is normal. Indeed, given that the instance are small, the overhead of the threads (creation, repartition and synchronization for example), surpass the gain that we have by the multithreading. For the instance `world_10.in`, we notice it from 4 threads. For the biggest instance, `world_1000.in`, we see the speedup with severals threads. The speedup decreases with the number of thread [2]. A last comment is that, given that the computer has 4 cores, we loose the interest of multithreading after 4 threads.

Now, let's compare the execution time for world_1000.in with the ideal speedup [3] with different numbers of threads :

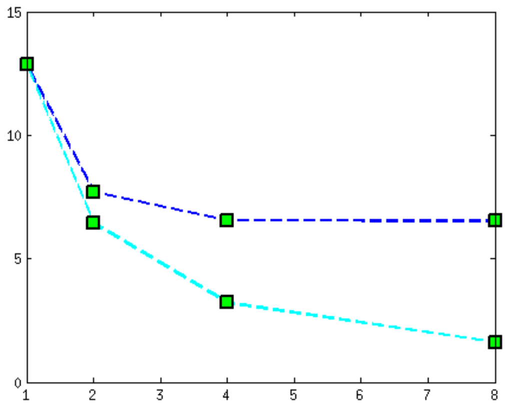

Figure 1 – Comparison with the ideal speedup.

We two threads we are relatively close to the ideal speedup. However the gap grows with more threads. We expected this king of results. Indeed, there is always overhead when we deal with threads and all of our code is not multithreaded, so it's normal that we don't reach the ideal result.

2. We gain more from 1 to 2 threads than for 2 to 4 threads.
3. $S = \frac{T_{serial}}{N_{threads}}$