
# Wolves and squirrel : Part 2

# Parallel and Distributed Computing
# CPD

**Group 30 :**
Cappart Quentin (77827)
Mikołaj Jakubowski (77610)
Paulo Tome (72419)

# MPI Architecture

Our program is based on a Master and Servant model. In this model, we have two sorts of computer, the master and the servants. In a few words, the master is the coordinator of the work, and the servants will deal only with a sub-board of the total board.

### Master

There is only one master in the application. He's in charge of coordinate all the work. He does the following actions :

1. Loads the world from the input file.
2. Splits the world in different parts (One per servant).
3. Sends a sub-board to every servants.
4. ???, Whats this step ?
5. Sends a message to the servant to tell that he has send all the cells.
6. Proceed a loop of $2 \times$ `Number of generation`.
   (a)
7. Sends a message to all the servants to notify that all the generations are finished.
8. Starts to listen for an update message from the servants. This message contains the state of the cells.
9. Saves the contains of the cells into the world array.
10. Counts the number of finished messages from the servants. When all the servants has sent the message, the computation is finished.
11. Prints the world and exits the program.

### Servants

All the other computers are the servants. Each receives a sub-board from the master

# Ghost lines

# Performances analysis

# Decomposition done

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

```
1 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

```
1 #pragma omp parallel for private(x,y,cell)
2 for(y = 0 ; y < worldSideLen ; y++){
3       for(x = 0 ; x < worldSideLen ; x++){
4          //proceeding the cells
5       }
6 )
```

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

Logic of every sub-generation is divied in two steps. Selecting updates and updating itself. Second phase is done per cell, so it is easy to parallelize. Every cell has a list of updates that needs to be invoked on it. Load balancing is done by diving those lists evenly between cores by dynamic sheduling. It's done by this pragma in the `update` fonction :

```
1 #pragma omp parallel for schedule(dynamic,1)
```

We add it in the code, but it gave a slower execution time, so we didn't integrate it in our final version.

## Performances analysis

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

| 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.

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`, more threads we have, the program is slower. This was expected. Indeed, given that the instance are small, the overhead of the threads (creation, repartition and synchronization for example), surpass the gain of multithreading. For the instance `world_10.in`, we notice it from 4 threads. For the biggest instance, `world_1000.in`, we see the speedup when we deal with severals threads. The speedup per core decreases with the number of threads[2]. Finally, given that the computer has 4 cores, we are not interested in creating more than.

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

With 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 not all of our code is multithreaded, so it's normal that we don't reach the ideal result.

[1] If we remove it, the execution time is to fast to do an useful analyze.
[2] We gain more from 1 to 2 threads than for 2 to 4 threads.
[3] $S = \frac{T_{serial}}{N_{threads}}$