INSTITUTO SUPERIOR TÉCNICO

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 nodes, the master and the servants. In few words, the master is the coordinator of the work, and the servants will deal only with a sub-board of the total board. Let's describe all of this more concretely.

MPI Message

We have severals type of messages between the master and the servants :

1. `NEW_BOARD(side:int)` : Message sent by the master to the servants to give them a part of the board.
2. `UPDATE_CELL(c : cell_t)` : Message sent by the master or by the servants to notify a modification on particular cells.
3. `FINISHED()` : Message sent by the master or by the servants to notify the finishing of a particular task.
4. `START_NEXT_GENERATION(RED or BLACK)` : Message sent by the master to tell the servants to begin the computation of a new generation.

Master

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

```
Load the world from the input file
Split the world in different parts (One per servant)
Send a sub-board to every servants.
Send UPDATE_CELL to the servants with the cells of their sub-bord + the cells in
the bord around.
Send FINISHED to confirm sending all cells

for i in 0 -> 2*Nb of generations:
    Send START_NEXT_GENERATION(color) to every node
    Listen for incoming updates and save them
    Count FINISHED messages
    if(count == Nb of slaves):
   break;
    Send stored updates to all servants
    Send FINISHED to all servants

Send FINISHED to all servants // all generations are finished
Listen for UPDATE_CELL and save them
Count FINISHED messages
if(count == Nb of slaves):
    Print output
    Exit
```

Servants

All the other computers are servants of the master. Each of them receives a sub-board. The distribution of the work follow this idea. Each servants do the following actions :

```
Listen for NEW_BOARD message
Allocate memory for its board part
Listen for FINISHED message // all cells are in place
while true:
   Listen for message
   if(message = FINISHED):
  break;
   elif(message = START_NEXT_GENERATION(genInfo):
  Start a new generation
   Compute its part of the board
   Send UPDATE_CELL to the master the message with the modified cells
   Send FINISHED to the master
   Listen for UPDATE_CELL messages from master
   Update the cells and resolve conflicts
   Listen for FINISHED message from master
Sends UPDATE_CELL to the master with all the cells.
Exit
```

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]$S = \frac{T_{serial}}{N_{threads}}$