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(from:Position, to:Position)` : 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 :

```
1  Load the world from the input file
2  Split the world in different parts (One per servant)
3  Send position of a sub-board to every servant.
4
5  for i in 0 -> 2*Nb of generations:
6      Send START_NEXT_GENERATION(color) to every node
7      Listen for incoming updates and save them
8      Count received FINISHED messages
9      if(count == Nb of slaves):
10       break;
11     Send stored updates to all servants
12     Send FINISHED to all servants
13
14 Send FINISHED to all servants // all generations are finished
15 Listen for UPDATE_CELL and save them to master board
16 Count FINISHED messages
17 if(count == Nb of slaves):
18     Print output
19     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 :

```
1  Load the world from a file
2  Listen for NEW_BOARD message to select a piece
3
```

```
4   while true:
5      Listen for message
6      if(message = FINISHED):
7        break;
8      elif(message = START_NEXT_GENERATION(genInfo):
9      Compute its part of the board
10     Send UPDATE_CELL to the master the message with the modified cells
11     Send FINISHED to the master
12     Listen for UPDATE_CELL messages from master
13     Update the cells and resolve conflicts
14     Listen for FINISHED message from master
15
16  Sends UPDATE_CELL to the master with all the cells.
17  Exit
```

Conflict resolution

Given that the algorithm keeps the same idea than the serial version, the way conflict resolution will be nearby the same. The only difference is in the sub-board edges where the servants need information about a part of the board that not belongs to them. In these cases, it's the master that's in charge to send messages to ensure the consistency of the board.

Load balancing

With this model, all servants receive a part of the board of the same size[1] However, the number of dynamic elements in each parts is not taken into account for the shattering of the board. So it's possible that some servants has more or less work to do.

Performances analysis

The following array recaps the execution time of the MPI version with severals numbers of nodes in the cluster.

Instance	2 nodes	4 nodes	8 nodes
ex3.in	6 sec	8 sec	13 sec
world_10.in	5 sec	8 sec	
world_100.in	10 sec		
world_1000.in	6 min 1 sec	3 min 52 sec	3 min 19 sec

Table 1: Performances for the different instances.

Instance	Serial	2 nodes	4 nodes
ex3.in		6 sec	8 sec
world_10.in	0.028 sec	5 sec	8 sec
world_100.in	1.86 sec	10 sec	
world_1000.in	2m20.469s	4m27.751s	3 min 52 sec

Table 2: Performances for the different instances.

[1] Except the last, due to the remainder of the division.

We did these experiments on the `Borg` cluster with 2000 generations.

At a first sight we can observe that the execution time tends to increase with the number of nodes for the small instances, and to decrease for the big ones. This can be explained easily. The bottleneck of a distributed program is the cost of the communications between the nodes. For the smallest instances, the gain of the distribution of task is overtaken by the cost of the communication. Contrariwise, for the biggest, we can see the interest of the distribution. The computation of the board begins to be costly, and the distribution between servants is useful. We can see it for the `world_1000.in` instance where the execution time significantly decreases.