[INFO-F404] Real-Time Operating Systems
MasterMind - Project report

BUI QUANG PHUONG Linh - ULB ID : 000427796
PAQUET Michael - ULB ID : 000410753
SINGH Sundeep - ULB ID : 000428022
MA1 Computer Sciences

December 2018

Introduction

The goal of this project is to implement the *MasterMind* game using a parallel solver. To do this, we will use the interface MPI on HYDRA cluster. This parallel solver implementation will help to prove the quickness of parallelism.

1 Pseudo-code and implementation choices

1.1 Code structure

For this project three main files are used :

1. `master.cpp`
2. `player.cpp`
3. `mastermind.cpp`

1.1.1 The master entity

The `master.cpp` is the entity representing the master in the game. Its main purpose is to create a solution, print it and evaluate the solution given by a player.

- `void Master::buildSolution()`: build the solution given the number of spots and number of colours
- `void Master::printSolution()`: display the solution created randomly
- `vector<int> Master::evaluate(vector<int> solution)`: take a vector of integers as a parameter which will be the solution of a player and will evaluate it

1.1.2 The player entity

The `player.cpp` is the entity representing the player in the game. The player will create a set of possible solutions and send them to the master for an evaluation.

- `void Player::generateSolutions()`: generate the possible solutions of a player. It will choose a colour whom will be the first colour for every solution created by the player then it will create a subset of colour by considering the number of spots and colours available. Once a subset is created, the player will do all the permutation possible on that subset to create all the solution possible. Finally it will create a new subset and it will do it all over again until there is no more possible solution possible given his first choice of colour.
- `bool Player::isPlausible(vector<int> solution, vector<int> evaluatedSolution, vector<int> evaluation)` : will check if one solution of the player is possible given an evalution of another player. Let x be the number of *perfect* and y the number of *color only* then a solution is plausible regarding an evaluation if and only if it matches $(x + y)$ colors in the evaluation including x times perfectly.
- `vector<int> Player::createStopSolution()`: sends a fake solution constituted of zeros to the master to indicate him that the player doesn't play anymore

1.1.3 The MasterMind entity

The mastermind is the entity that will handle the running of the game. It will make a distinction between the master and the players due to their rank. Moreover, the master will listen for the possible solutions of the players, evaluate them and diffuse the result of the evaluation to all the players so that everyone can send another plausible solution. Consequently the player here will be sending a solution and waiting for a response of the master which will be the evaluation

1.2 Pseudo-code

1.2.1 `mastermind.cpp`

The main file `mastermind.cpp` is implemented following the pseudo-code presented below.

Algorithm 1 MasterMind implementation

```
1:  procedure MasterMind
2:      Initialization of MPI
3:      if is Master case then
4:          Generate a solution of w number of spots from z colors
5:          while a player doesn't find the solution do
6:              Gather the solutions received from the players
7:              for each solutions received do
8:                  print it and determines if it's a legit solution
9:              Pick a legit solution in the solutions list
10:             Evaluate the solution picked
11:             Broadcast the evaluation of the solution and the evaluated solution
12:
13:     else if is Player case then
14:         for each color j in the set of colors w do
15:             Attribute the color j to a Player
16:         Generate all possible solutions of the current Player
17:         while the Player doesn't find the solution of Master do
18:             Find a plausible solution
19:             Send the plausible solution to the master (Gather)
20:             Receive the evaluated solution from the master (Broadcasted by Master)
21:             Receive the evaluation from the master (Broadcasted by Master)
```

1.2.2 `master.cpp`

The `master.cpp` file is composed by two main methods : `buildSolution` and `evaluate` respectively building a solution of a size corresponding to the number of spots chosen from the colors list and evaluating a solution given by a Player. `buildSolution` is represented by the pseudo-code in **??** and `buildSolution` is represented by the pseudo-code in **??**.

Algorithm 2 Building a solution of z spots from the w colors

```
1: procedure BuildSolution
2:     for each spot do
3:         Pick a random color from the w colors
4:         Check that this color is not already in the generated solution
5:         Add the color to the solution
```

Algorithm 3 Evalation of a solution

```
1: procedure Evaluate(solution)
2:     for each spot of the solution given do
3:         if color found in both the solution given and the true solution at this spot then
4:             Increment the number of perfect
5:         else if color found but not in the good position then
6:             Increment the number of colorOnly
7:     if perfect number = spots number then
8:         solution found → set gameNotOver value to false
9:     return the number of perfects and color only
```

1.2.3 `player.cpp`

The `player.cpp` file is composed by the methods `generateSolutions`, `isPlausible` and `getPlausibleSolutions`. The first one is generating all the possible solutions starting from a certain color while the two following are checking if a solution is plausible from the evaluation returned by the Master and are getting all the plausible ones.

Algorithm 4 Generation of the solutions

```
1: procedure generateSolutions
2:     Initialization of the vectors color, solution, itand it2
3:     for each colors attributed do
4:         Create all subset without that color
5:         for each subset do
6:             Compute all permutation of the subset
7:             Add all computed permutation
8:         for each subsets added do
9:             Add own color in first position of subset
```

Algorithm 5 Get the plausible solutions

```
1: procedure getPlausibleSolution(bool isBegin)
2:     if sending first plausible solution then
3:         send first solution of all the plausible solutions
4:     for each possible solutions do
5:         for each evaluations sent by the master do
6:             if current solution is not plausible given the evaluation then
7:                 set this solution has not plausible
8:         if the solution is plausible then
9:             return the solution
```

Algorithm 6 Verify if a solution is plausible

```
1: procedure isPlausible(solution, evaluatedSolution, evaluation)
2:     x equals the number of perfect matches of the evaluated solution
3:     y equals the number of colors only of the evaluated solution
4:     for each element of our plausible solution do
5:         for each element of the evaluation sent by the master do
6:             if an element of the plausible solution matches an element of the evaluation then
7:                 Increment the number of matches
8:             if position of the elements are the same then
9:                 Increment the number of perfect matches
10:    if number of matches equals to (x + y) and the number of perfects equals to x then
11:        set the solution as plausible
```

2 Protocol description

There are two communications between the master and the player (represented by nodes).
The master can either receive solutions from the nodes or send to the nodes the evaluation of a solution. On the other hand, it should not be necessary to say that a node can either send a solution to the master or receive an evaluation from him.
Here is how we implemented it in order to do so :

- Receiving a solution from all processes (all nodes) can be interpreted as multiple receive on master's side and a send from each node on player's side, which is exactly the definition of the *MPI_ Gather* function. To implement such communication, we then call that function on both side, specifying what to send/receive, who will receive, and other useful parameters.
- Sending information from a process (the master) to all other processes (the nodes) can be interpreted as multiple sends master's side and a single receive player's side which is exactly the definition of the *MPI_ Bcast* function. We then call it on both side in order to send the evaluated solution with it's evaluation to all nodes.

To resume the protocol : first, all nodes compute a plausible solution while the master is blocking on the *Gather* function waiting for a solution from all nodes. Once computed, each node call the *Gather* function to send their solution.
Now that the master gathered all solutions, he can compute the evaluation of a randomly chosen one while the nodes are blocking on a *Bcast* function.
Once the evaluation computed, the master broadcasts it to each node so that they can again find another plausible solution and so on.

3 Performance description

Let x be the number of all possible guesses and p the number of processes.
Evaluating a solution on master's side can be considered as constant in term of computation time. The two variables are the number of guesses to be generated by each node and the number of solution we need to get through before reaching the good solution.
Without parallelism :

- We would generate x possible guesses.
- We would at most get through x possible guesses before finding the good one.

With parallelism :

- Each node generates $\frac{x}{p}$ possible guesses.
- We at most get through $\frac{x}{p}$ guesses before reaching the solution.

It has to be said that, currently, in some case, parallelism could not be worth.
Indeed, let *Node 1* have the guesses <0,...,n> and let *Node 2* have the guesses <n+1,...,x> and n be the solution of the game.
When looking for a plausible solution, if *Node 1* finds *0* and *Node 2* only finds x, then *Node 1* have to wait in its call to the Broadcast function *Node 2* to get through all its possible guesses. At next round, *Node 1* then have to get through all its guesses to find out that n is the good solution.
In this case, we then got through all possible guesses instead of only n in case of non-parallelism. A way to make parallelism worth in any case would be to still get through guesses while waiting the other nodes in the Broadcast.

4 Calculation of the formula of possible guesses

Let n be the number of colors and k the number of spots. The number of subsets of k elements in n is defined as

$$C_k^n = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

For each subset, we permute it in order to have all possible guesses. The number of guesses given by the permutation of a subset of size k is $k!$.
Then, the final formula to calculate the number of possible guesses is :

$$k!\ C_k^n = k!\binom{n}{k} = k!\frac{n!}{k!(n-k)!} = \frac{n!}{(n-k)!} = A_k^n$$

5 Difficulties encountered

Two problems were encountered during the implementation of the project :

- The first one was to correctly set up the communication between player and master. Indeed, we never had to use functions such as *MPI_ Gather* for example, therefore it was at first hard to see how to make it work properly. An explanation on how we made it can be found in the Protocol description section.
- The second problem, which is still remaining, is the fact that when a node has no more plausible solution, he must stop running and inform it to the master so that he does not wait for a message from that node anymore. As just said, this problem is still remaining but here is how we currently deal with it : once a node finds out he has no more plausible solution, he will keep running but always sending a "fake" solution which the master will understand as "Do not evaluate this".