[INFO-F-404] Real-Time Operating Systems
Project 2 - Mastermind

Ricardo Gomes Rodrigues - 000443812
Alexandre Heneffe - 000440761
Madalin Ionescu - 000442828

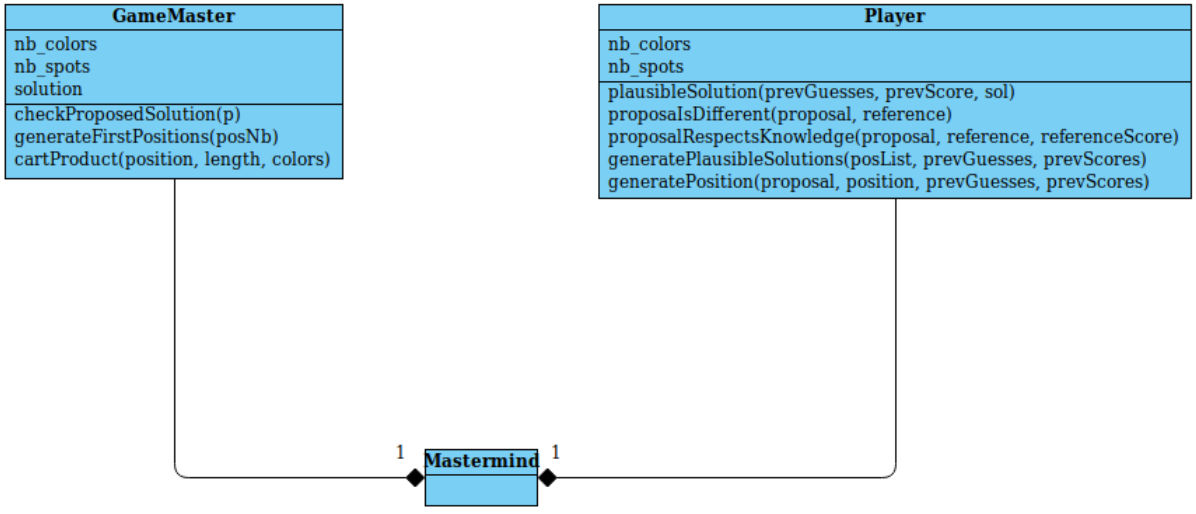
December 2019

1 Introduction

The objective of this project is to present a possible implementation of the *Mastermind* game, in C++, using multi-processing and the *Hydra* cluster. A base observation regarding this implementation is that, mimicking the real game, in which the number of rounds is limited, it tries to keep the number of guesses checked by the game master minimal.

2 Implementation choices

2.1 Class diagram



1. The gameMaster class represents the Master player possesing the solution of the game. He is in charge of evaluating some guesses and initialise the game for the challenger.

2. The Player class represents the Challenger player, able to generate plausible guesses that can be assessed by the game master.

3. Mastermind is not a class strickly speaking, but rather a representation of the game itself, having exactly one game master and one challenger (each process will access this challenger object and use its methods).

2.2 Combinations management

We know that there are C^S (where C is the number of colours and S the number of spots) possible combinations (i.e. *guesses*), which is the cardinality of cartesian product of the set of colors for each spot. Computing all those combinations sequentially and evaluating them is hence not feasible. Therefore, in order to split the work into multiple processes, we first compute all the possible combinations of *colors* for the *number of fixed spots*, namely the prefix of a plausible solution. Each node will then receive its unique prefix (a *color* for each *fixed spot*) and compute the following positions, until founding a plausible solution. This way, for each fixed position, we divide the number of combinations to be generated by the number of *colors*, by taking advantage of the in-parallel execution. It is important to mention that, in order for the previous statement to hold, $C^{S_{fixed}}$ must be smaller then the number of available CPUs.

Exemple Suppose we have 4 colours $(0, 1, 2, 3)$, 5 nodes (because we count the master node as well), 3 spots in total and 1 fixed spot for the prefix. The workload distribution will take the following shape :

Node 0 : this is the master. It will assign a fixed beginning of length 1 to each node. Therefore, there will be a list [0, 1, 2, 3] which will be used to send, to each node, its prefix.
Node 1 : will check the combinations starting with a **0** : [**0**, 0, 0], [**0**, 0, 1], [**0**, 0, 2], [**0**, 3, 0], ..., [**0**, 3, 3]
Node 2 : will check the combinations starting with a **1** : [**1**, 0, 0], [**1**, 0, 1], [**1**, 0, 2], [**1**, 3, 0], ..., [**1**, 3, 3]
Node 3 : will check the combinations starting with a **2** : [**2**, 0, 0], [**2**, 0, 1], [**2**, 0, 2], [**2**, 3, 0], ..., [**2**, 3, 3]
Node 4 : will check the combinations starting with a **3** : [**3**, 0, 0], [**3**, 0, 1], [**3**, 0, 2], [**3**, 3, 0], ..., [**3**, 3, 3]

The algorithm presented hereabove still has a huge complexity. For this reason, we chose to implement multiple optimizations, which we will present later in this report.

3 Communication Protocol

3.1 Master

Tasks corresponding to the master node :

- Compute the cartesian product of the colors for the number of fixed spots and send to each node the corresponding prefix.
- Broadcast the list of previous guesses to all nodes, as well as the list of previous evaluation of these guesses.
- Receive a new guess from each node and pick one randomly (if the guess is a list containing only the number of colors for each cell, it means that the node that has sent this guess hasn't found any plausible solution, therefore the master will pick another guess)
- Broadcast a value to all nodes which represents a boolean specifying if the node should finish the loop or not because the solution has been found already.

3.2 Challenger

Tasks corresponding to the challenger node :

- Receive the prefix
- Receive the result of the broadcast from the master : the list of previous guesses and the list of scores
- Generate a plausible solution and send it to the master

- Receive an integer specifying if it must stop (because a solution has been found by him or another challenger).

3.3 Main loop

Now that we have defined the tasks of both the master and its challengers, we can simply put these in a `do while` loop (so it runs at least once) and keep looping until the solution is found. We have to pay special attention to a particular case, represented by the first round, when we use the GameMaster to generate a secret solution, which must be found by the at least one challenger by the end of the game.

3.4 Data Types

We are sending and receiving 3 kinds of data types :

- A simple Integer
- A vector of unsigned
- A vector of vectors of unsigned

For the vectors, we send and receive pointers as in the following example:

```
MPI_Bcast(&vec[0], vecSize, MPI_INT , MASTER_ID, MPI_COMM_WORLD);
```

4 Performances

4.1 Brute Force

Using brute force is probably the easiest algorithm. It will try all the possible combinations until finding the right one. There are C^S (where C is the number of colors and S is the number of spots) possible combinations. This method is actually faster, for smaller games, than the one discribed in section 4.2 because of the reduced number of required computations with respect to the other (i.e. keeping vectors into memory, comparing to previous guesses, etc). But the downside is that it does not respect the constraints given in the statement of the project. To be more precise, each node will compare its current guess to the solution. But the node can not know the solution.

4.2 Our Solution

4.2.1 Principle

The implemented solution is based on the idea of improving the guess at each round. That is, each new guess will be at least as good as all the previous ones. To do that, we compute for each such combination of colors the number of changed ones, with respect to all the previous guesses. This number should never be greater than the number of wrong colors in the guesses' history. We also impose that, when all colors of the actual solution are found, we will not change them anymore, only their order in the proposed solution will be different.

In order to minimize the number of tested combinations at each round, each solution is built step-by-step and after completing one new position, the partial guess is tested. In the case when this partial guess fails the aforementioned improvement criteria (i.e. it changed more colors then the number of wrong colors with respect to some other previous guess), it will abandoned. This improvement significantly reduces the number of combinations to test. For example, suppose that we already found $S - 2$ out of the S colors in the solution, so the respective proposal contains 2 wrong colors. During the next round, when building a new proposal, if none of the first 3 colors is included in the aforementioned solution, we will abort the construction of proposal having the respective prefix. By doing this incremental test, in this specific case, we

avoid building and testing C^{S-3} combinations of colors, where C is the number of colors the game accepts and S is the number of spots (i.e. the length of the solution). We can generalize this formula:

$$\text{number of avoided combinations} = C^{S-\alpha}$$

In the notation hereabove, C is the number of colors in the game, S is the number of spots and α is the length of the prefix of a proposal which does not respect the knowledge induced by the history of guesses.

4.2.2 Plausible solutions

In order to check if a guess made by a node is plausible or not, we introduce some conditions :

1. The intersection between the guess and the last previous guess must contain at least the same number of colors as the evaluation obtained by this previous guess.

   Example {1 2 3 4}, evaluation = (2, 1), sum = 3 good colors
   {1 2 3 5} = plausible since at least 3 same colors as the previous one
   {1 2 4 6} = not plausible since just 2 colors are the same as the previous

2. If the new guess is equal to one of the previous guesses, it is, of course, not plausible.

3. If, the size of the intersection between the previous guess and the new guess is smaller than the evaluation of the previous guess, it means that the new guess is not plausible.

   Example {1, 2, 3, 3}, evaluation = (2, 0)
   {1, 2, 4, 5} is plausible since the size of the intersection of sets (thus {1, 2, 3} and { 1, 2, 4, 5}) is equal to the size of {1, 2} which is 2. It is bigger or equal to (2, 0), therefore the guess is plausible.
   {1, 4, 5, 6} is NOT plausible since the size of the intersection of sets (thus {1, 2, 3} and { 1, 4, 5, 6}) (which is equal to the size of {1}) is 1. It is smaller than (2, 0), therefore the guess is NOT plausible.

4. If, for a guess, we have have an evaluation of (0, 0). We can eliminate any permutation of the colors of this guess

   Example {1, 2, 3, 4}, evaluation = (0, 0) {4, 3, 2, 1} is not a plausible guess.

4.2.3 Run time

Here are some instances that we run on Hydra and their timing:

Example 1 4 spots, 6 colors. Solution = [2, 1, 0, 0]. Time to find de solution : 0.005 seconds.

Example 2 4 spots, 6 colors. Solution = [4, 4, 1, 2]. Time : 0.052 seconds

Example 3 5 spots, 6 colors. Solution = [4, 2, 1, 5, 2]. Time : 0.613 seconds

Example 4 5 spots, 9 colors. Solution = [0, 8, 2, 7, 8]. Time : 2.118 seconds

Example 5 6 spots, 9 colors. Solution = [2, 3, 3, 6, 5, 5]. Time : 90.28 seconds

Example 6 7 spots, 8 colors. Solution = [7, 0, 3, 0, 5, 2, 1]. Time : 68.57 seconds

Example 7 10 spots, 10 colors. Solution = [7 7 3 5 4 2 7 3 6 8]. Time : more than an hour (and potentially crashes).

Note: With a simple brute force algorithm where each node would know the solution, we had a runtime of less than 1 second for 20 colors and 10 spots.

4.3 Possible improvements of the given solution

4.3.1 Challengers' personal knowledge

In the presented solution, the challengers' knowledge is based only on the history of guesses and their evaluation. An improvement to that would be to add a second type of knowledge, based on personal experience, namely on the result given by each node in the previous rounds: if a node found no solution for a given prefix in a previous round, it will never find one in the following ones, since the constraints are only getting stricter. In order to do that we have to move the responsability to generate the prefix from the *game master* to the *challenger*. This way, when enough prefixes are eliminated, we can make them longer by adding another position to the prefix. As we already mentioned, the cardinality of the cartesian product of the set of colors for each position of the prefix must be smaller than the number of available nodes.

4.3.2 Keeping the nodes busy

A second improvement might regard the amount of work each node has to do and the time it takes to finish its tasks. Since each node returns the first plausible solution found, considering its prefix, some might finish their work a lot sooner than others. In this case, considering the current implementation, all the challengers will just wait for the others to finish, before receiving a new workload. This results in possibly significant idle times. In order to avoid that, we can forget about the relation between the length of the prefix and number of nodes and set a prefix way larger than that. Each node finding a solution for a prefix sooner than the others will send his solution back to the master and will receive, if not all the prefixes are already assigned to a node, a new one to work on. We can limit the length of the prefixes such that the possible solutions corresponding to each such prefix can fit in the master memory. When the master received a solution for each prefix, it can choose one randomly and evaluate it, and repeat the step until the challenegers find the correct answer. This improvement reduces the number of combinations the challengers have to compute each time, reducing also the possible time interval between the first job who finished and has no new prefix to receive and the last one to finish. However, the communication times (between the challengers and the master) become more significant with respect the actual processing times, hence while implementing this improvement we need to pay good atention to the length of the prefixes, in order not to reduce the working time of each node, for each such prefix, too much.

5 Difficulties

5.1 MPI

The usage of MPI has been quite challenging because of the exchange of messages (especially the vectors and the vectors of vectors) between the master and its challengers (the other nodes). We had to restart this part from scratch because we had some problems with how broadcasting works, since it has to be used on both the master and the challengers. This means that the challenger does a broadcast even though it receives data.

5.2 Algorithm

The difficulty was to split the workload between multiple processes equally and efficiently. We had explored different algorithms such as the Knuth algorithm, the genetic algorithm or the simple brute force but these did not respect the constraints given and/or could not be easily converted to a multi-processing procedure.

We believe that, by implementing the aformentioned optimizations, we can achieve an efficient solution of the Mastermind game. However, even after multiple trials, we could not arrive at a satisfactory implementation, mostly due to the limitations imposed by the MPI interface and the way the communication is effectuated.

6 How to run

To run the code, follow the steps :

1. Log in on Hydra
2. Run : `module load OpenMPI/3.1.4-GCC-8.3.0`
3. Run : `make`
4. Run, for 6 colors (+1 master) and 4 spots : `mpirun -np 7 mastermind 4`

7 Conclusion

We presented here a possible way to solve the Mastermind game, which proves itself correct, as the tests on relatively small scale games reveal, using multi-processing and the Hydra cluster. Our implementation mimics the real-life game, keeping the number of guesses checked by the game master as low as possible. We also looked at the limitations and the performance of such imeplementation, as well as we proposed a few possible ameliorations.