Fast and Furious Game Playing : Monte Carlo Drift

Specifications report

Prateek Bhatnagar, Baptiste Bignon,
Mikaïl Demirdelen, Gabriel Prevosto,
Dan Seeruttun--Marie, Benoît Viguier

Supervisors: Nikolaos Parlavantzas, Christian Raymond

11/27/2014

Abstract

This second report is related to the incremental development of the project *Fast and Furious Game Playing: Monte Carlo Drift*. This report is about the specifications of the project. A detailed study has been reported here about various components of the project. This report is a step ahead of the previous report and focuses on "what" factor ,about each and every component of the project that includes game, general architecture, application programming interface, Artificial Intelligence- implementation of MCTS Algorithm, User Interface and Application Program. A detailed focus is also made on the parallelisation techniques using OpenAcc and OpenMP.

Contents

1 Introduction

The project is called *Fast & Furious Game Playing, Monte Carlo Drift*. Its purpose is to create an Artificial Intelligence able to compete against humans using the *Monte Carlo Tree Search* algorithm.

We have chosen the game Arimaa to play ~~with. This game is good for us~~ because it is a two-players strategy board game not solved[1].

A human plays a game by thinking of all possible moves as per one's imagination and then opts for the best amongst them. Before taking ones turn, a player can visualize ones options and predict how an opponent will counteract them. All studied move solutions will be displayed in a search tree. The Minimax algorithm does exactly the same thing, but it explores all possibilities, which is heavy. That is why we have chosen to use the MCTS algorithm, lighter, and converging to the Minimax algorithm.

By exploring the numerous random possibilities at any one time, our program will be able to take decisions in order to win the game. The algorithm would be parallelized in order to exploit it in a multi-core machine, allowing it to go further into the search tree, thus improving its efficiency.

Some of the best parallelization methods will be explained more deeply to choose the most suitable to this project. The exploration of the tree will depend on the parallelization method. Finally, in the last phase, a solution will be implemented, and executed on Grid'5000, on a set of clusters of multi-core machines.

In this report, the parallelization method chosen will be explained, and the MCTS algorithm will be ~~described~~ Then, the general architecture of our project will be described, the behaviour and the API of our game. Finally, we will discuss the different suitable software with OpenMP, OpenACC, and MPI.

[1] A game solved is a game where good algorithms are able to find the perfect move in each situation to win, or to draw. For instance, *Tic Tac Toe* or *Draughts* are solved games.

2 Algorithmic methods

An explanation of our methods and concepts is necessary to understand how our project works. The use of some parallelization methods, and the MCTS algorithm is here developed.

2.1 Parallelization methods

In order to increase the speed of our program, it has been decided to parallelize it on a set of clusters of multi-core machines, in the more complex possibilty. But there are many ways to parallelize our algorithm and we have to choose how we want to do it. in a first time, let's talk about the parallelization method which have the most potential, then, we will see which of them can be great for our cases.

2.1.1 Previous Work

In the last report, we talked about the different methods, their advantages and their drawbacks. There are mainly two parallelization methods that are efficient in the conditions of our project, in terms of speed.

The first one is called the Root Parallelization. It consists in giving the tree to develop to every thread, let them develop it randomly without any communication with the environment during a certain amount of time and then, merge the results of each tree. This method has the great benefit of minimizing the communication between the actors (in this case, the threads). They only communicate at the beginning and at the end of the algorithm, without needing any further synchronization. The Root Parallelization is depicted in figure 1.

The other efficient parallelization method is called UCT-Treesplit and is depicted if figure 2. It looks like Root Parallelization as it gives to each actor the same tree to develop. But contrary to Root Parallelization, when the tree is developed on a certain node, it goes on working packages [2] who are distributed among every actor. The repartition of the packages are made by a scheduler. ~~In terms of performance, this method is very efficient but needs an High-Performance Computer,~~ or *HPC*, and is very sensitive to network latency.

We have to choose two parallelization methods, one for the cluster parallelization and another for the shared memory parallelization.

2.1.2 Cluster Parallelization

For the cluster parallelization, we have to take into account the fact that communication will be done by sending messages, that are relatively costly in terms of performance. Moreover, as the network can have latency, it is best to minimize the communication between the

[2]A working package is a set of nodes that can be possibly developped

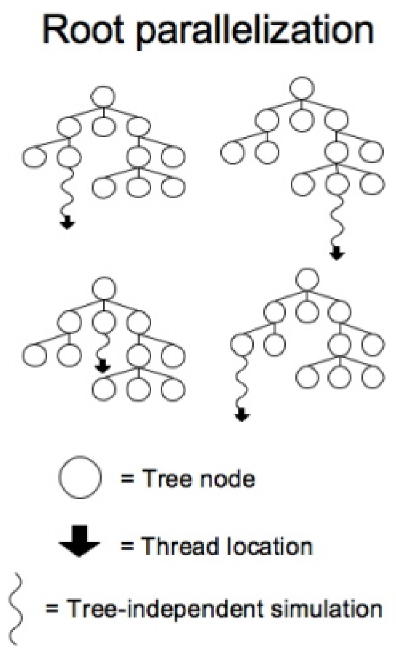


Figure 1: Overview of Root Parallelization

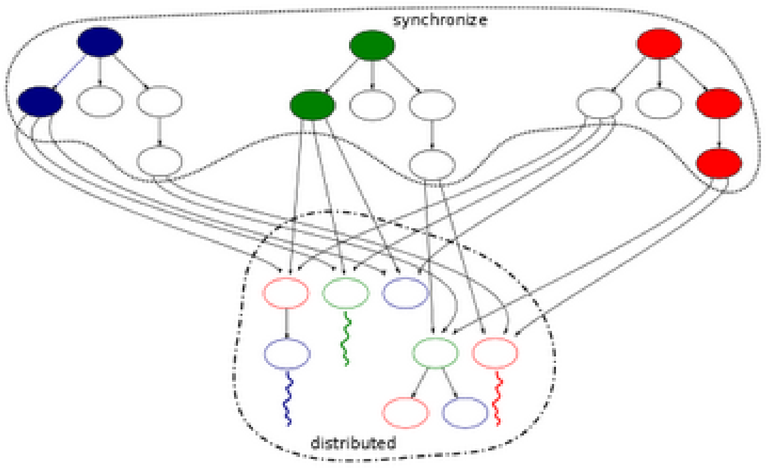


Figure 2: Overview of UCT-Treesplit Algorithm

computers and that's why Root Parallelization has been chosen. It reduces the cost in communication at maximum, is very simple to implement, does not depend on the configuration of each computer and is very efficient.

2.1.3 Shared Memory Parallelization

For the shared memory parallelization we could choose both Root Parallelization or UCT-Treesplit. If we choose UCT-Treesplit, it may be difficult to implement it correctly since it is a very complex strategy and, moreover, it would make the algorithm very sensitive to network issues. That is why we chose to implement another Root Parallelization. This way, the global strategy of our program will be simple and homogeneous.

2.2 Monte Carlo Tree Search algorithm

One of the advantages of the MCTS algorithm is its genericity. In fact it can be applied to a lot of games. In order to keep this genericity, we thought about two methods.

The first one is General Game Playing[3], a formal language which allows programs to create a model given the game formal rules. However, the problem of this approach is that even if it makes it possible to play unknown games, it prevents the use of efficient heuristics to improve the algorithm. That is why we chose ~~a~~ another way which consists in exploiting the properties of the MCTS algorithm. With this solution, the algorithm does not need to know the rules, only the moves. With this in mind we created an interface for the game which defines the methods that the MCTS algorithm requires. In other words our algorithm is compatible with all the two-players games implemented with this interface.

Only the following fonctions are required :

- return all the possible moves given position
- play a random move
- play a chosen move
- play random moves until the end of the game
- return whether the game is not finished or who won

The main game of our algorithm is Arimaa. Therefore we will be able to specialize our algorithm for it in order to improve its efficiency. The main problem is the branching factor[3] of the Arimaa game which average is 17 281 and reaches about 22 000 after 10 moves[4].

Game	Average number of possible moves
Othello	8
Chess	35
Game of Go	250
Arimaa	17 281

The branching factor of a game is important because it increases greatly the space that has to be searched to guess what will happend multiple moves ahead. In chess after 4 moves, the number of positions evaluated are about 35^4 which is roughtly equivalent to 1,8 billion. In Arimaa, after 3 turns (yours, the opponent and yours again), if you were to explore all

[3]In a tree, the branching factor is the number of children at each node.

positions, you would need to evaluate around 5,2 trillions[4] boards. It is about 2000 times more than *Chess* with half the number of moves).

In order to decrease the space to be search, our MCTS Algorithm will perform a big number of simulations before chosing the nodes to explore. After the selection, it will prune the tree in order to optimise search speed and the memory management. The following example has been simplified :

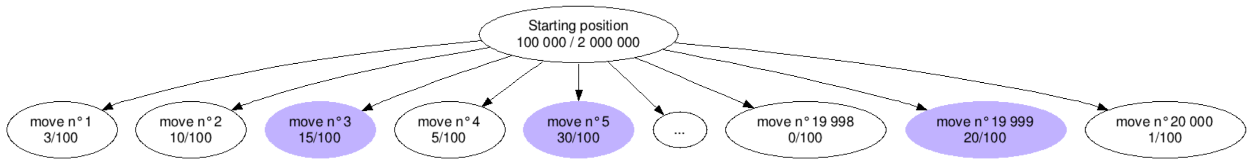


Figure 3: Run random simulations on each childs and select the ones with the highest winrate (Monte Carlo algorithm).

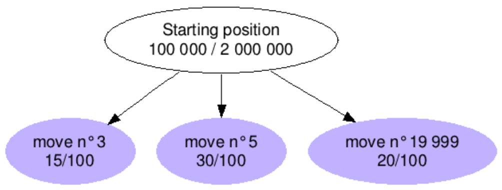


Figure 4: Prune the unselected children.

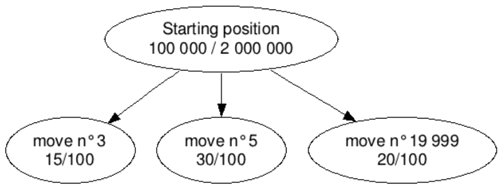


Figure 5: Select the node to expand next (here Move n°5) using the UCT function.

[4]1 trillion in short scale = 1 thousand billion = 10^{12}.

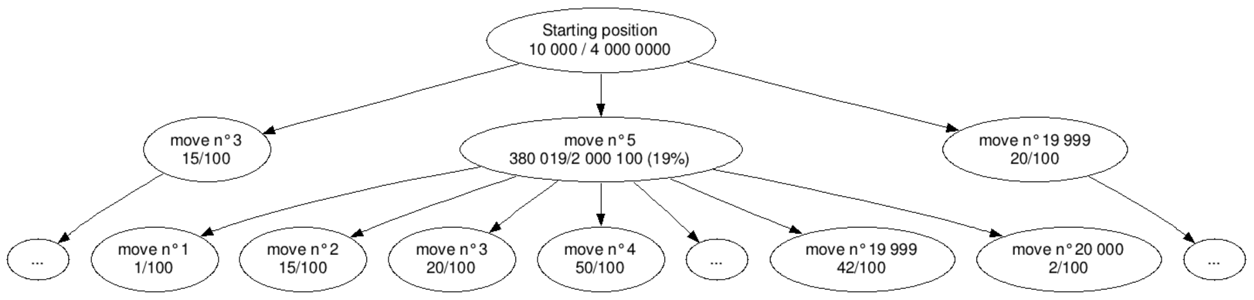


Figure 6: Run random simulations on each children and feed back the results to the parents.

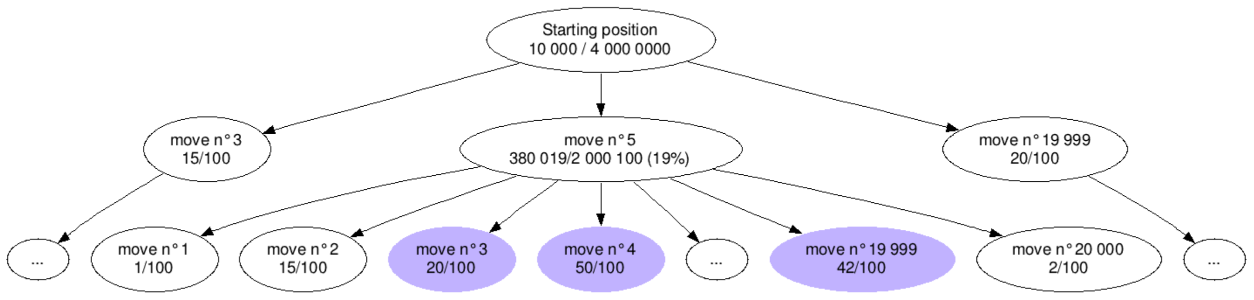


Figure 7: Select the nodes with the highest winrate.

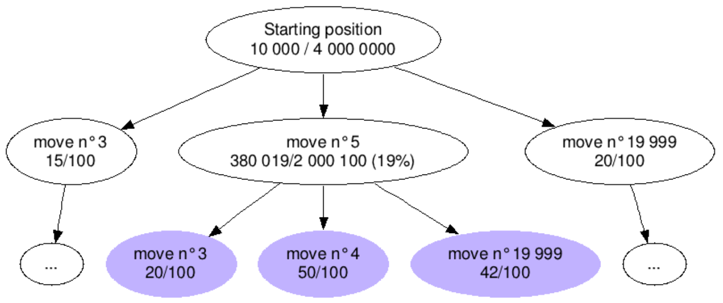


Figure 8: Prune unselected children.

Iterate the process until the time limit dedicated to the search is reached and return the move with the best results.

3 General Architecture

3.1 Game behaviour

This part is about the interaction between the artificial intelligence (*AI*) and the other modules. An application will be developed in ~~ordrer~~ to test the AI against human players as well as other AIs. In order to make the game easy to play, this application will provide a graphical User Interface (further referred to as *UI*), described in part 3.3. This UI, as well as the algorithm for the AI, will act upon the game model, so as to inform it of what moves were made. The UI will also regularly update to give the player feedback on the progression of the game (for more details, see part 3.2).

If at some point our AI is to be tested against an AI created by others, an API will be needed in order to interface it with the game model, as described in part 3.4.

3.2 Overview of the architecture

The organization of the architecture is described in Figure 9.

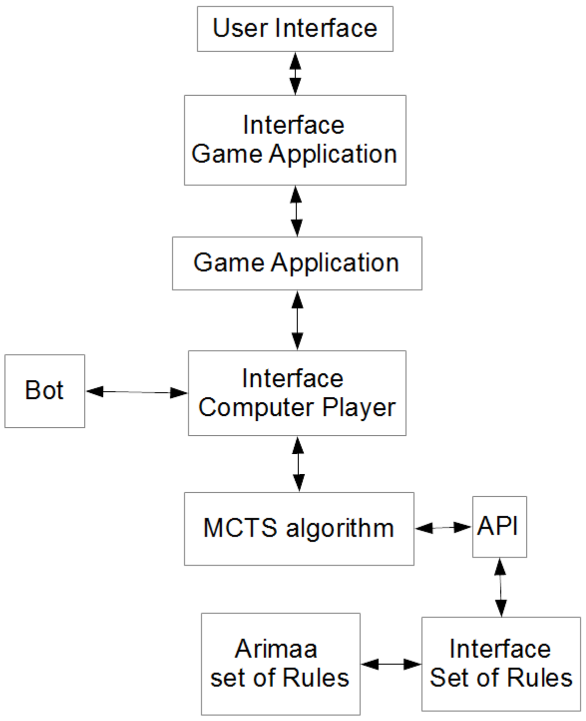


Figure 9: General view of the architecture

The user will interact with the user interface, which will communicate with the game application, where all data for the game will be stored. Then, it will communicate with an Interface which will handle the communications between the Game application and the computer player.

In our case, the computer player will be played by the AI using the MCTS algorithm, communicating with the Arimaa set of rules via the API and an interface for the set of Rules. If another algorithm is to be tested, it will be added.

Interfaces are very important because they make it possible for others to use our project for others games than Arimaa without rebuilding everything. They would only have to implement the Interface with their proper game, game rules.

Furthermore, if the time permits it, the management of computers bots[5] will be added. The only problem here is the computers bots are not similar to each other, thus it will take time to implement a class which will be able to communicate with both the computer bot and the interface.

The input of our software will be provided by the user using the mouse and keyboard. The output will only be the display of the screen, after the computer player makes a move.

3.3 User Interface

In order to test our program against human players, we will develop an application implementing a graphical user interface (*GUI* or also *UI*). Figure 10 represents the interactions between the player and the UI. It will interface directly with the model, and will be controllable by the human with the mouse or keyboard, in a manner as intuitive as possible. Thanks to this UI, the user will be able to choose between a one-player mode, a ~~two-placer~~ mode and a demonstration mode. This last mode consists in a match between two AIs, and will be helpful to compare different AIs. The user will also be able to choose and configure the AIs, if we have time to develop more than one.

[5]For instance of the site *Arimaa.com*

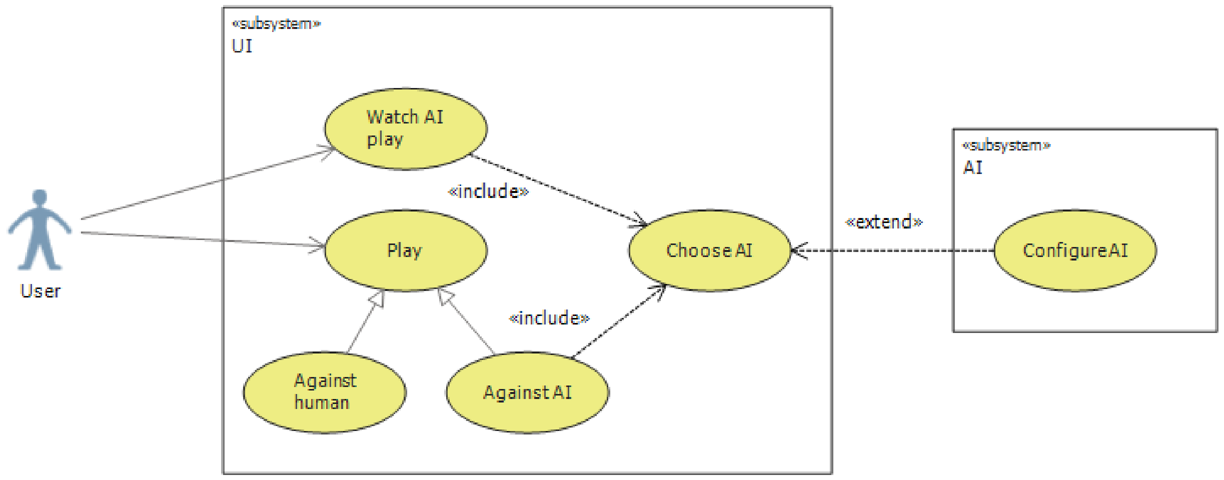
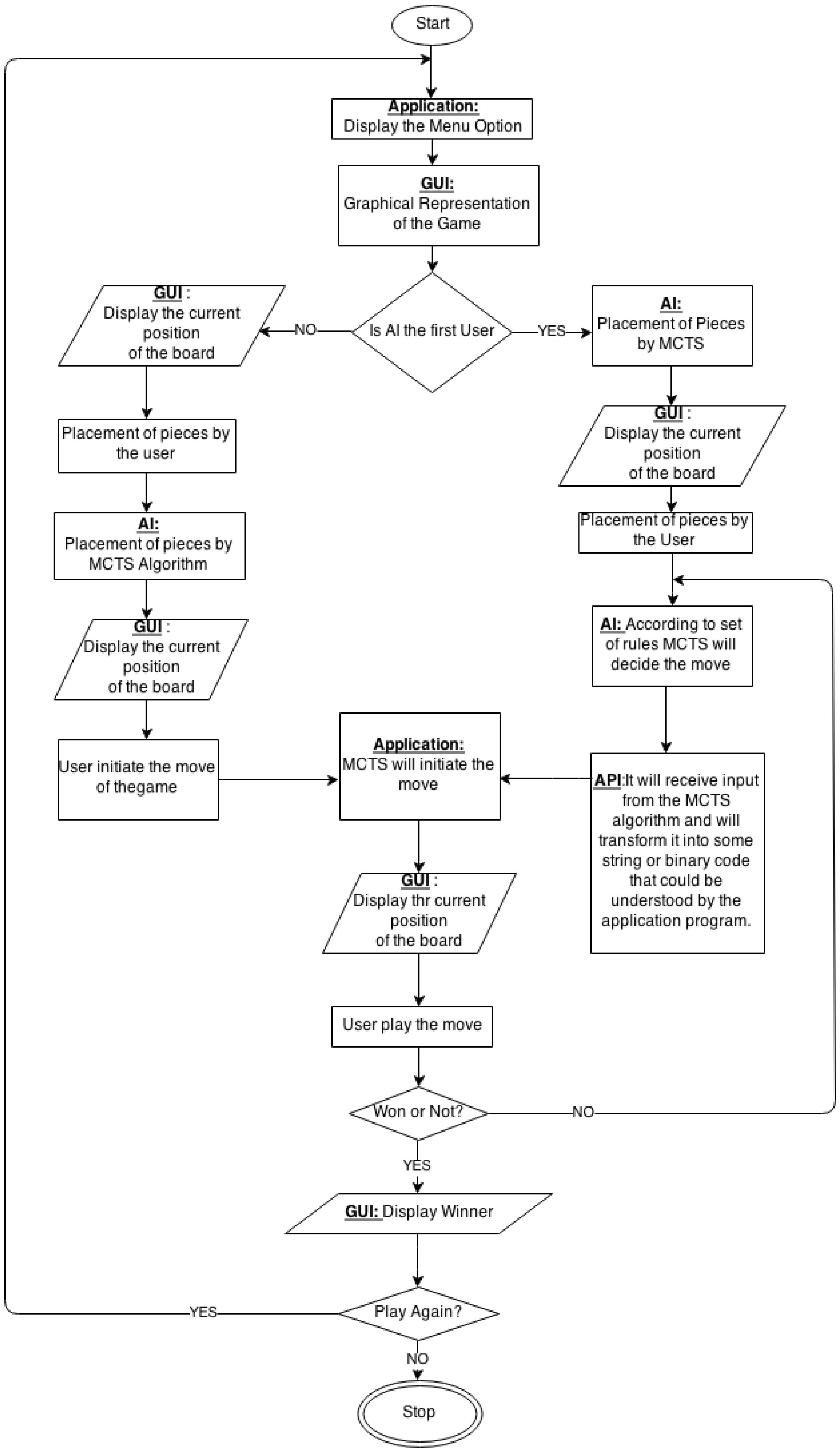


Figure 10: The user-case diagram describing the interactions made possible by the UI

3.4 Application Program Interface

API stands for Application Programming Interface. An API is a set of routines, protocols, and tools for building software applications. The API specifies how software components should interact and are used when programming graphical User Interface (GUI) components. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

In this project the API will be induced between the MCTS algorithm i.e. AI and the board game application. It will be responsible for providing the input to the board game application in understandable format and will receive the input from the MCTS algorithm. Its usage is described in figure 11.



Start
Application: Display the Menu Option
GUI: Graphical Representation of the Game
Is AI the first User
NO
YES
GUI : Display the current position of the board
Placement of pieces by the user
AI: Placement of pieces by MCTS Algorithm
GUI : Display the current position of the board
User initiate the move of thegame
AI: Placement of Pieces by MCTS
GUI : Display the current position of the board
Placement of pieces by the User
AI: According to set of rules MCTS will decide the move
API:It will receive input from the MCTS algorithm and will transform it into some string or binary code that could be understood by the application program.
Application: MCTS will initiate the move
GUI : Display thr current position of the board
User play the move
Won or Not?
NO
YES
GUI: Display Winner
Play Again?
YES
NO
Stop

4 Software solutions

4.1 CPU Parallelization

To parallelize our algorithm on one machine, we need to chose a framework. Four solutions have been found and compared :

- MPI
- OpenMp
- C++11 Threads (Boost)
- Pthreads (C)

4.1.1 MPI

MPI is a library mainly dedicated to parallelization between different machines, it could work on a single CPU but as we said in the previous report, it doesn't provide as many possibilites as its counterparts. Therefore we do not choose to do it for that part of the implementation.

4.1.2 Pthreads

Pthreads are out of order because they are depreciated : it is a C language library and the C++11 language provides more efficiency and possibilities. The threads management is heavy to code and requires a good knowledge of how threads work precisely.

4.1.3 OpenMP and C++11 language

OpenMP and C++11 language are the remaining options. C++11 language is a simplification of the Boost library inducing the latest to be more complete. The following table[5] summarizes each libraries pros and cons :

OpenMP	C++11 Threads	Pthreads (C)
+ Options	+ Flexibility	+ Flexibility
+ Portable	+ Type-Safety	+ Low-level
+ Languages	+ Possibilities	+ Compatibility
- Performances	- Fortran	- Efforts
- Memory	- Compiler	- Type-safety
- Unreliable	- Scalling	- Management

Both libraries provide similar performances[2]. However OpenMP is easier to use (precompilier declarations...) and keeps the code clean[1]. C++11 language allows a better thread management. Nevertheless it can easily also fall behind his counterpart in term of speed if some mistakes are made.

Considering that OpenMP is safer to use and get similar results to C++11 language, we chose OpenMP to parallelize our algorithm on CPU[6].

4.2 GPU Parallelization

During our research for GPU[7] parallelization, three possibilies have been found:

- OpenMp
- OpenACC
- CUDA

4.2.1 OpenMP

OpenMP supports the GPU programming since its version 4.0, it implements some of the methods of OpenACC. But it seems to be less efficient and complete than OpenAcc so we decided to avoid to use it.

4.2.2 CUDA

CUDA is a framework develop by NVIDIA dedicated to the use of GPU for complex calculation. It allows very efficient and low-level computations but it forces to rewrite all the parallel code. Its means that we will need one code for machines without GPU and one for machines with GPU. Furthermore, the product code is illegible for people who do not master CUDA.

4.2.3 OpenACC

OpenACC is an API created by the authors of OpenMP to implement the GPU computation before to integrate it in OpenMP. A part of its fonctionnalities had been added in OpenMP 4.0 but OpenACC is still more complete and efficient, especially for the data management. This point is very important, because if it is not correctly managed, there will be too many transfer of data between the CPU and GPU. In this case we can lose more time than we gained on the calculation.

As we consider OpenACC more simple, and easier to maintain for just a little drop of performance compared to CUDA, we chose it to design our software to take advantage of the presence of GPU.

[6]Central Processing Unit
[7]Graphic Processing Unit

4.3 Cluster Parallelization

Our last software need is about the cluster parallelization. As we have decided to use a Root Parallelization there will not be a lot of communication between different computers, we could choose between two solutions : the Sockets and MPI.

- A socket is used to communicate across a computer network. The socket is an end point of the communication flow. A socket is a low-level mechanism.
- MPI is a standardized message-passing system. It allows us to communicate between computers which belong to a network by sending messages between them.

4.3.1 The chosen solution

We decided to use MPI for many reasons.

- MPI is more high-level than the sockets. So, it will be simpler to implement in our software.
- The community behind MPI is large so there wouldn't be any problem to fix the different bugs. Moreover, the MPI documentation is clearer than the sockets' one.
- The operation of MPI is based on the sockets so it is similar, though a little bit inferior, to the sockets, in term of performances.

In conclusion, MPI would be simpler to implement, more documented than the sockets while they both have almost similar performance.

4.4 Actor Model

In the parts just before, three solutions are presented which permit using together to parallelize our algorithm. In this part will be quickly introduced another way to do this, the actor model.

In this model, each machine, each core and each GPU is seen as an actor. The actors can send messages amongst themselves. When they receive one, they can:

- send messages to other actors
- create new actors
- modifie their behavior

This actions can be realize in parallel. Moreover the system of message is totally asynchronus which is adapted to the root parallelization we will use at the beginning. But it will not be too interested for other parallelization methods. For the moment we have not found yet a framework implementing the actor model that we could use for our project but we are still investigating because it could be a very efficient solution.

5 Conclusion

The software specifications have been decided: further to the AI, a user interface (*UI*) will be developed. It will include every version of the AI that will have shown satisfying results. Therefore, this project will be composed of three parts : the UI, the AI and the game model that will handle the rules.

The AI will use the *Monte Carle Tree Search* algorithm. In order to make it more efficient, it will implement parallelization on threads and on multiple machines. Ultimately, the goal is to try and run our algorithm on *Grid 5000*, a set of clusters of multi-core machines. If the computers at our disposal have GPUs, they will also be used. This parallelisation will be performed using OpenMP and OpenACC on every machine (OpenMP for thread parallelization and OpenAcc for GPU parallelization), and MPI between machines. As for parallelization strategies, we will first use *Root parallelization*, and then try other ones if there's enough time to do so.

The next step will be about defining the details of the implementation, before developement starts.

References

[1] Andrew Binstock. intel developer zone, choosing between openmp* and explicit threading methods. `https://software.intel.com/en-us/articles/choosing-between-openmp-and-explicit-threading-methods`, 2010.

[2] Paul Petersen Bob Kuhn and Eamonn O'Toole. Openmp versus threading in c/c++. `http://www.cs.colostate.edu/~cs675/OpenMPvsThreads.pdf`.

[3] Stanford Logic Group. `http://logic.stanford.edu/classes/cs227/2013/readings/gdl_spec.pdf`, 2008.

[4] Brian "Janzert" Haskin. A look at the arimaa branching factor. `http://arimaa.janzert.com/bf_study/`, 2006.

[5] Florian R. intel developer zone, choosing the right threading framework. `https://software.intel.com/en-us/articles/choosing-the-right-threading-framework`, 2013.