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*. It is about the specifications of the project. This report focuses on "what" questions, such as what the algorithm will do, or what the application will do. It will describe each and every component of the project: the game, the general architecture, the Artificial Intelligence implementing the MCTS Algorithm, the Converter, the User Interface and the application Model. A detailed focus is also made on the parallelisation techniques and the frameworks that might be used.

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 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. The algorithm will do the same by building a search tree containing the different posibilities. The Minimax algorithm does it by exploring all possibilities, which is heavy. The MCTS algorithm is lighter and converges to the Minimax algorithm, therefore it has been chosen for this project.

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 will 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 four different modules 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. But there are many ways to parallelize our algorithm and we have to choose how we want to do it. The parallelization method which have the most potential will be developed first, then we will see which of them best apply to the project.

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.

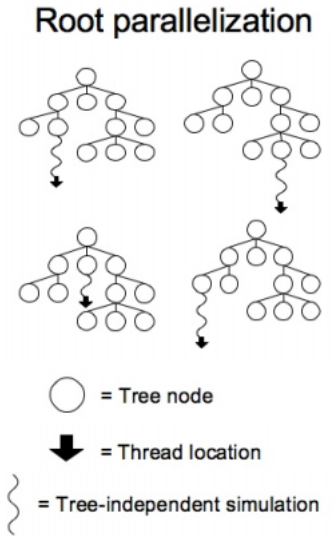

Figure 1: Overview of *Root Parallelization* [3]

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 to be optimized, or *HPC*, and is very sensitive to network latency.

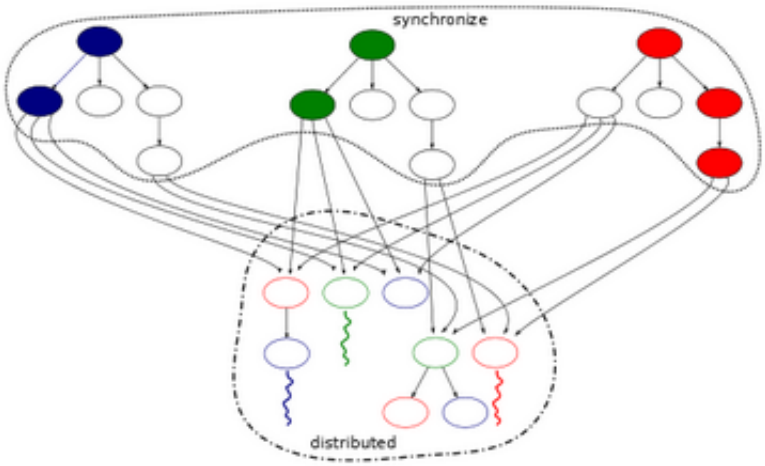

Figure 2: Overview of *UCT-Treesplit Algorithm* [7]

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

2.1.2 Cluster Parallelism

For the cluster parallelism, 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 computers. That is why *Root Parallelization* has been chosen over *UCT-Treesplit*. *Root Parallelization* reduces the cost in communication at maximum, and is very simple to implement. It does not depend on the configuration of each computer and is very efficient[3].

2.1.3 Shared Memory Parallelism

For the shared memory parallelism, we can choose between many methods but, as seen previously, *Root Parallelization* is the better one.[3] That is why this method will also be chosen for the shared memory parallelism. This way, the global strategy of our program will be simple, efficient and homogeneous.

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

2.2 *Monte Carlo Tree Search algorithm*

2.2.1 Genericity of MCTS 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[4], 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 from improving the algorithm. That is why we chose 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

2.2.2 Branching factor

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 possible moves and reaches about 22 000 possible moves after 10 moves[5].

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 explored 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 positions, you would need to evaluate around 5.2 trillion[4] boards. It is about 2000 times more than *Chess* with half the number of moves).

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

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

2.2.3 Decreasing the impact of the Branching factor

In order to decrease the space to be explored, our MCTS algorithm will perform a high 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:

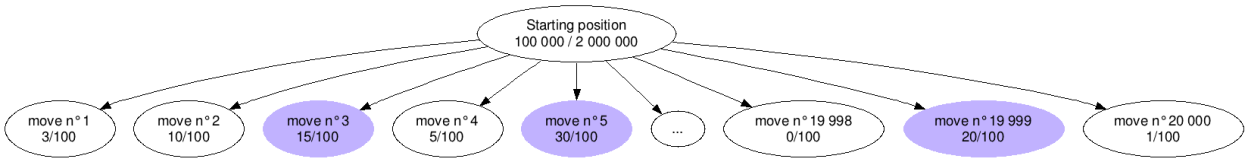

Figure 3: Run random simulations on each children and select the one with the highest winrate (*Monte Carlo algorithm*).

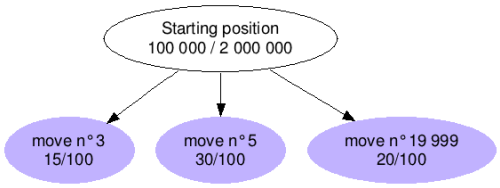

Figure 4: Prune the unselected children.

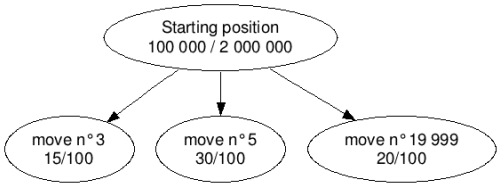

Figure 5: Select the node to expand next (here move n°5) using the UCT function.

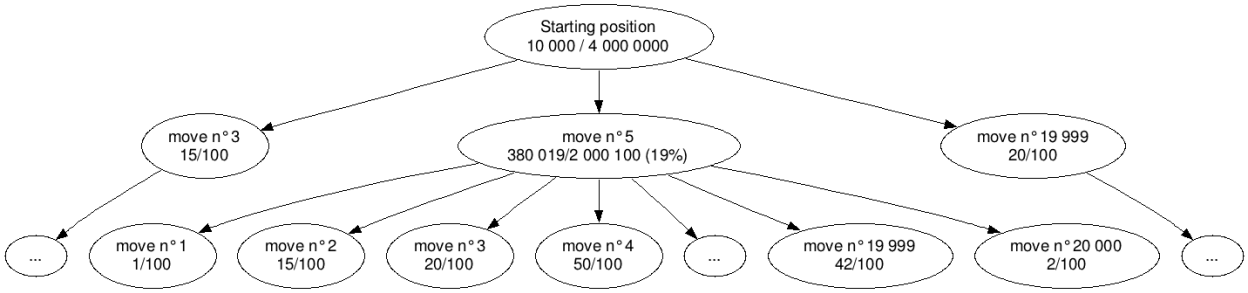

Figure 6: Run random simulations on each children and feed back the results to the parents.

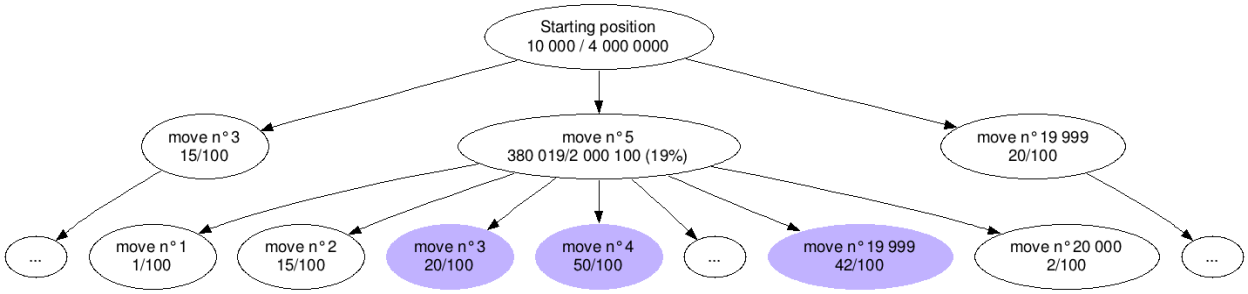

Figure 7: Select the nodes with the highest winrate.

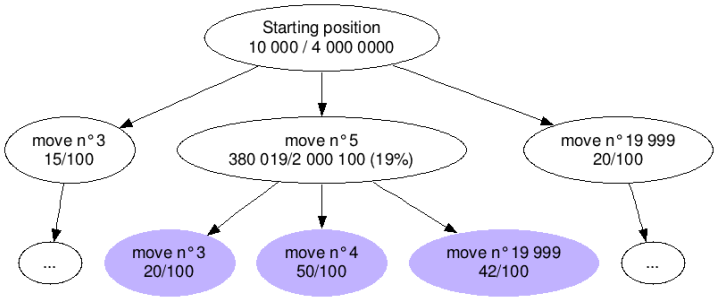

Figure 8: Prune unselected children.

Iterate the process until the limited time dedicated to the search is reached, and return the move with the highest winrate.

3 General Architecture

3.1 Introduction

An application will be developed in order to test the AI[5] against human players as well as other AIs. In our project, the AI is handled by the MCTS algorithm, crossing trees and doing simulations. We will see more on its implementation on part 3.5. Our application will integrate four modules:

- the Artificial Intelligence
- the game model containing the rules
- the User Interface (UI) described in part 3.3
- an API described in part 3.4 that will be used to integrate other AIs

In this part, the different interactions between these modules will be discussed. The 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 Context of the architecture

The Figure 9 presents how the application works.

[5]Artificial Intelligence

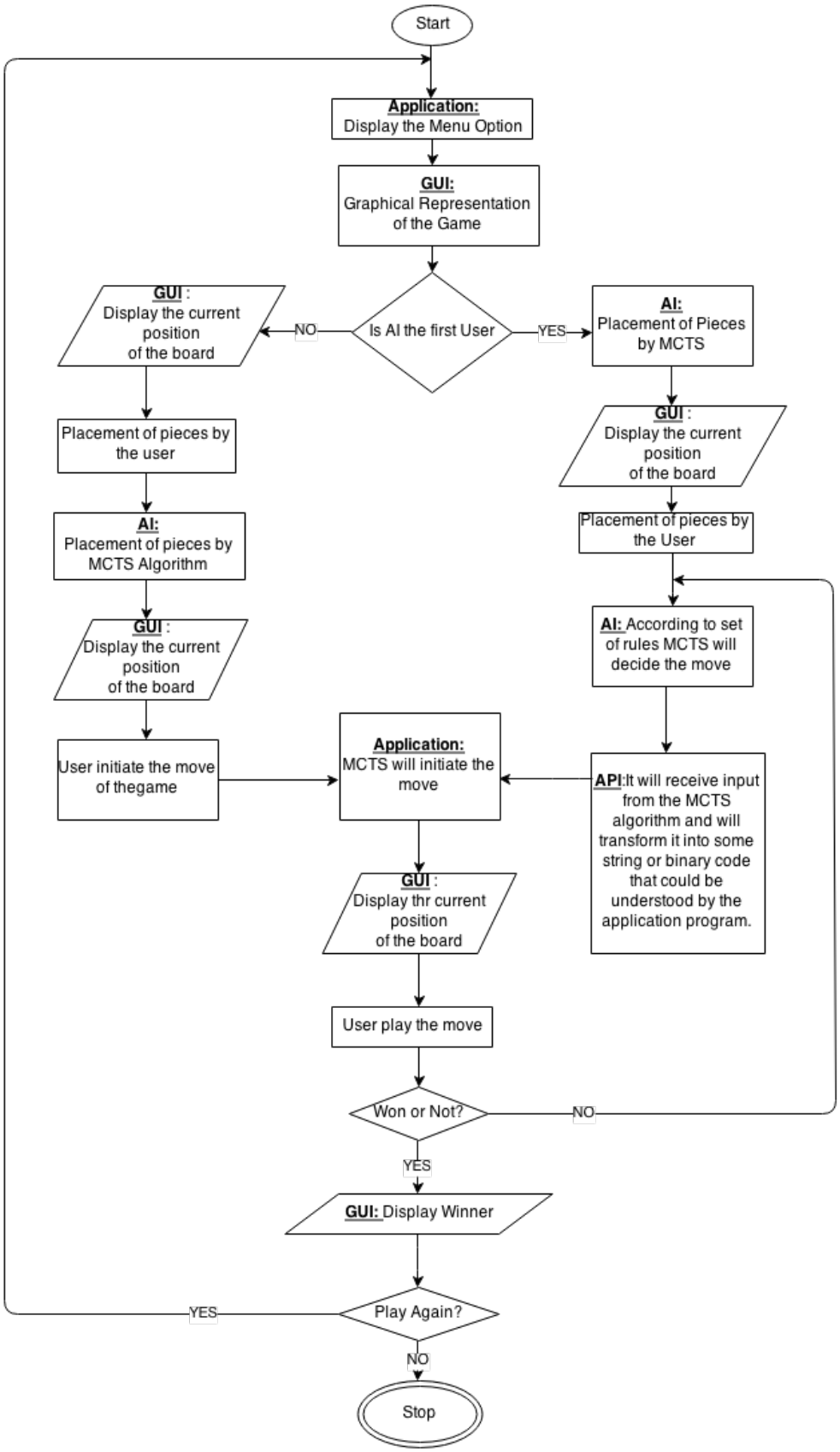

Figure 9: The flowchart describing the general usage of the application

First, the main menu displays. Then, the players place their pieces, and they play. It is possible to see which module interacts with each other. To go deeper, the four modules will be presented in Figure 10.

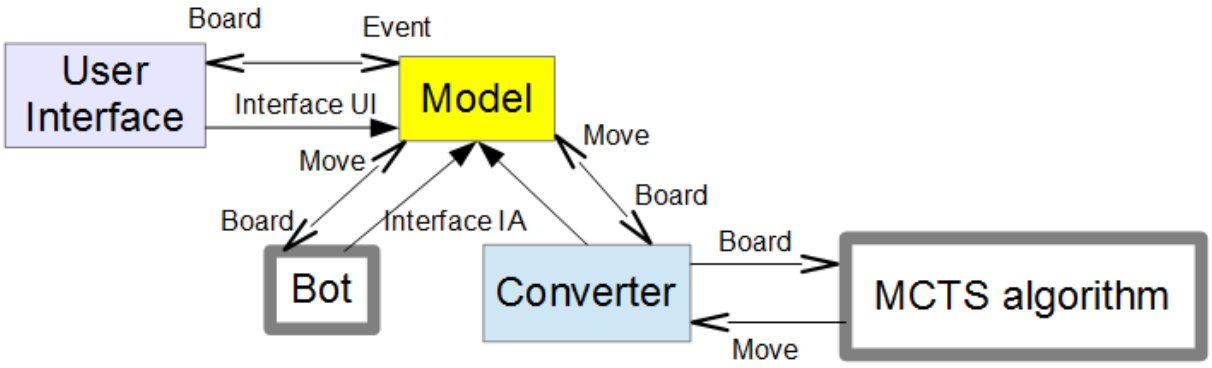

Figure 10: General view of the architecture

The 4 modules and the Bot will be described:

- **User Interface module**

  This module will communicate with the environnement, displaying the board, creating events, like mouse clics and keyboard events. Then it will send them to the Model module. The User Interface is studied more deeply in part 3.3. An interface is used to make it possible for the future user of our project to replace this User Interface by his own, just by implementing this interface.

- **Model module**

  This module will handle the game itself. It will be able to store and play games, receiving moves and giving the state of the board to the User Interface and the AI. A type Move will be created, to impose the move format. The Model module contains as well the rules of the game. It is the link between the user and the AI.

- **Converter module**

  This module is the link between the Model module and our AI, the MCTS algorithm. It forwards the board sent by the Model module to the MCTS algorithm, adapting the format. Furthermore, it does the same communication in the other way with the final move, from the MCTS algorithm to the Model module with the adapted format. The Converter module is described in part 3.4.

- **The MCTS Algorithm module**

  This module is the Artificial Intelligence of our project. Giving a board by the Converter module, it is able to send back the move. The MCTS algorithm module is defined in the part 3.5.

- **Bot**

  A Bot is an Artificial Intelligence as well. Our work will make possible the adding of other AI, or Bots, conceived by anyone, just by implementing the AI interface. For instance, there are some free bots already available on the site *Arimaa.com*.

3.3 User Interface module

In order to test our program against human players, we will develop an application implementing a graphical user interface (*GUI* or also *UI*). Figure 11 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-player 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.

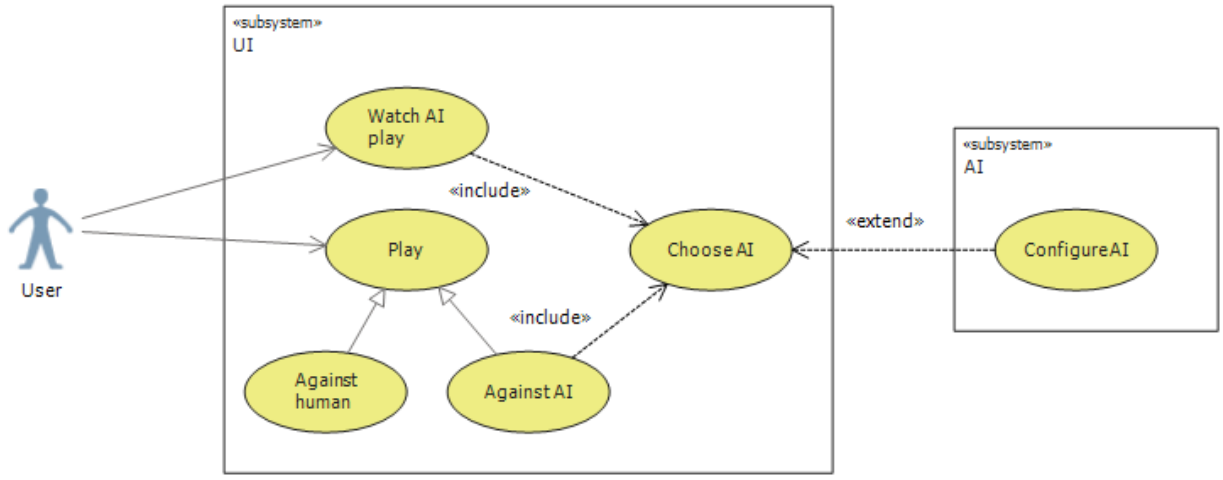

Figure 11: The user-case diagram describing the interactions made possible by the UI

3.4 Converter module

This part is responsible for doing the interpretation between the two different modules i.e. the Model module and the MCTS Algorithm module, our AI . The Model and the AI are different modules. The Figure 12 shows the converter relations.

Figure 12: Block Diagram of Converter

The output of AI will be utilised by the Model in order to make a move on the board. It is possible that the output generated by the AI would be in the format that is not understandable by the Model. So there is a converter required to solve the purpose.

The converter will receive input from the MCTS algorithm module in some format such as tree. Then, it converts it into some other format that could be understood by the model such as a string or a binary code.

3.5 The MCTS algorithm module

The next Figure 13 describes the AI, i.e. the MCTS Algorithm module.

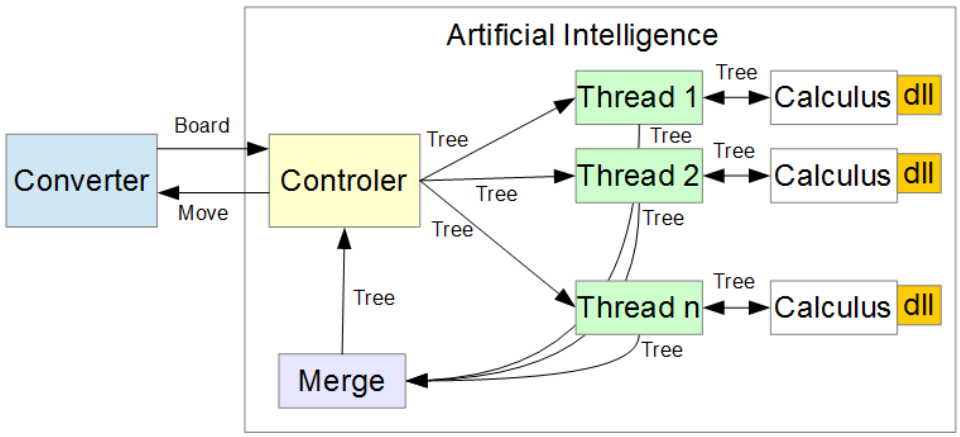

Figure 13: Block Diagram of Converter

The converter sends the state of the board to the controler, which creates trees in different threads. Each of these threads computes his tree, getting the possible moves thanks to a set of rules, gathered together in a dynamic link library dll.

Then, when the calculus is over, the thread hands over the tree to the Merge module. When all threads have done it, the module merges the trees, gathering data in an only tree, that is sent to the controler.

Finally, the controler is able to decide what move to choose and it sent this move to the converter.

4 Software solutions

4.1 Shared memory Parallelism on CPU

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

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

4.1.1 MPI

MPI is a library mainly dedicated to parallelization between different machines. It could work on a single CPU[6] but as we said in the previous report, it does not provide as many possibilites as its counterparts. Therefore we did not choose to use it for that part of the implementation.

4.1.2 Pthreads

Pthreads are out of the question 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[6] 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

[6] Central Processing Unit

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

Considering that OpenMP is safer to use and provides similar results to C++11 language, it is the best of present choices to parallelize our algorithm on CPU.

4.2 Shared memory Parallelism on GPU

During our research for GPU[7] parallelization, three possibilities 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 using 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 of the parallelized code. Its means that we will need one version of the code for machines without GPU and one for machines with GPU. Furthermore, the specific code is unreadable 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 integrating it in OpenMP. A part of its fonctionnalities have been added in OpenMP 4.0 but OpenACC is still more complete and efficient, especially for data management. This point is very important, because if it is not correctly covered, there will be too much data to transfer between the CPU and the GPU. In this case we would lose more time than we would gain on the calculation.

[7]Graphic Processing Unit

As OpenACC is more simple, and easier to maintain for just a little drop of performance compared to CUDA, it seems be the most adapted of this solutions to design our software to take advantage of the presence of GPU.

4.3 Cluster Parallelism

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

- A socket is a low-level mechanism used to communicate across a computer network. The socket is an end point of the communication flow.
- 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 the following reasons:

- MPI is more high-level than the sockets, thus it will be simpler to implement in our software.
- The community behind MPI is large so there would not be any problem to fix the different bugs. Moreover, the MPI documentation is clearer than the sockets'.
- MPI uses sockets so it is similar, though a little bit inferior to the sockets, in term of performance.

In conclusion, MPI would be simpler to implement, is more documented than the sockets while they both have almost similar performance. So, it seems more adapted to our project than the sockets.

4.4 Actor Model

From parts 4.1 to 4.3, three solutions allowing the parallelization of our algorithm have been presented. In this part, the actor model, another way to handle parallelization, will be introduced.

In this model, every machine, core or GPU is seen as an actor. These actors can send messages to each other, using adresses. When they receive one, they will be able to:

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

These actions can be realized in parallel. Moreover, the message system is totally asynchronous, which is adapted to the *Root Parallelization* we will use at first. However, it will not be as interesting as other parallelization methods.

In the next report, we may present a framework implementing the actor model suitable for our project.

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 *Monte Carlo Tree Search AI* that will show satisfying results. In the case we test them against other AIs, these AIs will be included as well. Therefore, this project will be composed of four modules: the UI, the MCTS algorithm, the Model, and the Converter that will help integrate other AIs.

The AIs that will be developed will use the *Monte Carle Tree Search* algorithm. In order to make it more efficient, they will implement parallelization on threads and on multiple 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 is enough time to do so.

The next step will be about defining the details of the implementation, before the 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] Guillaume Chaslot, Mark Winands, and H. Jaap van den Herik. *Paralel Monte Carlo Tree Search*. PhD thesis.

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

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

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

[7] Tobias Graf, Ulf Lorenz, Marco Platzner, and Lars Schaefers. *Parallel Monte-Carlo Tree Search for HPC Systems*. PhD thesis.