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 is the second report related to the incremental development of the project "Fast and Furious Game Playing: Monte Carlo Drift". This report is related to the specification and ~~requirement~~ of the project. A detailed study on specification and requirement analysis is done, keeping in mind, all the aspects of the project. This report is a step ahead of the previous report regarding user interface, application programming interface, implementation of the MCTS etc. A detailed focus is also made on the parallelisation techniques using OpenAcc and OpenMP.

Contents

1 Introduction

In 1997, Deep Blue, a supercomputer built by IBM, won a six games match against Garry Kasparov, the current world chess champion. Humans ~~got~~ beaten in Chess, but remain undefeated in other games. Since then, researchers have been developing improvements in artificial intelligence.

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 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 at the infinite.

By exploring the numerous random possibilities at any one time, it 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.

Different parallelization methods will be studied to choose the most suitable to this project. The exploration of the tree will depend on the parallelization method. We have chosen the root parallelization method, and the UCT Tree split. 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 general architecture will be described, the behaviour and the API of our game. Then, the parallelization method chosen will be explained, and the MCTS algorithm will be described Finally, we will discuss the different software with OpenMP, OpenACC, and MPI.

For us, the most interesting part regarding this project is the creation of an artificial intelligence which is as optimized as possible.

[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 General Architecture

2.1 Game behaviour

In order to test the AI, an application will be developped. This application will include a two-player mode, a one-player mode and a demonstration mode (AI versus AI). In the case where several AI have been implemented, the user will be able to choose which one to play against.

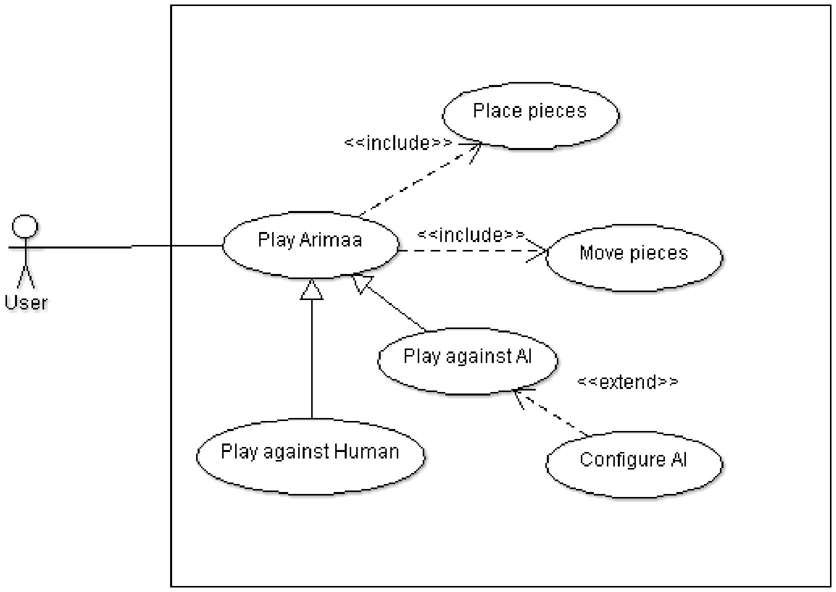


Figure 1: The user-case diagram describing the application

In order to make the game easy to play, this application will provide a graphical uer interface (further referred to as *UI*). 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.

2.2 Application Program Interface

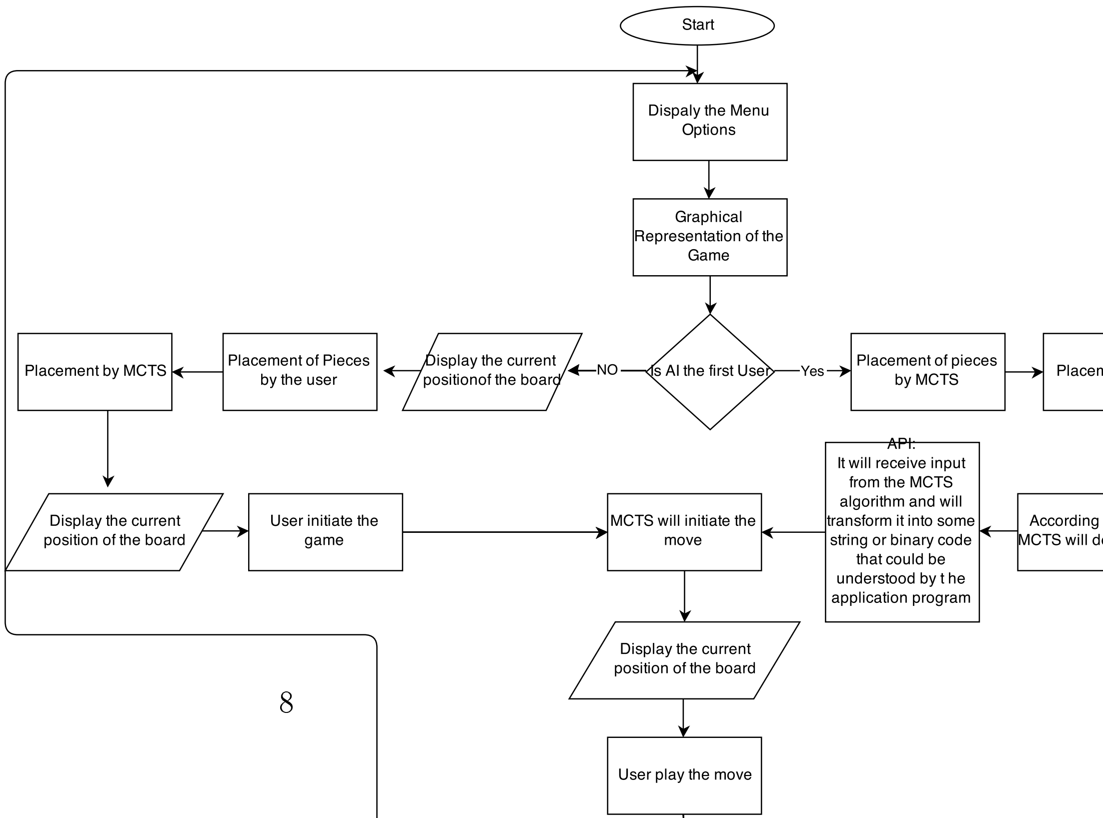
API stands for Application Programming Interface. API, is a set of routines, protocols, and tools for building software applications. The API specifies how software components should interact and are used when programming graphical user interface (GUI) components. A good API makes it easier to develop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

In this project the API will be induced in between the MCTS and the board game application and will be responsible for providing the input to the board game application in understandable format and will receive the input from MCTS .The following diagram will help in understanding the things better.

The input of our software will be the action of the users, his mouse clics, moves with keyboard. The Output will only be the display of the screen, after a move of the opponent in 1VSComputer or nothing in 1VS1.

To test our game, an interface will be implemented multiple times for diferent way of playing. This interface will define a serie of unique moves that will be used by all implementations to play. A first implementation has been done in the first application game, to make play two players against each other. The second implementation will be about 1VS-Computer, linked with the MCTS algorithm. If another algorithm is willing to be tested, it will be added.

Furthermore, if the time permits it, the management of computers bots (of the site arimaa.com) will be added. The only problem here is the computers bots are not similar to each others, thus it will take time to implement a class which will be able to communicate with both the computer bot and the interface.


Start
Dispaly the Menu Options
Graphical Representation of the Game
Is AI the first User
Yes
NO
Placement of pieces by MCTS
Placen
Display the current positionof the board
Placement of Pieces by the user
Placement by MCTS
API:
It will receive input from the MCTS algorithm and will transform it into some string or binary code that could be understood by t he application program
According
MCTS will de
Display the current position of the board
User initiate the game
MCTS will initiate the move
Display the current position of the board
User play the move

3 Algorithmic methods

3.1 Parallelization methods

In order to increase the speed of our program, we have 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.

3.1.1 Previous Work

In the last report, we talked about the different methods, their advantages and their drawbacks. We have seen that there are mainly two parallelization methods that are efficient.

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 3.

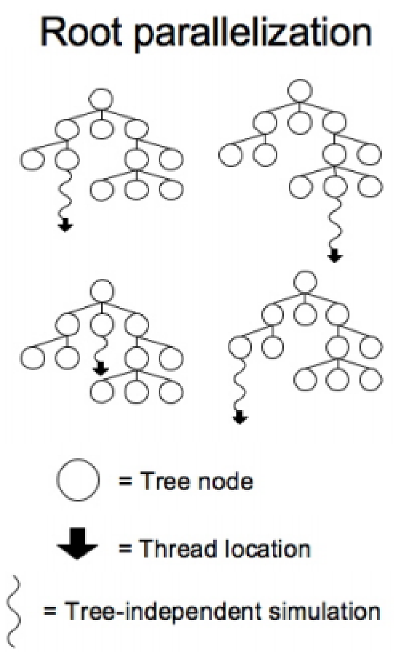


Figure 3: Overview of Root Parallelization

The other efficient parallelization method is called UCT-Treesplit and is depicted if figure 4. It looks like Root Parallelization as we give 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 who are distributed among every actor. In terms of performance, this method is very efficient but needs an High-Performance Computer, or *HPC*, and is very sensitive to network latency.

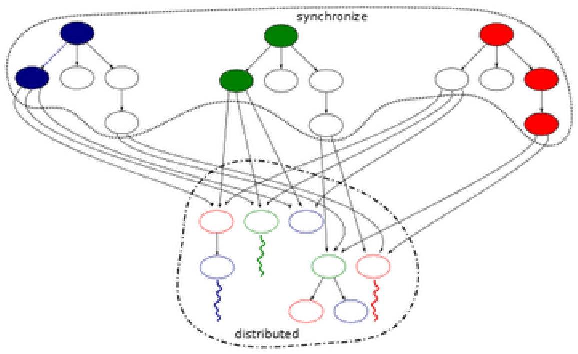


Figure 4: Overview of UCT-Treesplit Algorithm

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

3.1.2 Cluster Parallelization

For the cluster parallelization, we have to take into account the fact that we will need to communicate by sending messages, that are relatively costly in terms of performance. Moreover, as the network can have latency, we should minimize the communication between the computers and that's why we have choose to implement a Root Parallelization. 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.

3.1.3 Shared Memory Parallelization

For the shared memory parallelization we could choose both Root Parallelization or UCT-Treesplit. If we choose UCT-Treesplit, it may prove 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.

3.2 Monte Carlo tree search

One of the advantages of the MCTS algorithm is its generalization. In fact it can be applied to a lot of games. We decided to keep this generalization, therefore we thought about two solutions.

One of them is Genaral Game Playing, a formal language which allows softwares to create the model given its formal rules. However the problem of this approach is that even if it permits to play to unknow games, it prevents the use of efficient heuristic to improve the algorithms. So we decided to choose a second way which consist to exploit the properties of MCTS algorithm. It does not need to know the rules, only the moves. With this in mind we created an interface for the game which define the methods that MCTS algorithm demands. 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 possibles 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 application of our algorithm is the Arimaa game. Therefore we will be able to specialize our algorithm for it in order to improve its efficiency. The main problem is the branching factor[2] of the Arimaa game which average is 17 281 and reaches about 22 000 after 10 moves[3].

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

The reason why the branching factor of a game is so important is because it increases greatly the space that has to be searched in order to guess what will happend multiples moves ahead. In chess after 6 moves, the number of positions evaluated are about 35^6 which is roughtly equivalent to 1,8 billions. In Arimaa, after 3 turns (yours, the opponent and yours again), if you were the explore all positions, you would need to evaluate around 5,2 trillions[4] boards (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 managment.

[2]In a tree, the branching factor is the number of children at each node.
[3]http://arimaa.janzert.com/bf_study/
[4]1 trillion = 1 thousand billions = 10^{12}.

4 Software solutions

4.1 OpenMP

To parallelize our algorithm on the CPU, 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 differents machines, it could works on a single CPU but doesn't provide as many possibilites as its counterparts. Therefore we chose to not use it for that part of the implementation.

4.1.2 Pthreads

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

4.1.3 OpenMP and C++11

OpenMP and C++11 are the only remaining options. C++11 is a simplification of the Boost library inducing the later to be more complete. The following table sumurizes each librairaries 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 get similar performances[5]. However OpenMP is easier to use (precompilier declarations...) and keeps the code clean. C++11 allows a better thread management. Nevertheless it also can easily fall behind his counterpart in term of speed if some mistakes are made : a high price has to be paid.

Considering OpenMP safer to use and get similar results to C++11, we chosed OpenMP to parallelize our algorithm on CPU.

4.2 OpenACC

During our researchs for GPU parallelization, three possibilies have been found:

- OpenMp
- OpenACC
- CUDA

4.2.1 OpenMP

OpenMP supports the GPU programmation 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 decide to not use it.

4.2.2 CUDA

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

4.2.3 OpenACC

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

[5]http://www.cs.colostate.edu/ cs675/OpenMPvsThreads.pdf

lose more time than we gained on the calcul. Somes comparisons have been done, the result is that OpenACC is less than 10% below CUDA in term of performance.

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

4.3 MPI

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

4.3.1 Sockets

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 mecanism.

4.3.2 MPI

MPI, which means Message Passing Interface, is a standardized message-passing system. It allows us to communicate between computers which belong to a network by sending messages between them.

4.3.3 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 is more documentation about MPI than the sockets. It would be simpler to fix the different problems.
- The operation of MPI is based on the sockets so it is similar to the sockets, in terms of performances.

In conclusion, MPI would be simpler to implement, more documented than the sockets while they both have the same performances.

4.3.4 Basic operations

The basic operations of MPI are quite simple. In the main, we first have to initialize MPI with this function :

```
MPI_Init();
```

After the initialization, we have to precise how many computers there are in the network with :

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

We can also define individually the 'name', or rank, of each computers in the network with :

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Then we can send and receive messages with those functions, respectively :

```
MPI_Send();
MPI_Recv();
```

When we have finsh to use MPI, we can terminate it properly :

```
MPI_Finalize();
```

5 Conclusion

The software specifications have been decided : further to the AI, a user interface (*UI*) will be developed. It will include all versions 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 it on *Grid 5000*, a set of clusters of multi-core machines. This parallelisation will be performed using ApenACC on every machine, and MPI between machines. We will first use *Root parallelization*, and try other strategies if there's enough time to do so.

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