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

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

[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.

intelligence which is as optimized as possible.

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.

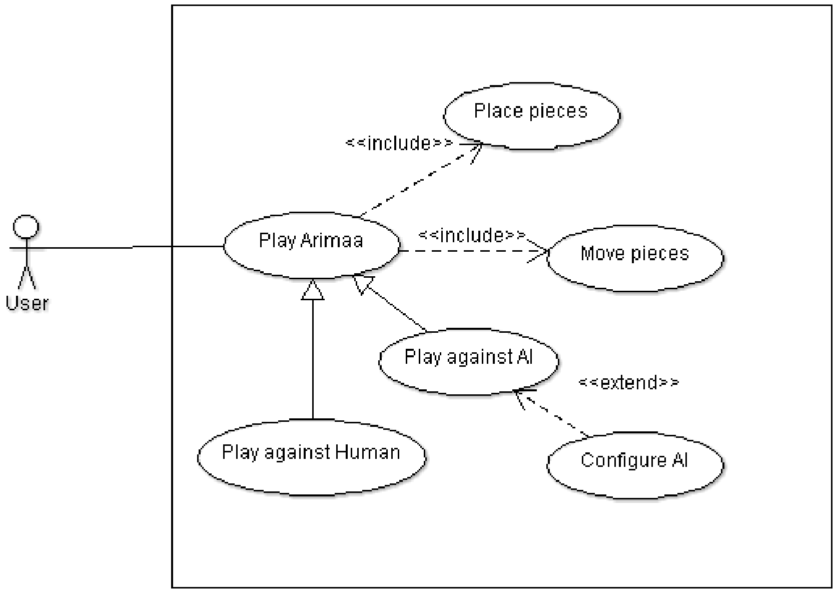

Figure 1: The user-case diagram describing the application

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

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 **??**.

The other efficient parallelization method is called UCT-Treesplit and is depicted if figure **??**. 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.

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 : it will be BLA BLA.

3.2 Monte Carlo tree search

4 Software solutions

4.1 OpenMP

4.2 OpenACC

4.3 MPI

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.