Fast and Furious Game Playing : Monte Carlo Drift
Pre-study and analysis report

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

Supervisors: Nikolaos Parlavantzas, Christian Raymond

10/23/2014

Abstract

This project is about developing an artificial intelligence for a board game using the *Monte Carlo Tree Search Algorithm* and is named as *Fast & Furious Game Playing, Monte Carlo Drift*. The board game chosen for this project is *Arimaa*.

Arimaa was conceived and developed in 2003 by Umar Syed, a computer science engineer. It was intentionally made difficult for computers to play, while following simple rules. Umar Syed offered a prize of 10,000 USD to the first program that could beat a human player in a game of 6 or more matches.

A revolution in scheduling algorithms originated in the *Monte Carlo Tree Search* algorithm (*MCTS*). *MCTS* is a heuristic search algorithm used for making decisions. It concentrates on analyzing the optimum moves by expansion of a search tree of random samplings called the search space. In this project, the techniques of the *MCTS* algorithm are utilized to find what move to make.

The program will be implemented using parallelisation techniques, so as to run it on different systems simultaneously. Eventually, the program will be executed on Grid'5000, on a set of cluster of multi-core machines.

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.

The project is focused on two-players strategy board games, while avoiding games already solved[1]. That is why this project is about the game *Arimaa*.

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 Minimax algorithm does exactly the same thing, but it explores all possibilities. The current position is represented by a node, and the possible states after a move are represented by other nodes, pointing to the parent node. Then it forms a tree. Minimax algorithm develops a tree by creating nodes for all possible moves.

The problem is to compute this heavy algorithm. The *MCTS* algorithm was conceived to choose the nodes it wants to develop; therefore, it was chosen for the project. The *Monte Carlo Tree Search* algorithm has been used in the past for the game of *Draughts*. 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. The initial phase of the project would be the analysis of the latest papers concerning the technologies that might be of use. The consecutive phases will be about making a choice among these technologies, to decide on the specifications. 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 game Arimaa is described in part 2. Then, different algorithms about the game are introduced in part 3.1. Part 4 is dedicated to parallelization methods, and part 4.8 to state of the art technology. In the end, solutions are presented in part 5.1, as well as the schedule in part 5.2. 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 The Game: presentation of Arimaa

Arimaa is played on a board composed of 64 tiles, similar to a chess board. Like Chess, there are 6 types of pieces, instead they are totally different from those in Chess. From weakest to strongest, they are: rabbits (8 per player), cats, dogs, horses (2 of each per player), camels and elephants (one of each per player).

Figure 1: The different piece types in Arimaa.

Each player, starting with the gold player, places all of ones pieces on the two back rows of ones respective side. Then, the gold player plays the first turn. On ones turn, each player disposes of four moves. One can use these moves on a single piece, or on as many pieces as they desire.

All pieces can move on an adjacent square (but not diagonally), except for the rabbit which cannot move backwards. A piece can instead use two moves to push or pull a weaker adjacent enemy piece, as shown in figure 2.

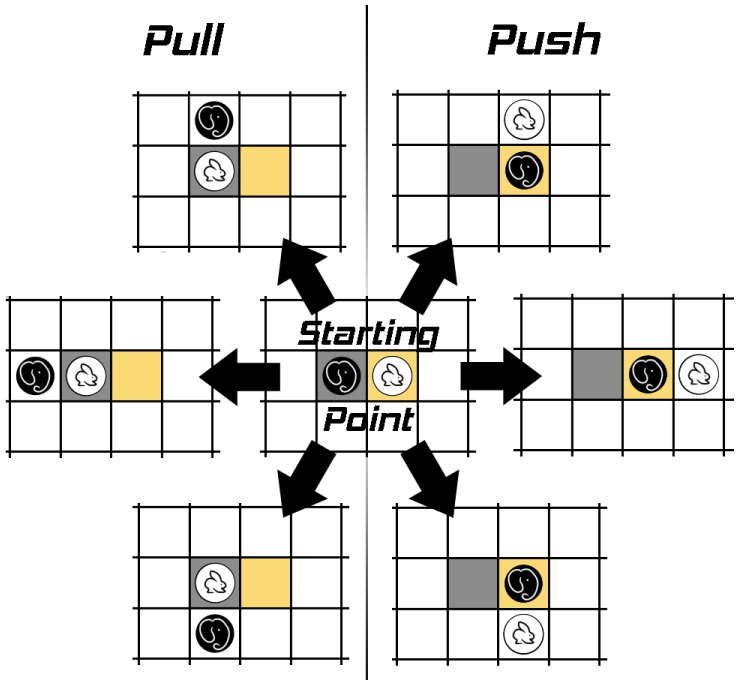


Figure 2: The different ways of pushing or pulling a weaker enemy piece. Here the elephant pushes or pulls the rabbit.

A piece placed adjacent to a stronger enemy piece is frozen. When a piece is frozen, it cannot move. As shown in figure 3, a piece cannot be frozen while there is an ally piece

adjacent to it.

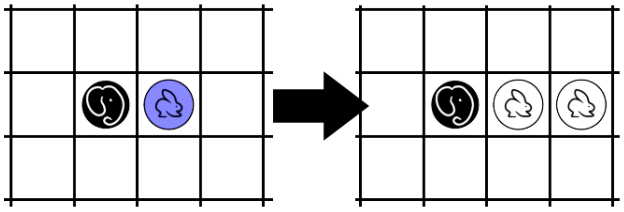


Figure 3: Example of the freezing mechanic. When the two rabbits are together, the one next to the elephant is no longer frozen.

There are four traps on the board. As shown in figure 4, any piece sitting on a trap with no ally piece next to it dies.

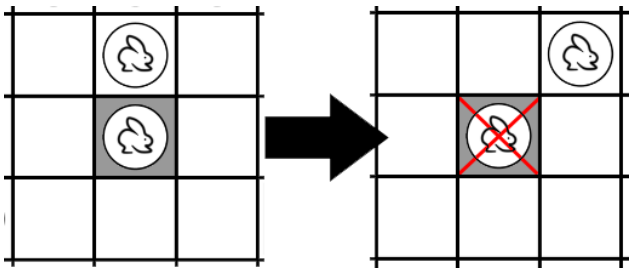


Figure 4: Example of the traps mechanic. As soon as there is no other ally pieces adjacent to the rabbit standing on the trap, it dies.

There are four ways to win the game:

- Victory by reaching the goal: A player wins the game if one of ones rabbits reaches the other end of the board.
- Victory by elimination: A player wins the game if one eliminates all the rabbits belonging to his opponent.
- Victory by elimination: A player wins the game if ones opponent cannot make a move on his turn.
- Victory by repetition: If the same position happens three times in a row, the player that makes it happen the third time loses the game.

3 Algorithms

3.1 Search tree

In order to find the best possible move starting from a position, algorithms build search trees. For example, consider Tic-Tac-Toe: [3]

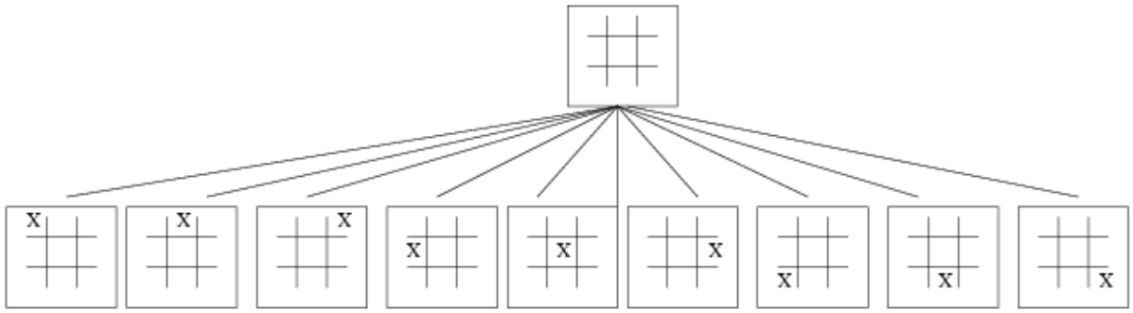


Figure 5: All possible moves starting from an empty board.

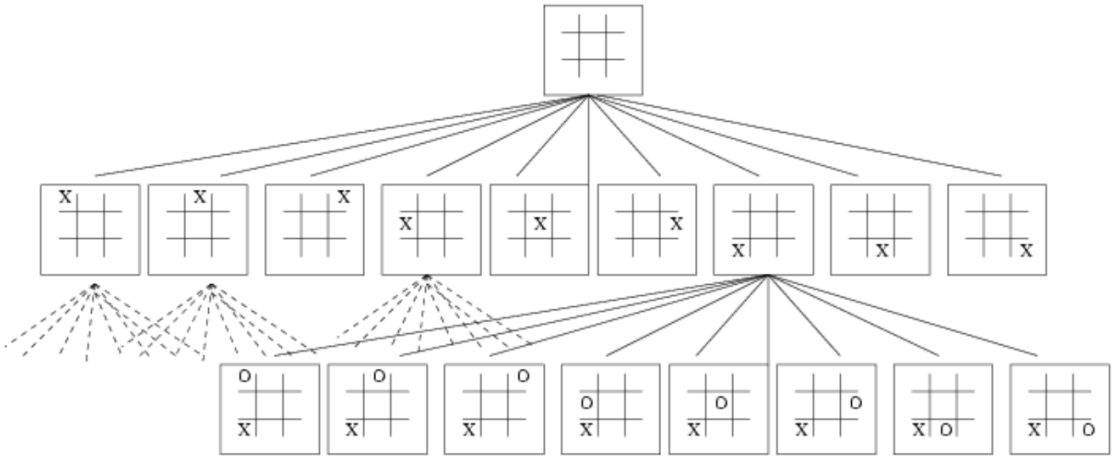


Figure 6: List of all possible moves for the other player.

This expansion is continued until a winning position for the required player is found. Such a tree is called a search tree.

However depending on the number of moves possible for the chosen game, the tree might be too large to be explored completely. Therefore in order to simplify the search, the algorithm will try to evaluate the odds of winning in each position that is explored. Those values will be stored in each node and used by the program to find a good move to play from the present position.

3.2 The Minimax algorithm

The Minimax algorithm is a way of finding an optimal move in a two-players game. In the search tree for a two-players game, there are two kinds of nodes, nodes representing ones

moves and nodes representing the opponent's moves.[1]

Figure 7: Nodes representing ones moves are generally drawn as squares, these are also called *MAX* nodes.

Figure 8: Nodes representing the opponent's moves are generally drawn as circles, these are also called *MIN* nodes.

The goal of a *MAX/MIN* node is to maximize/minimize the value of the subtree rooted at that node. To do this, a *MAX/MIN* node chooses the child with the greatest/smallest value, and that becomes the value of the *MAX/MIN* node.

Note that it is typical for two-players games to have different branching factors at each node. The move one makes could have an impact on what moves are possible for the opponent. In this example, one is ignoring what the game is in order to focus on the algorithm.

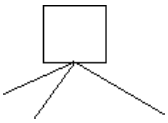


Figure 9: At the start of the problem, Minimax checks the single present node.

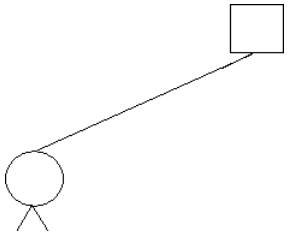


Figure 10: It begins like a depth first search, generating the first child.

So far, no evaluation values has been seen. The way Minimax works is to go down a specified number of full moves (where one *full move* is actually a move by each player), then calculate the evaluation values for states at that depth. For this example, the tree will be explored one move down, which is one more level.

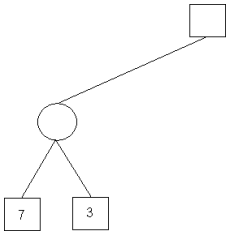


Figure 11: The values for those nodes are generated.

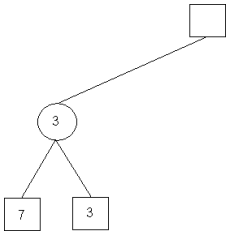


Figure 12: It chooses the minimum of the two child node values, which is 3.

The *MAX* node at the top still has two other children nodes that will be generated and searched.

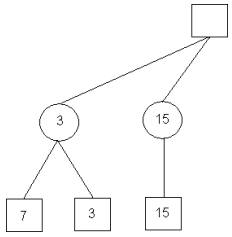


Figure 13: Since there is only one child, the *MIN* node must take its value.

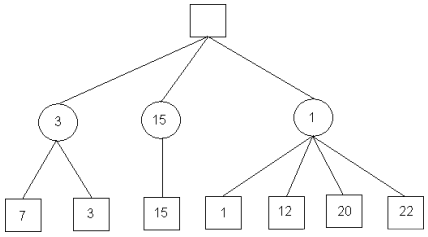


Figure 14: The third *MIN* node chooses the minimum of its child node values, 1.

Finally, all values of the children of the *MAX* node are at the top level, so it chooses the maximum of them, 15, and we get the final solution.

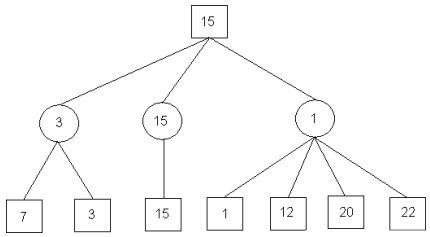


Figure 15: Final tree.

The result of the algorithm is that the best move corresponds to the middle *MIN* node, since it will lead to the best possible state for us one full move down the road.

3.3 The $\alpha\beta$ pruning

The $\alpha\beta$ method is a heuristic that decreases the number of leaves that will be explored by the Minimax algorithm. That way, the size of the tree will be smaller, the algorithm will be able to dive further and the time spend on more interesting subtree is greater.
If the leaf's position is less interesting than its parents, the algorithm will not explore anyfurther.

3.4 Monte Carlo Tree Search Algorithm

3.4.1 Introduction

Monte Carlo Tree Search (MCTS) algorithm is an algorithm used for taking decisions in artificial intelligence (AI) problems such as solving games or decision making in project management. It is based on generating a big number of random simulations in order to get reliable datas. To make such simulations, the program play the moves randomly for each players. Once it reaches a conclusion (*win* or *loss*), the program computes the statistics to get the odds of winning.

3.4.2 How does it works ?

The Algorithm creates a tree with all possible solutions with a small depth. Then it starts to run random simulations starting from the leaves, to test the odds of the outcome. Once the results are numerous enough (time based simulations are usually used), the decision is made depending on the odds of each subsequent leaves.

3.4.3 Example

MCTS algorithm relies on generating a great number of simulations. The following example has been simplified :

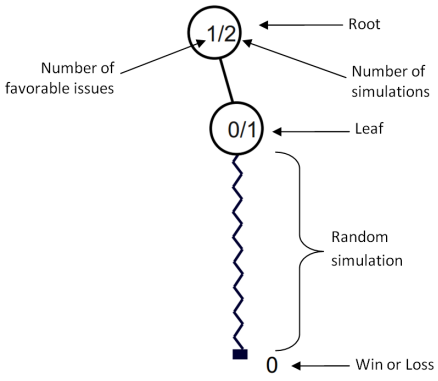


Figure 16: Explanation of the following figures.

Figure 17: Run a first simulation from the root, get a favorable outcome (will be considered as a *win*).

Figure 18: Create a first leaf at depth 1 and run the simulation, get an unfavorable outcome (considered as a *loss*).

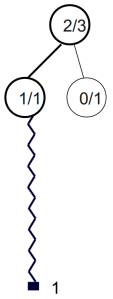


Figure 19: Create a second leaf at depth 1 and run the simulation (*win*).

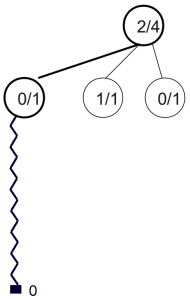


Figure 20: Create a third leaf at depth 1 and run the simulation (*loss*).

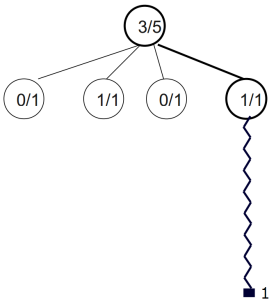


Figure 21: Create a fourth leaf at depth 1 and run the simulation (*win*).

Right now the odds of winning are 3/5. Now, all the possible outcomes at depth 1 have been tested, the tree will be expand on the favorable leaves (here the second and fourth).

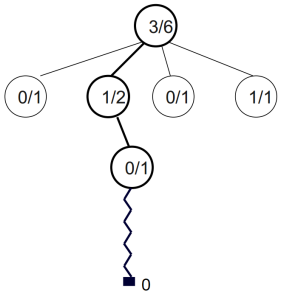


Figure 22: Create a leaf at depth 2 with parent the second leaf at depth 1 and run the simulation (*loss*), update the odds value of the node and making it less interesting than the fourth node. The algorithm will now work on the fourth node.

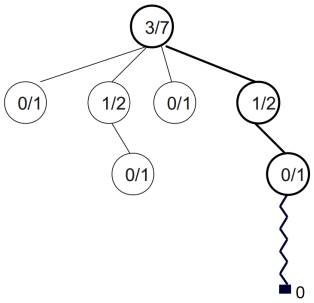


Figure 23: Create a leaf at depth 2 with parent the fourth leaf at depth 1 and run the simulation (*loss*), update the odds value of the node and making it as interesting as the second node. The algorithm will now work on the second node.

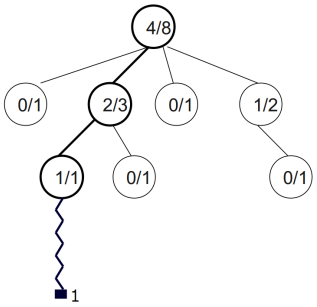


Figure 24: Create a second leaf at depth 2 with parent the second leaf at depth 1 and run simulation (*win*), update the odds value and continue to develop this leaf.

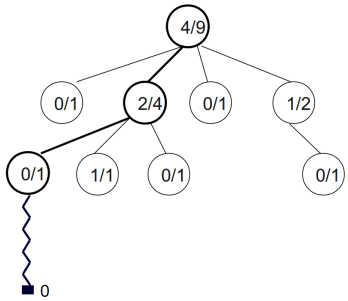
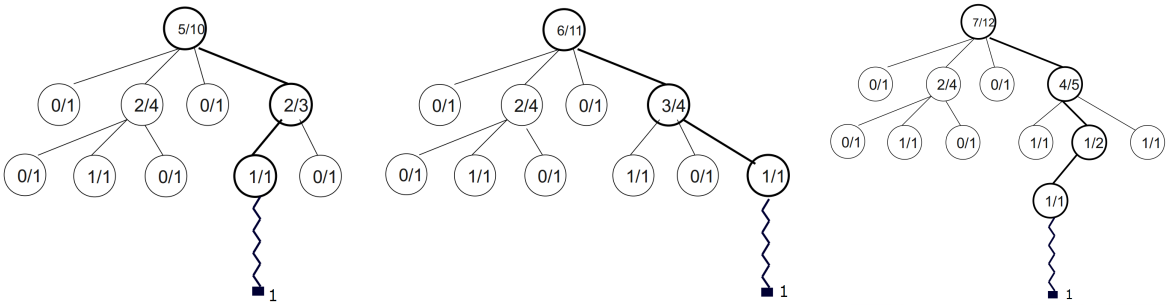


Figure 25: Create a third leaf at depth 2 with parent the second leaf at depth 1 and run simulation (*loss*) and update the odds value and switch to the fourth leaf to balance the search.

Continue the Algorithm until a decent amount of simulation are run and/or the time limit is reached.



Make a decision: the fourth leaf is chosen.

3.4.4 How to select the leaves to develop ?

In the previous example, the leaves without winning rate were not expanding. But depending on the results of the simulations, wins can vary greatly. Therefore, more simulations will be run, on each leaf before choosing the ones to develop. For practical purpose, we will select the leaves to expand that has the highest value of the cost function UCT (Upper Confidence Bound 1 applied to Trees).

$$f = \frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

$UCT\ function$[2]

- w_i: number of wins after the i^{th} node
- n_i: number of simulations after the i^{th} node
- c: exploration parameter – theoretically equal to $\sqrt{2}$ but in practice chosen empirically
- t: total number of simulations in a given tree node, equal to the sum of all n_i

If a leaf is too developed, the development of the nodes will not be worth it. Furthermore, a leaf with low winrate is not completely forgotten.

3.4.5 Why using the Monte Carlo Tree Search algorithm?

The advantage of MCTS with its basic form is that ones does not need to implement functions to improve the researches. Based on its random simulations, it will determine by itself which are the good options and which are not.
The more simulations are run, the more accurate the results will be.

3.4.6 How much power is needed ?

The more possible moves the game has, the more computing power it is required to solve it. In order to get decisions, it needs to go deeper in the tree and to search enough leaves. If the time or number of simulations is not sufficient, the algorithm might miss some important branches and fail to give plausible results. Therefore in order to get decent results, using high-end computer is mandatory, it allows us to get access to multi-threading technology in order to parallelize the simulations.

4 Strategies and state of the art

4.1 Introduction

This part focuses on how the *Monte Carlo Tree Search* is parallelized as a means of optimization. A classic *MCTS* is an algorithm which sequentially creates random developments of the game. In order to speed up the results, develop more nodes of the tree search or even have more realistic statistics,the exploration of the tree will be parallelized. It means that parts of the tree development will be distributed to multiple threads, among multiple computers. Therefore, each thread on each computer will have less executions and the algorithm will be more efficient.

Currently, there are three principal strategies about how to parallelize the tree. They are called: Leaf Parallelization, Root Parallelization and Tree Parallelization[6, 8].

4.2 Leaf Parallelization

The Leaf Parallelization is the easiest way to parallelize the exploration of the tree. In this method, only one thread traverses the tree and adds one or more nodes to the tree when the leaf node is reached. Then a set of threads is created, each one playing the game independently Once they all have finished, they back-propagate their results to the leaf and then, a single thread changes the tree's global results. The Leaf Parallelization method is depicted in figure 28.

The advantage of this method is that its implementation seems very simple. The threads do not need to be synchronized. However, there are two significant problems.

- The time consumed by a thread to finish the game is unknown. Therefore, it will take, on an average, more time to do n games with n threads, than one game with one thread, since this method waits for the last one.
- There is no communication between the threads. If a majority of the threads (the faster ones) have led to a loss, it would be very likely that all of them lead to a loss. And so, the last thread will be executed for nothing.

4.3 Root Parallelization

The second method is the Root Parallelization. It consists in giving each thread the same tree during the same amount of time. They will independently and randomly develop their tree and, at last, they will merge all of the results. This method can also be called *Single-run parallelization* or *Slow-Tree Parallelization*, and is depicted in figure 28.

Its biggest advantage is also one of its drawbacks. Indeed, the threads do not communicate with each other. On one hand, it means there is some redundancy in the development of the

tree. On the other hand, the lack of communication drastically increases the speed of the program. Actually, the strength of this strategy lies in little communication.

4.4 Tree Parallelization

Finally, the third method is called the Tree Parallelization. In this method, multiple threads share the same tree and can randomly choose a leaf to develop it. The main problem of this method is that multiple threads can access the same node and corrupt data given by another thread. To prevent this corruption there is two main methods proposed by Guillaume Chaslot, Mark Winands, and H. Jaap van den Herik [6]. The first one is to use mutexes, or mutual exclusions (mutex), on the tree and the second is to implement a virtual loss. The first method is depicted in figure 28. The other one is explained below.

The mutexes can be either global or local. The global mutexes block access to the whole tree when a thread is accessing it. The time loss induced by the mutexes increases with the number of thread, up to a point where ading another thread does not cause any time gain. The local mutexes block only the node that the thread is using in order not to block the entire tree. This method is better but still implies an important number of locking and unlocking. However, fast-access mutexes and spinlocks can be used to increase the speed of the program.

Nevertheless, according to Markus Enzenberger and Martin Müller [9], by the lack of mutexes, the problem of the data corruption is negligible compared to the speed decrease of the program, especially when the number of threads exceeds two. Therefore, one can assume that, to be the most efficient, one can simply eliminate mutexes.

The second method, the virtual loss, consists in decreasing the value of the node that the first thread accesses. When the second thread searches a node to develop, it will only take this node if it is considerably better than the others. This strategy allows nodes with high probability of winning to be visited by multiple nodes and avoid redundancy on the other ones.

4.5 Hybrid Algorithms

Research about hybrids technologies of parallelization has been investigated. Those technologies use a mix of the parallelization methods eplained in sections 4.2, 4.3 and 4.4. Some parallelization strategies are a combination of multiple methods with some additional content, but they are very complex to compute and are efficient in specific cases.

4.5.1 UCT-Treesplit

The *UCT-Treesplit* algorithm[11] retakes the base of *Root Algorithm* and makes it work on clusters which can compute the nodes. It is depicted in figure 26. It's made for High-Performance Computers (or *HPC*) which have cluster parallelization and shared memory parallelization. Moreover, this method demands a lot of communication and so, the network latency is a very important factor of slow-down.

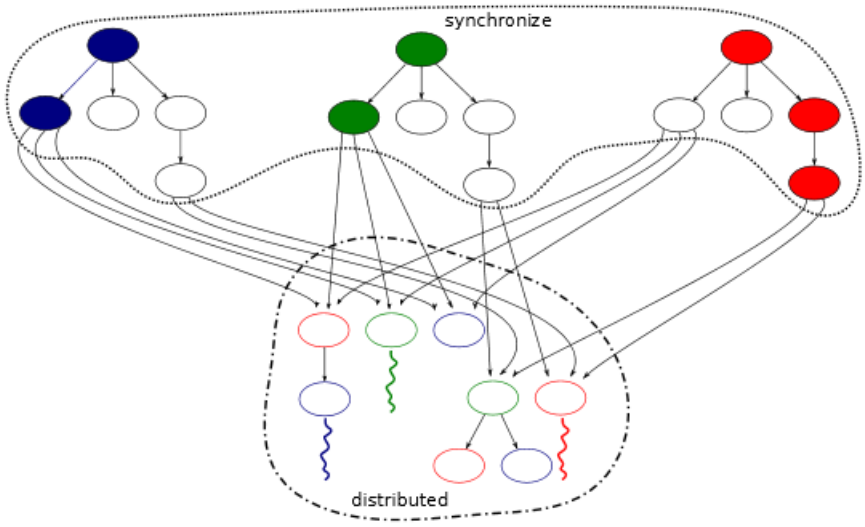


Figure 26: UCT-Treesplit overview

4.5.2 Block Parallelization

Another efficient hybrid algorithm is the *Block Algorithm*[7]. It is depicted in figure 27. It is working by giving some instructions to GPU[2] and some other instructions to CPU[3]. The Block Parallelization is a blending of *Leaf Algorithm* and *Root Algorithm*. As GPUs can run hundreds of threads, it is used to develop a specific node with the *Leaf Algorithm* whereas CPU is used to develop the trees using the *Root Algorithm*.The benefit of this method is that each of these components are used in the way they were meant to be used.

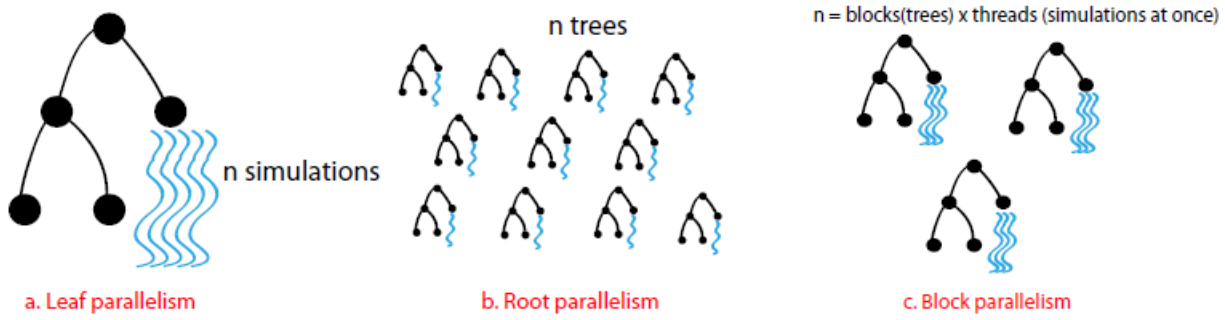


Figure 27: Scheme of the Block Parallelization comparing to Leaf and Root

4.6 Comparison of simple strategies

Currently, the best strategy to adopt is the *Root Parallelization* on those specific test conditions. According to some tests [6, 10], the advantages of the *Root Parallelization* is its

[2]GPU: Graphics Processing Unit, processor dedicated to graphics
[3]CPU: Central Processing Unit, main processor on the computer

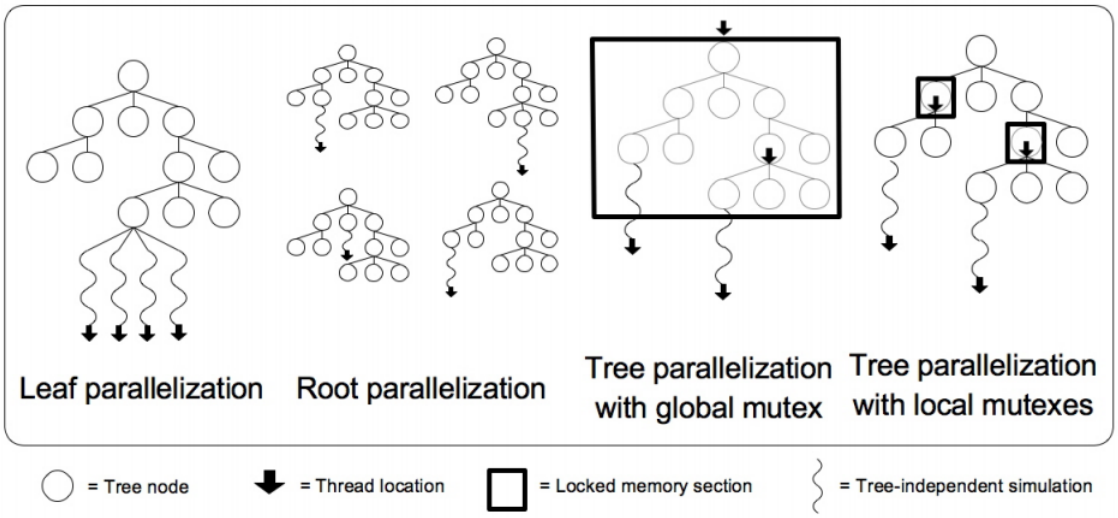


Figure 28: Comparison of all trees

drawbacks. Indeed, for 16 threads, a program using *Root Parallelization* will win 56% of the time against 36.5% for the *Leaf Parallelization*, and 49% for the *Tree Parallelization* with the use of a virtual loss. Moreover, the *Root Parallelization* is always twice as fast as the *Tree Parallelization* with virtual loss. It can be explained by the fact that numerous trees are massive and so much time will be lost doing communication and synchronization. None of these issues exist with the *Root Parallelization*. Furthermore, this method is also very simple to implement.

4.7 Conclusion

The strategy used to parralelize the algorithm depends on the hardware that will be used. Since both cluster parallelization and shared memory parallelization will be used, a simple *Root Parallelization* between the different computers can be chosen, in addition to an algorithm specialized in shared memory parallelization, like *Tree Parallelization*. If the use GPU is preferred over the CPU or even both of them, *Block Parallelization* is used. Also, if the network and the computers are fast, UCT-Treesplit can be used to parallelize the algorithm.

4.8 State of the art of MCTS algorithm

Until 2002, methods based on decompositions and positions evaluations were used in order to solve two-players board games. From 2002 to 2005 the *Monte Carlo Tree Search* algorithm was used in order to find the best moves. 2006 was the first time that it was implemented in a tree (*MCTS*) algorithm has been developped, skyrocketing the results in terms of Artificial Intelligence on the game of Go. On june 5th, 2013, Zen a Go program defeated Takuto Ooomote (9 Dan) with a three stone handicap.[4]

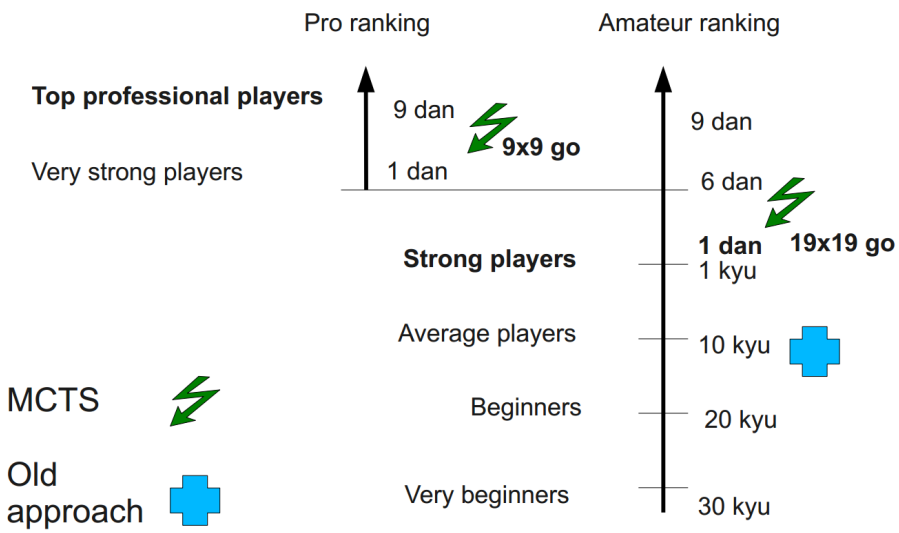


Figure 29: Comparison between Go algorithms and human skill (*2011*)[5].

The ranking system of the game of Go is the following: kyu are for students' ranks, Dan are masters' ranks. Beginners start at 30 kyu and advance downward on the kyu grades system. Once a player ranks over 1 kyu, he will receive the 1st Dan (like the black belt in Judo) and will move upward through the Dan ranks until the 9th.

For *Arimaa*, the *MCTS* algorithm was not the first method applied in order to solve it; the $\alpha\beta$ method was used beforehand. At the moment the top programs (Bomb by David Fotland: 2002 to 2008, Clueless by Jeff Bacher 2009) are ranked about 1800 elo[4]. For comparison, strongest humans players are rated around 2450 elo.[8]

[4] *The Elo rating system is a method for calculating the relative skill levels of players in competitor-versus-competitor games such as chess. It is also used for Arimaa. Beginners rank around 1200 elo, experts around 2000 elo and International Masters over 2400 elo.*

5 Solutions and schedule

5.1 Candidates software and technologies

5.1.1 Language

From the beginning of the project, C++ was chosen to code the software. This is for simple reasons: C++ being compiled, it runs faster so it is more prefered than Java to create an efficient algorithm. Moreover it has several complete graphical libraries, like *Qt* and *SFML*, so it is easy to create the graphical interface that will be needed for the project.

5.1.2 Software

The software used to develop the project has already been chosen. The coding will be realized on *Microsoft visual studio 2013*, and the version manager will be git. These two software are used a lot in the industry so one should to be familiar with them. For the bibliography, Zotero will be used as it allows for the creation of .bib files, the bibliographic format use by LaTeX.

5.1.3 Grid'5000

Grid'5000 is a French set of cluster of computers that links a lot of computers from different research centers. Some of these machines have, in addition to a good CPU, a NVIDIA Tesla GPU that could be use to parallelize the *MCTS* algorithm. Of course using this cluster will require some specific parallelization that will be introduced in the next part.

5.1.4 Parallelization technologies

Thread parallelization and shared memory

OpenMP is an efficient API[5] that supports parallelization and is simple to implement. It consists some pre-compilation instructions, with only a few lines of code. It allows to obtain a parallel version of the algorithm. As it can be seen in the figure 30, it uses a large part of shared memory which allows a quick and efficient communication between the different threads. It is not designed to be used on a cluster of computers, but this problem that is addressed later.

[5]Application Programming Interface

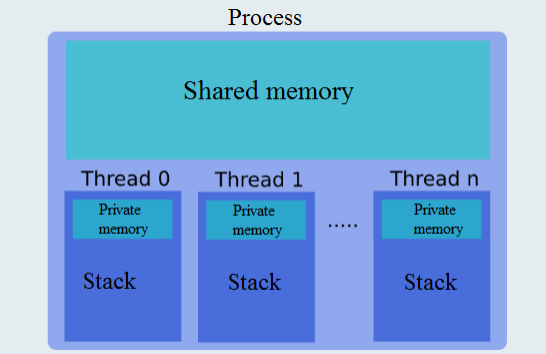


Figure 30: OpenMp: memory management

Computer parallelization and distributed memory

Unlike OpenMP, MPI is able to parallelize a task for several computers. It can be used to design software that uses a cluster of machines. As it can be seen in the figure 31, it does not use shared memory: the data of the parent processes is duplicated at their creation of the child process. It uses messages to permit the communication between processes. One of its advantages is that it makes possible the use of multiple computers: as the data is duplicated, there is not the problem of time to access the memory. But attention is to be paid for the cost of communication between the threads, as if too many messages are used, the time that was gained using the parallelization will be lost. It is adapted for root parallelization, because the only things to be done are to duplicate the data at the beginning and return the result at the end of the assigned time.

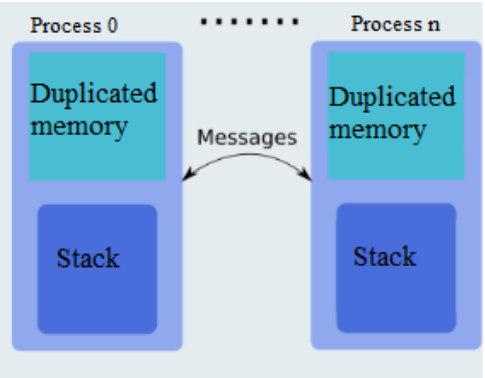


Figure 31: MPI: memory management

Parallelization using a mix of technologies

As stated before, OpenMP and MPI have their respective pros and cons, but their respective problems can be tackled by using both. That means MPI can be used to dispatch

the work onto the different computers and openMP can be used to divide it on the different threads of each machine. This can permit the use of multiple parallelization strategies, for example tree parallelization between the machines and leaf or root parallelization on each machine. Also, it reduces the amount of code needed to manage the threads. It also means that as the interaction between the threads is reduced, to maximize parallelization.

Boost Library

Boost is a well known library for C++ which permits the management of threads, amongst other things. Basically, it can only be used for one computer but it also covers socket handling which makes parallelization between several machines easier. It also holds tools for graph modelisation that can improve a lot the efficiency of the algorithms.

GPU implementation

During the research, it was observed that efficiency of the *MCTS* algorithm improves by using GPU. As Kamil Rocki said in his thesis "Large Scale Monte Carlo Tree Search on GPU"[7], one GPU's performance is equivalent to fifty CPU threads. But this implementation is not perfect, in fact that GPU possess a lot less cache memory, so if the data model is too big the parallelization will be inefficient. Also the trees that it creates will be less deep than those of a CPU, but when a CPU can develop two branches, a GPU will be developing hundreds of branches simultaneously. Another thing that is to be kept in mind is that a GPU can switch to another thread immediately without any cost.

Some machines from Grid5000 have NVIDIA GPUs, so one of the possibilities is to use hybrid parallelization as it described in section 5.1.4 by using MPI (or boost library) and OpenACC (an equivalent of OpenMP which allows the use of the GPU) or CUBA (a framework for GPU parallelization develop by NVIDIA). This way, bigger trees will be developed and give a better solution than using only the CPU, even if the branches are less deep.

5.2 Planning

To manage the work on this project, MS Project will be used to record the length and assigned team member of each task.

At the start of the project, the deadlines were added to MSProject. These deadlines concern the reports (analysis, specifications, ..), the orals and the finalization of the project. After this, task assignements will be decided on a weekly basis, and be added to the planning.

For the moment no programming has been planned because the specifications are not decided yet. The programming that has been done is the creation of a model of the game and its graphical interface, for the team to play the game and to display the future results of the algorithm. This will allow us to try our hand at the game and make the implementation easier.

In the next part of this project, the specifications as well as the model for our project will be decided. Accordingly, the implementation will be planned.

Another important point is role distribution. For the moment, three specific roles have been assigned:

- Gabriel is in charge of the application, who supervises the implementation.
- Baptiste is in charge of the planning, who sets the deadlines of every task, and make sure the deadline are respected.
- Dan is the secretary, who writes the weekly reports and organizes the meetings.

In addition to these roles, the whole team has to research the informations that is needed to realize the software. Those who do not have a special role for the moment will be responsible for one of the next reports.

6 Conclusion

Finally, our project is about creating an Artificial Intelligence able to compete with others computers or players.

Arimaa is a two-players game. It has been designed to be difficult to foresee for computers, but easy to play for humans. In order to realize this project, the *MonteCarlo Tree Search* algorithm will be used to decide what move to play according to its algorithm figures. Because of the high number of moves possible, it will need to choose to only develop some of them, that look most promising. The state of the art of *Arimaa* and the *Monte Carlo Tree Search* algorithm has been already analyzed. Consequently, our work will be based on these theses, to best predict decisions, without doing again what has already been done.F

The point of this project is as well to test our program upon Grid5000, a network of multi-core machines. Then we will be able to compete with other computers, comparing algorithms and power of calculus.

References

[1] http://cs.ucla.edu/ rosen/161/notes/minimax.html.

[2] http://en.wikipedia.org/wiki/monte_carlo_tree_search#exploration_and_exploitation.

[3] https://www.cs.tcd.ie/glenn.strong/3d5/minimax-notes.pdf.

[4] http://www.computer-go.info/h-c/.

[5] Bruno Bouzy. http://www.slideshare.net/bigmc/montecarlo-tree-search-for-the-game-of-go, 2011.

[6] Guillaume Chaslot, Mark Winands, and H. Jaap van den Herik. *Paralel Monte Carlo Tree Search*. PhD thesis.

[7] Kamil Rocki and Suda Reiji. *Parallel Monte Carlo Tree Search on GPU*. PhD thesis.

[8] Tomas Kozelek. Method of mcts and the game arimaa. Master's thesis, 2009.

[9] Markus Enzenberger and Martin Müller. *A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm*. PhD thesis.

[10] Méhat Jean and Cazenave Tristan. Tree parallelization of ary on a cluster.

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