Fast and Furious Game Playing: Monte Carlo Drift

Final report

Prateek Bhatnagar, Gabriel Prevosto,
Benoît Viguier

Supervisors: Nikolaos Parlavantzas, Christian Raymond

02/02/2015

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. This is the final report for the project and includes the specifications about the implementation of the project. In this report, various modules and functions that are developed and implemented will be discussed. A comparison will also be made between the conceptualization and implementation, thereby discussing the achievements and future enhancements. Finally, the tests conducted will be analyzed as well as their results.

2 Achievements with respect to conceptualisation

Specification Vs Implementation

Specification	**Implementation**
root parallelisation on CPU	*tree* parallelisation on CPU
pruning in MCTS while simulation	no pruning in MCTS while simulation
interface between bot & other AI	not yet implemented
parallelization : choice between OpenMP, C++11 and MPI	openMP was chosen
actor model	cannot be used as it requires administrative rights of machine to get installed
GPU parallelisation	could not be done because of time constraint
converter	✓
use of sockets/ZMQ for distribution on clusters use of MPI for distribution on clusters	✗ ✓

Design Vs Implementation

Design	**Implementation**
base algorithm(MCTS)	✓
data structure as assumed in reports	✓
pruning after simulation	✓
bitboard	✓
game abstraction	✓
parameters(MCTS arguments)	✓
fast log	✓
Mersennes twister as a random generator	✓
root parallelisation on cluster switching from root parallelisation to tree parallelisation on CPU	✓ ✓
GUI on Arima GUI on connect4	implemented but not yet linked implemented and linked

Most of the tasks that have been conceptualised during the specification and design phase are implemented as per the plan. Some issues were not implemented like GPU parallelization, because of time constraints and not because of technical reasons.

3 Optimisations and results

3.1 Node structure and impact on the size of the tree

As stated in the previous report, the order of the data in the node has been changed in order to have it use the least possible space.

At the begining of the project each node used to have its board game stored as a *Bitboard* and the move that led to it. As some nodes were having the same state (*Bitboard*), we tried to use an *unordered_map* in order to group them to reduce redundancy and thereby decreasing the size of the tree. However, due to the high number of nodes, each time we created one we had to check whether it was referenced or not. The time required for this operation increases with the size of the tree. This lead to a big cut down in the number of simulations and therefore decreased the reliability of the results provided.

As the algorithm goes down through the search tree, the game can be retraced by playing the chosen move on the a clone of the root board. That way, we saved the space used by each node's board.

Size of a board : 120 octets
Size of a node with the board : 168 octets
Size of a node without the board : 48 octets

With the same number of nodes this represent a reduction in the tree size by 70%. Therefore, while using the same space the new tree is $3,5$ times bigger.

3.2 Tree structure : List vs Array

Initially, each node used to contain the list of its children. These were allocated at the time of creation of the list. Thus the costly memory allocation was done during exploration. In order to improve, we created a list of "available nodes" where nodes were allocated in the memory but not used by the tree. On exploration, they were moved to the children list. Whereas on pruning they were put back in the "available nodes" list.

While the idea is interesting, the child nodes were not stored in continuous regions. During the UCT selection, each child node is visited, thus we didn't take advantage of caching. This is the reason, we moved on to an array structure following the advice of Pascal Garcia. At the start of the program, a large array of node is allocated thus making provisions for the follow up of demands. Each node has to contain the number of children and a pointer to the first one, thereby making sure that they are stored consequently. In the process of loading the first child with the processor optimizations, the follow-up nodes are also loaded, resulting to a great increase of speed.

Moving from a list without pre-allocation implementation to the Array structure and optimized implementation resulted in an increase of the number of simulations by 700%.

3.3 Prunning : Defragmentation model vs buffer copy model

Two ways of prunning the tree have been thought of. The first one, as explained in the previous report, uses two arrays *buffer* and *tree*. While the second one uses only one array *tree*.

The first and simplest one had three steps :
- copy only the selected nodes to the *buffer*
- clean the first array (*tree*)
- copy back the *buffer* to the *tree*.

The second one is inspired on the defragmentation of a hard disk drive: the idea is not to copy the elements to be kept but move them inside the array. It is done in two steps :
- mark the nodes to be deleted.
- move the nodes in order to compact the tree.

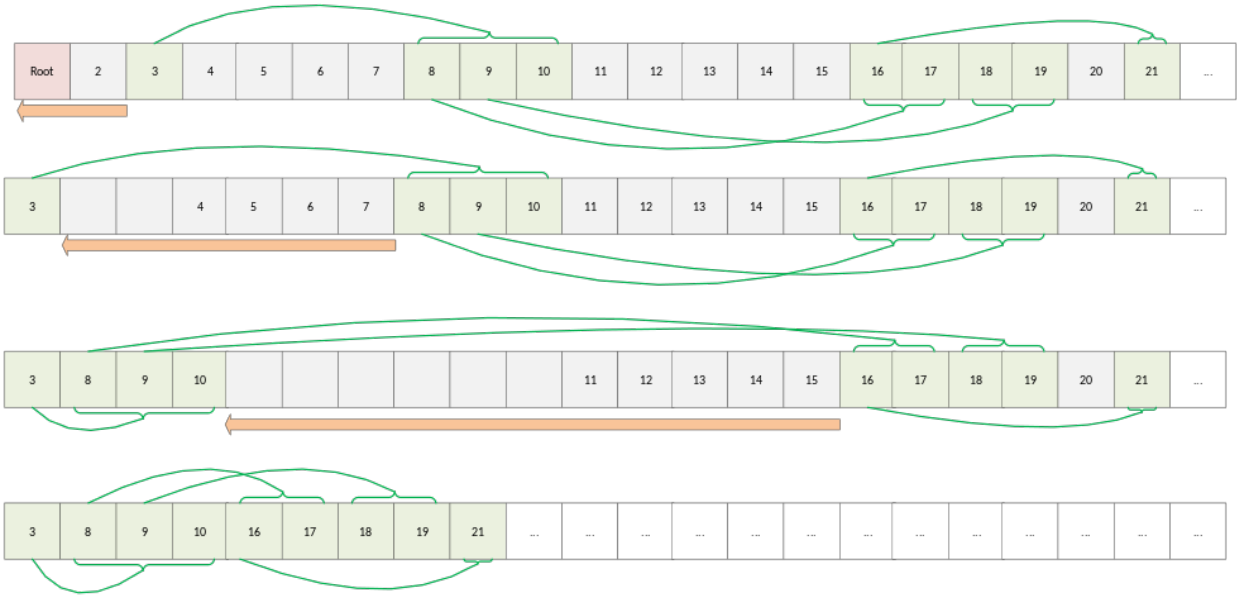

Figure 1: *defragmentation model, the nodes are compacted to the left.*

The aim of this method is to use the RAM for only one array instead of using half of it, due to the presence of the *buffer*. This way, the size of the tree is increased by roughly 50%.

However that second method requires many more updates than expected: each time a node is moved, it is required to change the pointer of its parent, if it is the first node. It also require to update the parent pointer of its children. All this greatly increases the number of calls in the memory. In order to solve such a problem, we used another array, *index* which reference the address of each node. When moving one node, it is required to change its reference while the parent and children points to that direction.

While the benefits of such implementation are clear, the time spent compacting greater trees grows much in comparison to the first method. Due to such heavy drawback, the second method has been dropped.

3.4 Scaling on multi-core machine

A bench mode (*-b* or *–bench*) has been implemented in order to easily follow the evolution of scaling through the subsequent iterations and improvements. We had access to Rodrigue, an old (2009) 24-core-server. It allowed us to compare the number of simulations done by our algorithm through an increasing number of cores (see the following figures).

```
Results :
cores  --- simulations
   1    |   6 778 850
   2    |   8 495 430
   3    |   8 991 876
   4    |   12 746 556
   5    |   15 602 502
   6    |   13 612 338
   7    |   16 426 276
   8    |   16 753 612
----------------------
```

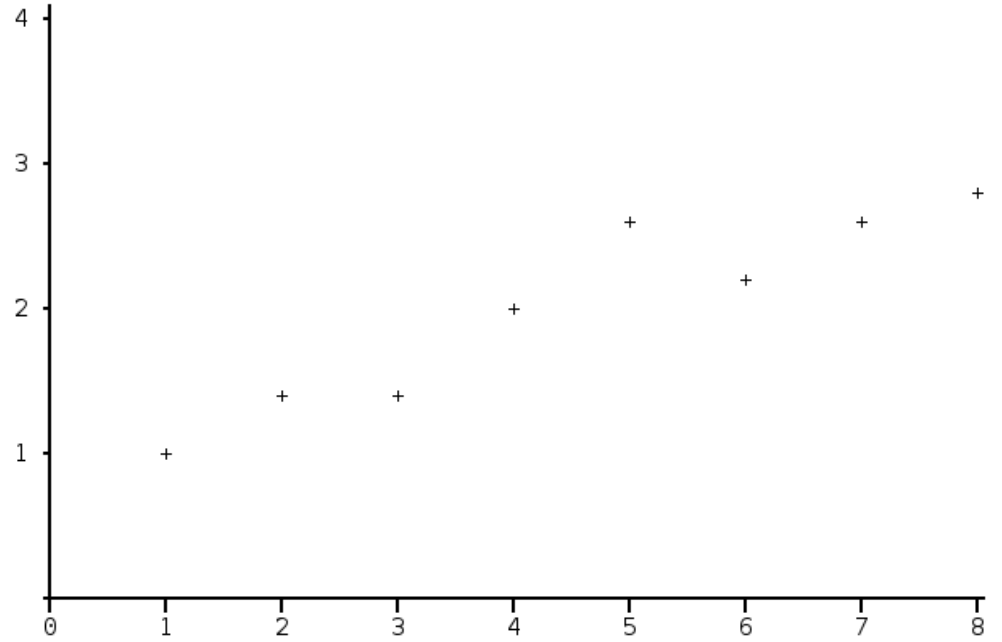

Figure 2: *Bench test on Dell Precision T1700 (4 physical cores, 8 logical cores)*

It is shown that the number of simulations doesn't scale well with the number of cores. We expected better results. However these figures are probably due to a poor implementation. We noticed on Rodrigue that from 9 cores, the number of simulations stabilizes. This might be caused because of possible saturation of the memory bus as a result of too much access from the processors and a possible recurring cache invalidation.

```
Results :
cores  --- simulations
   1    |   1 472 250
   2    |   1 529 165
   3    |   2 009 030
   4    |   2 408 153
   5    |   2 741 521
   6    |   3 146 414
   7    |   3 378 940
   8    |   3 769 850
   9    |   3 896 623
  10    |   3 866 983
  11    |   4 063 647
  12    |   4 312 909
  13    |   4 339 273
  14    |   4 401 165
  15    |   3 897 084
  16    |   4 316 001
  17    |   4 265 860
  18    |   3 916 966
  19    |   4 274 123
  20    |   3 914 136
  21    |   3 877 771
  22    |   4 127 977
  23    |   4 033 295
  24    |   4 174 619
----------------------
```

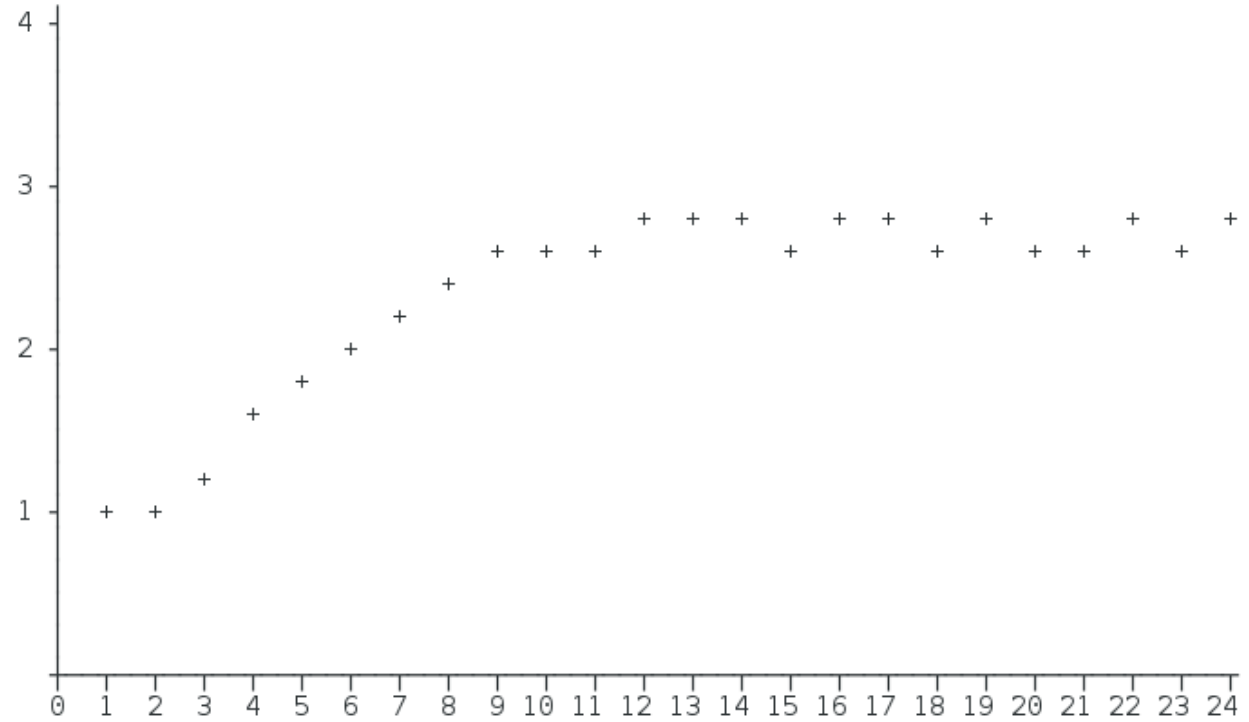

Figure 3: *Bench test on Dodrigue (24 physical cores, 24 logical cores)*

4 Implementation of distribution

4.1 Distribution strategy

Distribution was achieved with *MPI* (Message Passing Interface). It was, as planned, designed using a Master-Worker approach. While the first design planned for a merging of all trees, testing proved it to be too heavy on both processing and communications. That is why a different approach was used: each worker choses the n more promising moves, and sends their winning ratio and number of simulations. Thanks to this, results can be easily merged while only losing data on the moves that were not promising. The n factor is chosen in regards to the tested time of communications and merging, as well as on the game (simple games can have this n be their total number of possible moves).

4.2 Non-blocking calls

With optimization in mind, non-blocking calls were used as often as possible. They allow the process to keep running while the message is not sent or received. However, when the results of a non-blocking receive must be used (or when the buffer of a non-blocking send is reused or otherwise made inaccessible), it requires the use of a waiting function, to ensure that all the data has been correctly sent or received. That means the time gained is comprised between the non-blocking call and this waiting function.

5 Planning

As stated in the third report, we used the agile methodology with iterations to implement our solutions. However, some parts of the planning went wrong. While we were ahead of schedule for two months, we relaxed and lost our advantage in time. We also didn't considerate the weeks spent for the preparation of the exams, resulting into a delay of one month.

5.1 The base algorithm

Despite some memory leaks problem in January, the implementation of the parallelization went smoothly. We also had time to improve the algorithm and to test different kind of implementations during the next two months.

From April, we started to work on the distribution of the algorithm on a cluster and implementation of Arimaa. The later was far more complex than expected and took four times the estimated duration of the task. This was due to the use of bitboards to generate the moves.

5.2 Distribution

For the distributed version of our algorithm, development was pushed back because of blocked ports at the computer science department, that made it impossible for us to test it there. As we had no means to easily recreate a Linux environment of a sufficient amount of machines, we had to wait for this issue to be addressed. The development of the distributed algorithm was thus delayed by almost one month. As the development went on and it appeared that we would be short on time, we decided to drop the ZeroMQ implementation in order to concentrate our efforts on MPI.

5.3 Interface

The development of the interface happened without much problems, but we realized after it was finished that it used 32bit, and that or algorithm needed 64bits. At the time of this writing a solution is still to be found, as SFML 1.6 seems not to be compatible with 64bit, and the use of SFML 2.0 would require some changes done to the code.

6 Conclusion

All the modules of the project are developed and are fully functional. Most of the modules have been implemented as per the plan.

A comparison is made between specification, design and implementation with the initial conception during various phases of the project. Initially, the implementation was going ahead of our planning schedule by two weeks but due to some of the modules not planned properly as we underestimated the work load, we lagged behind. Also, only one week was planned to take off from development, for the preparations of exams but due to the workload of the studies it was increased to two weeks.

A lot of optimization has been done on MCTS algorithm in order to improve its implementation with Arimaa.

Due to the lack of time, only some of the tests have been carried out successfully by the time of this report submission. For example, CPU parallelization, GUI for Arimaa, GUI for connect4, Converter for Arimaa are some of the modules that are tested and implemented successfully.