Tree Parallelization of Ary on a Cluster

Jean Méhat
LIASD, Université Paris 8, Saint-Denis France,
jm@ai.univ-paris8.fr

Tristan Cazenave
LAMSADE, Université Paris-Dauphine, Paris France,
cazenave@lamsade.dauphine.fr

Abstract

We investigate the benefits of Tree Parallelization on a cluster for our General Game Playing program Ary. As the Tree parallelization of Monte-Carlo Tree Search works well when playouts are slow, it is of interest for General Game Playing programs, as the interpretation of game description takes a large proportion of the computing time, when compared with program designed to play specific games. We show that the tree parallelization does provide an advantage, but that it decreases for common games as the number of subplayers grows beyond 10.

Introduction

Monte-Carlo Tree Search is quite successful for General Game Playing (Finnsson and Björnsson 2008; Méhat and Cazenave 2010b) even if other approaches such as the knowledge-based approach also exist (Haufe et al. 2011). An important feature of Monte-Carlo Tree Search is that it improves with more CPU time. Therefore in the time allocated to make a move, it is desirable to develop as much as possible the Monte-Carlo tree in order to gain as much as possible information on the available moves. Parallelizing Monte-Carlo Tree Search is a promising way to make use of more CPU power.

In this paper we investigate the parallelization of our General Game Playing player Ary (Méhat and Cazenave 2010a) on a cluster of machines.

The next section details the parallelization of Monte-Carlo Tree Search. The third section shows how we have applied it to Ary. The fourth section gives experimental results for various games from previous General Game Playing competitions.

Parallelization of Monte-Carlo Tree Search

There are multiple ways to parallelize Monte-Carlo Tree Search (Cazenave and Jouandeau 2007). The most simple one is the *Root Parallelization*. It consists in running separately on different machines or cores the Monte-Carlo Tree Search algorithm developing independently its specific tree, and in collecting at the end of the allocated time the results of the separate searches. Each move at the top of the tree is qualified by combining the results of the independent searches. This way of parallelizing is extremely simple and works well for some games such as Go (Chaslot, Winands, and van den Herik 2008) or some games from General Game Playing represented in the Game Description Language such as Checkers or Othello (Méhat and Cazenave 2010c).

Another way of parallelizing Monte-Carlo Tree Search is the *Tree parallelization* (Cazenave and Jouandeau 2008; Chaslot, Winands, and van den Herik 2008). It consists in sharing the tree among the various machines or cores. On a multi-core machine there is only one tree in memory and different threads descend the tree and perform playouts in parallel. On a cluster the main machine holds the tree and descends it. After each descent it selects another available machine of the cluster and sends the moves associated to the descent to this machine. The remote machine then plays the moves it has received, starting from the position at hand and continues with a random playout. It then sends back to the main machine the result of the playout and becomes available again.

Tree parallelization of the Fuego Go program using lock-free multi-threaded parallelization has been shown to improve significantly its level of play (Enzenberger and Müller 2009).

Centurio is an UCT based General Game Playing agent. It uses multi-core tree parallelization and cluster based root parallelization (Möller et al. 2011).

Gamer is also an UCT based General Game Playing agent. Experiments with the tree parallelization of Gamer on a multi-core machine brought speedups between 2.03 and 3.95 for four threads (Kissmann and Edelkamp 2011).

Tree parallelization of Ary

Current General Game players using Monte-Carlo Tree Search do not perform many simulations when compared with programs playing specific games. This is due to the way the game description is used for generating legal moves, applying joint moves, determining if a situation is terminal and getting the scores of the players.

In Ary, the game description received from the Game Master in the Game Description Language (GDL) is translated into Prolog and interpreted by a Prolog interpreter. When a node is created in the tree, its legal moves — or the scores of the players in terminal situations – are obtained from the interpreter and stored in the node; they are available for further descents without interaction with the interpreter.

On the other hand, when performing playouts, the interpreter is used at each step to analyze the current situation. The results of this analysis are discarded once they have been used to avoid saturating the memory.

Playouts are slow in General Game Playing, and tree parallelization of Monte-Carlo Tree Search on a cluster gives better speedups when playouts are slow (Cazenave and Jouandeau 2008). It is therefore natural to try Tree Parallelization of our General Game Playing agent Ary on a cluster.

In the cluster, one machine is distinguished as the Player: it interacts with the Game Master and maintains the UCT tree. We name the other the Subplayers; they only perform playouts at the request of the Player. All transmission between the Player and a subplayer are done via standard TCP streams.

At the beginning of a match, the Player transmits to all the subplayers the GDL description of the game received from the Game Master.

Result reception in the Player

Before requesting a playout and before each descent in the UCT tree, the Player scans with a *select* system call its connections with the Subplayers to detect which ones have data available. The available data are playout results: they are read and used to update the UCT tree, and the Subplayers are marked as available for another playout.

Playout request in the Player

Algorithm 1 Main algorithm in the Player.

```
while the available time is not elapsed do
  receive playout results if any
  process received playout results
  node ← root node
  while it is possible to descend the UCT tree do
    select child node
    node ← child
  end while
  expand node
  if node is terminal then
    update tree
  else
    receive playout results if any
    while not available Subplayer do
      process received playout results
      wait for data from any Subplayer
    end while
    send node description to the available Subplayer
    process received playout results
  end if
end while
```

Algorithm 1 presents the main algorithm in the Player, descending in the UCT tree, requesting playouts from the Subplayers and receiving their results.

The Player descends the UCT tree. When it arrives at a leaf of the built tree, it expands it into a new node and if the node is not terminal, it selects a Subplayer, by scanning their states until finding one marked as available. This scan is done in a fixed order, permitting to establish a preference order between the Subplayers. When all the Subplayers are busy, the Player waits until one has finished the task at hand and reaches the available state.

Once a subplayer is found available, the Player sends to it the situation in the node in GDL. We opted to send the current situation instead of the sequence of moves used from the root node as done usually in Tree Parallelization. It avoids to have to interpret the application of this sequence of moves in the subplayer, which necessitates slow interactions between the Subplayer and its GDL interpreter.

Subplayer loop

Algorithm 2 Subplayer algorithm

```
receive game description
while true do
  get a state description
  play a playout
  send the playout result to the main machine
end while
```

The algorithm 2 resumes the work a Subplayer.

The Subplayers receive a game description in GDL, load it into their GDL interpreters and then enter a loop.

They wait for a description of a situation of the game, play a completely random playout until a terminal situation, and send the results to the Player. In the current setting, the result is only the score of each player in the final situation, but they might send back the sequence of moves played in the playout, at the cost of a slightly slower communication.

Algorithm 3 Send algorithm in the Player.

```
while not available Subplayer and not time elapsed do
  receive playout results if any
end while
if not time elapsed then
  send current node description
  receive playout results if any
  process received playout results
end if
```

Experimental results

We made a single process Ary, using a single thread to descend into the tree and run the playouts, play matches in a variety of games against a version of Ary running on a cluster using between 1 and 16 Subplayers.

The cluster is made a mixture of standard 2 GHz, 2.33 GHz and 3 GHz PC with two gigabytes of central memory running Linux connected via a switched 100 Mbits Ethernet network. Each machine hosted only a single Subplayer or Player to avoid race for memory between the players. For each match, the single Player, the parallel Player and the

	Number of subplayers				
game	1	2	4	8	16
Breakthrough	18	40	58	68	77
Connect 4	26	38	38	42	50
Othello	41	68	67	81	96
Pawn whopping	43	36	51	54	59
Pentago	38	40	55	73	87
Skirmish	73	76	76	78	79

Table 1: The results of the Tree Parallel Player running as second player against a single player, averaged over 100 matches.

	Number of subplayers				
game	1	2	4	8	16
Breakthrough	44	65	60	67	65
Connect 4	28	44	63	66	75
Othello	59	60	72	84	83
Pawn whopping	44	45	43	46	35
Pentago	35	54	68	64	68
Skirmish	71	71	74	76	71

Table 2: The results of the Root Parallel Player running as second player against a single player, from [Méhat and Cazenave, 2010c].

Subplayers were dispatched at (pseudo)-random between available machines.

The matches were run with 8 seconds of initial time and 8 seconds per move. The games tested were *Breakthrough*, *Connect 4*, *Othello*, *Pawn whopping*, *Pentago* and *Skirmish*. The rules used were the ones available on the Dresden Game Server.

For each game, we ran 100 matches with the Tree Parallel Player as second player, except for setting with 16 subplayers where the number of matches was limited to 70 because of time constraints on the use of the cluster.

Results of the matches

The results of each player are presented in table 1. There is only a slight improvement for the games *Skirmish* and *Pawn whopping*, while it is particularly notable for *Breakthrough*, *Pentago* and particularly at *Othello*. The game *Connect 4* is in-between, with results that get better as the number of subplayers augments, but not as much as in *Breakthrough*, *Pentago* and *Othello*.

Comparison with Root Parallelism

These results can be compared with those presented in (Méhat and Cazenave 2010c), where the same games were played using Root Parallelism in the same settings on the same machines, except that the matches were run with a 10 seconds playing time (figure 2). The results used are those obtained by combining the accumulated values and number of experiments in the root nodes of the trees developped independently in the subplayers, as it is the one that gave the best results for multiplayer games.

Root Parallelism did work for *Breakthrough* and *Skirmish*, and Tree Parallelism also does bring an amelioration for these games. While *Pawn whopping* did not get better with Root Parallelism, it shows some amelioration until eight subplayers with Tree Parallelism.

Secondly, the overall results with Tree Parallelism with 16 subplayers are better than with Root Parallelism in all the games, except *Connect 4*.

The differences between Root Parallelism and Tree Parallelism reside in the sharing of nodes, the choice of branches to explore and the cost of communications. With Root Parallelism, the same node has to be expanded into every Subplayer where it is explored, while with Tree Parallelism, the node is expanded only once. In Root Parallelism, the choice of the branch in the UCT descent phase are only based on the node explored in the the Subplayer, while with Tree Parallelism the results of the playouts of all the Subplayers are taken into account. Finally, Root Parallelism incurs only one interaction per played move, when Tree Parallelism needs an interaction for every playout delegated to a Subplayer.

When there is only one Subplayer, there is only one tree, developped in the Subplayer for Root Parallelism or in the Player for Tree Parallelism. The only distinguishing factor between the two meethods is then the communication cost, whose impact should be greater in games with short playouts. It comes as a surprise that Root Parallelism with one Subplayer exhibits significantly better results than Tree Parallelism with one Subplayer for *Breakthrough* and *Othello*, the two games where the playouts are slow. This point needs more investigations.

Use of the subplayer during the game

The benefits obtained from delegating playouts to subplayers vary between phases of the match. At the match goes on, the time of the descent of the tree tends to augment with the depth of the tree, while the time for a playout tends to diminish with the number of moves in the playout. Moreover, when the match is nearly finished, the descent in the UCT tree arrives with a growing frequency to terminal positions where there is no need to run playouts.

This variation has an influence on the benefit brought by using Subplayers. To measure it, we computed the average number of playouts computed by each subplayer at each move in the *one against 16* matches. The following figure shows these numbers for the first, the fourth, the eighth and the sixteenth Subplayer for some of the studied games. As the first available subplayers is solicited when one is needed, it allows to evaluate how useful is each subplayer.

For the game *Skirmish*, the evolution is presented in figure 1. The subplayers are able to compute about 120 playouts at the beginning of the match, and the last subplayer is only used at half of it capacity. As the match advances, the playouts get shorter and their number grow. After the tenth move, the 8th subplayer is less used, until move 27 where its

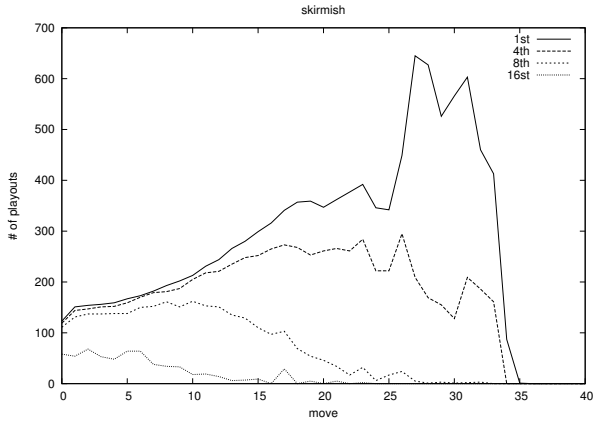

Figure 1: The evolution of the number of playouts for some subplayers in the game of *Skirmish* with 16 Subplayers.

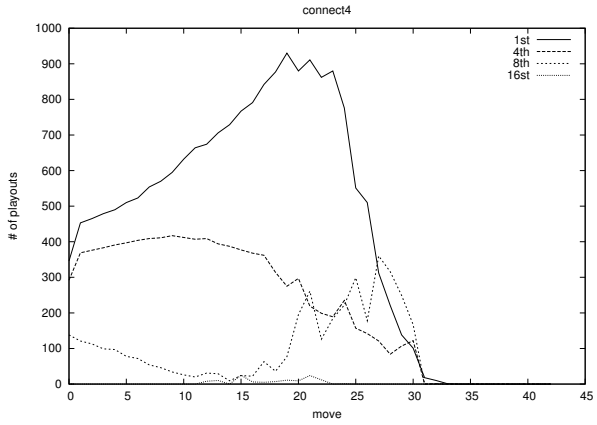

Figure 2: The evolution of the number of playouts for some subplayers in the game of *Connect 4* with 16 Subplayers.

use descends to 0. The curves for *Pawn whopping* are quite similar.

For the game *Connect 4*, presented in figure 2, the 16th subplayer is not solicited during the whole match, and the 8th subplayer is only half busy at the beginning. After move 17, it enters into action. The 4th and 8th subplayer are as busy between moves 20 and 25.

For the game *Pentago*, presented in figure 3, all the subplayers are used at full capacity until move 11 ; then the utility of the 16th subplayer diminishes until getting nearly not used at move 30. The 8th subplayer is used until move 20.

For the game *Breakthrough*, the evolution presented in figure 4 has the same structure, but here the 16th subplayer is kept busy nearly until the end of the game but presents a peak of activity near the end of the game.

The curve for *Othello* appears in figure 5. The interpretation of these rules are pretty slow and the number of playouts at the beginning is around 25 for all the subplayers. The 8th subplayer is kept busy until move 35 and the 4th subplayer nearly until move 50.

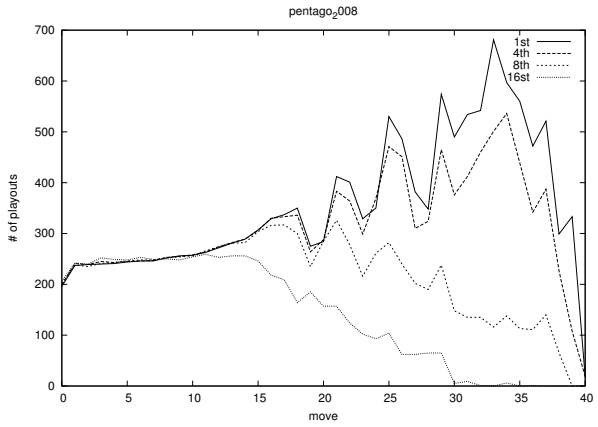

Figure 3: The evolution of the number of playouts for some subplayers in the game of *Pentago* with 16 Subplayers.

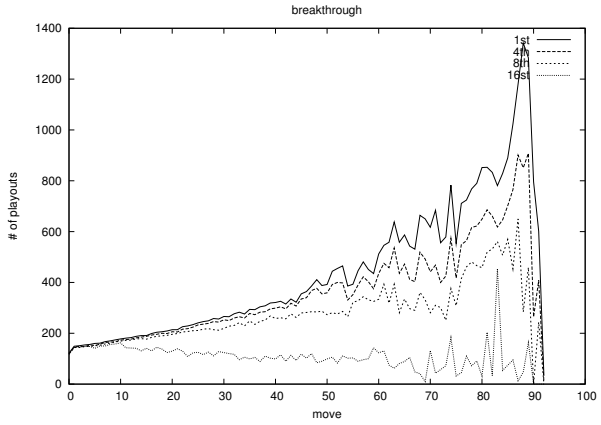

Figure 4: The evolution of the number of playouts for some subplayers in the game of *Breakthrough* with 16 Subplayers.

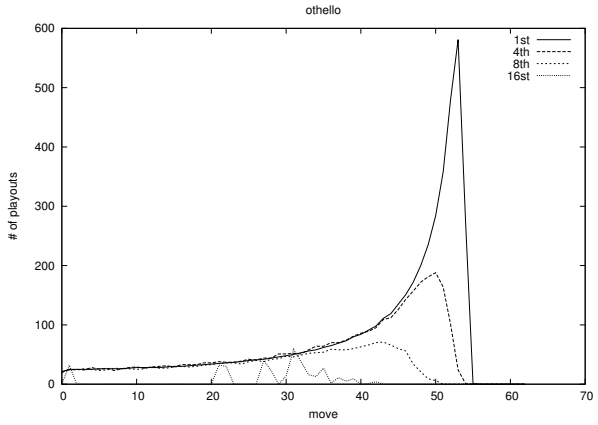

Figure 5: The evolution of the number of playouts for some subplayers in the game of *Othello* with 16 Subplayers.

Conclusion

We have implemented a Tree Parallel version of our General Game Playing agent Ary, and tested it on a variety of games.

We have shown that, in contrast with the Root Parallel version studied in (Méhat and Cazenave 2010c) that worked for some games but not for others, the Tree Parallel version improves the results against a serial player on all considered games, on some games more that others. This improvement is not directly related to the length of the playout, but to the ability of the Player to keep the Subplayers busy at the beginning of a match.

For ordinary games, there is no great benefit to be expected from a number of subplayers over 16.

Acknowledgement

We are grateful to David Elaissi, Nicolas Jouandeau and Stéphane Ténier who gave us access to the machines where the tests were run.

References

Cazenave, T., and Jouandeau, N. 2007. On the parallelization of UCT. In *CGW*, 93–101.

Cazenave, T., and Jouandeau, N. 2008. A parallel Monte-Carlo tree search algorithm. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, 72–80. Springer.

Chaslot, G.; Winands, M. H. M.; and van den Herik, H. J. 2008. Parallel monte-carlo tree search. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, 60–71. Springer.

Enzenberger, M., and Müller, M. 2009. A lock-free multithreaded monte-carlo tree search algorithm. In *ACG*, volume 6048 of *Lecture Notes in Computer Science*, 14–20. Springer.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*, 259–264.

Haufe, S.; Michulke, D.; Schiffel, S.; and Thielscher, M. 2011. Knowledge-based general game playing. *KI* 25(1):25–33.

Kissmann, P., and Edelkamp, S. 2011. Gamer, a general game playing agent. *KI* 25(1):49–52.

Méhat, J., and Cazenave, T. 2010a. Ary, a general game playing program. In *Board Games Studies Colloquium*.

Méhat, J., and Cazenave, T. 2010b. Combining UCT and nested monte-carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4):271–277.

Méhat, J., and Cazenave, T. 2010c. A parallel general game player. *KI* 25(1):43–47.

Möller, M.; Schneider, M.; Wegner, M.; and Schaub, T. 2011. Centurio, a general game player: Parallel, java- and asp-based. *KI* 25(1):17–24.