A Parallel Monte-Carlo Tree Search Algorithm

Tristan Cazenave and Nicolas Jouandeau

LIASD, Université Paris 8, 93526, Saint-Denis, France
`cazenave@ai.univ-paris8.fr`
`n@ai.univ-paris8.fr`

Abstract. Monte-Carlo tree search is a powerful paradigm for the game of Go. We present a parallel Master-Slave algorithm for Monte-Carlo tree search. We experimented the algorithm on a network of computers using various configurations: from 12,500 to 100,000 playouts, from 1 to 64 slaves, and from 1 to 16 computers. On our architecture we obtain a speedup of 14 for 16 slaves. With a single slave and five seconds per move our algorithm scores 40.5% against GNUGO, with sixteen slaves and five seconds per move it scores 70.5%. We also give the potential speedups of our algorithm for various playout times.

1 Introduction

Works on parallelization in games are mostly about the parallelization of the Alpha-Beta algorithm [2]. We address the parallelization of the UCT algorithm (Upper Confidence bounds applied to Trees). This work is an improvement over our previous work on the parallelization of UCT [3]. In our previous work we tested three different algorithms. The single-run algorithm uses very few communications, it consists in having each slave computing its own UCT tree independently of the others. When the thinking time is elapsed, it combines the results of the different slaves to choose its move. The multiple-runs algorithm periodically updates the trees with the results of the other slaves. The at-the-leaves algorithm computes multiple playouts in parallel at each leaf of the UCT tree. In this paper we propose a different parallel algorithm that develops the UCT tree in the master part and performs the playouts in the slaves in parallel, it is close to the algorithm we presented orally at the Computer Games Workshop 2007 and that we used at the 2007 Computer Olympiad.

Monte-Carlo Go has recently improved to compete with the best Go programs [4, 6, 7]. We show that it can be further improved using parallelization.

Section 2 describes related work. Section 3 presents the parallel algorithm. Section 4 details experimental results. Section 5 concludes.

2 Related Works

In this section we expose related works on Monte-Carlo Go. We first explain basic Monte-Carlo Go as implemented in GOBBLE in 1993. Then we address the combination of search and Monte-Carlo Go, followed by the UCT algorithm.

2.1 Monte-Carlo Go

The first Monte-Carlo Go program is GOBBLE [1]. It uses simulated annealing on a list of moves. The list is sorted by the mean score of the games where the move has been played. Moves in the list are switched with their neighbor with a probability dependent on the temperature. The moves are tried in the games in the order of the list. At the end, the temperature is set to zero for a small number of games. After all games have been played, the value of a move is the average score of the games it has been played in first. GOBBLE-like programs have a good global sense but lack of tactical knowledge. For example, they often play useless Ataris, or try to save captured strings.

2.2 Search and Monte-Carlo Go

A very effective way to combine search with Monte-Carlo Go has been found by Rémi Coulom with his program CRAZY STONE [4]. It consists in adding a leaf to the tree for each simulation. The choice of the move to develop in the tree depends on the comparison of the results of the previous simulations that went through this node, and of the results of the simulations that went through its sibling nodes.

2.3 UCT

The UCT algorithm has been devised recently [8], and it has been applied with success to Monte-Carlo Go in the program MOGO [6, 7] among others.

When choosing a move to explore, there is a balance between exploitation (exploring the best move so far), and exploration (exploring other moves to see if they can prove better). The UCT algorithm addresses the exploration/exploitation problem. UCT consists in exploring the move that maximizes $\mu_i + C \times \sqrt{\frac{log(games)}{child_i \rightarrow games}}$. The mean result of the games that start with the c_i move is μ_i, the number of games played in the current node is $games$, and the number of games that start with move c_i is $child_i \rightarrow games$.

The C constant can be used to adjust the level of exploration of the algorithm. High values favor exploration and low values favor exploitation.

3 Parallelization

In this section, we present the run-time environment used to execute processes on a cluster. Then we present and comment the master part of the parallel algorithm. Eventually, we present the slave part of the parallel algorithm.

3.1 The Parallel Run-Time Environment

To improve search, we choose message passing as parallel programming model, which is implemented in the standard MPI, also supported by Open MPI [5]. Open MPI is designed to achieve high performance computing on heterogeneous clusters. Our cluster is constituted with classical personal computers and with a SMP head node that has four processors. The resulting cluster is a private network connected with TCP

Gigabit network. Both communications are done only with the global communicator MPI_COMM_WORLD. Each hyper-threaded computer that allows to work on two threads at once, supports of one to four nodes of our parallel computer. Each node runs one task with independent data. Tasks are created at the beginning of the program's execution, via the use of the master-slave model. The SMP head node is always the master. All Go Text Protocol read and write commands are realized from and to the master. Slaves satisfy computing requests.

3.2 The Master Process

The master process is responsible for descending and updating the UCT tree. The slaves do the playouts that start with a sequence of moves sent by the master.

```
Master ()
    MasterLoop($board[\,], color, ko, time$);
    for($s \leftarrow 0; s < nbSlaves; s$++)
        send($s, END\_LOOP$);
    return bestUCTMove ();

MasterLoop($board[\,], color, ko, time$)
    for($s \leftarrow 0; s < nbSlaves; s$++)
        send($s, board[\,], color, ko$);
        $seq[s][\,] \leftarrow$ descendUCTTree ();
        send($s, seq[s][\,]$);
    while(moreTime($time$))
        $s \leftarrow$receive();
        $result[s] \leftarrow$receive();
        updateUCTTree($seq[s][\,], result[s]$);
        $seq[s][\,] \leftarrow$ descendUCTTree ();
        send($s, seq[s][\,]$);
    for($i \leftarrow 0; i < nbSlaves; i$++)
        $s \leftarrow$receive();
        $result[s] \leftarrow$receive();
        updateUCTTree($seq[s][\,], result[s]$);
```

ALG. 1: Master Algorithm.

The master starts sending the position to each slave. Then it develops the UCT tree once for each slave and sends them an initial sequence of moves. Then it starts its main loop (called MasterLoop) which repeatedly receive from a slave the result of the playout starting with the sent sequence, update the UCT tree with this result, create a new sequence descending the updated UCT tree, and sends this new sequence to the slave.

The master finishes the main loop when no more time is available or when the maximum number of playouts is reached. Before stopping, it receives the results from all the children that are still playing playouts until no more slave is active.

The master part of the parallel algorithm is given in algorithm 1.

3.3 The Slave Process

The slave process loops until the master stops it with an END_GAME message, otherwise it receives the board, the color to play and the ko intersection and starts another loop in order to do playouts with this board configuration.

In this inner loop, it starts receiving a sequence of moves, then it plays this sequence of moves on the board, then completes a playout and sends the result of the playout to the master process.

The slave part of the parallel algorithm is given in algorithm 2.

```
SlaveLoop()
    $id$ ←slaveId()
    while($true$)
        if(receive($board[\,], color, ko$) $== END\_GAME$)
            break;
        $state \leftarrow CONTINUE$;
        while($state == CONTINUE$)
            $state \leftarrow$ SlavePlayout();
    return;

SlavePlayout()
    if(receive($sequence[\,]$) $== END\_LOOP$)
        return $END\_LOOP$;
    for($i \leftarrow 0; i < sequence$.size()$; i$++)
        playMove($sequence[i]$);
    $result \leftarrow$ playRandomGame();
    send($id$);
    send($result$);
    return $CONTINUE$;
```

ALG. 2: Slave Algorithm.

4 Experimental Results

Tests are run on a simple network of computers running LINUX 2.6.18. The network includes 1 Gigabit switches, 16 computers with 1.86 GHz Intel dual core CPUs with 2 GB of RAM. The master process is run on the server which is a 3.20 GHz Intel Xeon with 4 GB of RAM.

In our experiments, UCT uses $\mu_i + 0.3 \times \sqrt{\frac{log(games)}{child_i \rightarrow games}}$ to explore moves.

The random games are played using the same patterns as in MOGO [6] near the last move. If no pattern is matched near the last move, the selection of moves is the same as in CRAZY STONE [4].

Table 1 gives the results (% of wins) of 200 9×9 games (100 with black and 100 with white, with komi 7.5) against GNUGO 3.6 default level. The time limit is set to five seconds per move. The first program uses one slave, it scores 40.5 % against GNUGO. The second program uses sixteen slaves, it scores 70.5 % against GNUGO.

Table 1. Results against GNUGO for 5 seconds per move.

1 slave	40.50%
16 slaves	70.50%

Table 2 gives the results (% of wins) of 200 9×9 games (100 with black and 100 with white, with komi 7.5) for different numbers of slaves and different numbers of playouts of the parallel program against GNUGO 3.6 default level.

Table 2. Results of the program against GNUGO 3.6.

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
100,000 playouts	70.0%	69.0%	73.5%	70.0%	71.5%	65.0%	58.0%
50,000 playouts	63.5%	64.0%	65.0%	67.5%	65.5%	56.5%	51.5%
25,000 playouts	47.0%	49.5%	54.0%	56.0%	53.5%	48.5%	42.0%
12,500 playouts	47.5%	44.5%	44.0%	45.5%	45.0%	36.0%	32.0%

Table 2 can be used to evaluate the benefits from parallelizing the UCT algorithm. For example, in order to see if parallelizing on 8 slaves is more interesting than parallelizing with 4 slaves, we can compare the results of 100,000 playouts with 8 slaves (70.0%) to the results of 50,000 playouts with 4 slaves (65.0%). In this case, parallelization is beneficial since it gains 5.0% of wins against GNUGO 3.6. We compare 8 slaves with 100,000 playouts with 4 slaves with 50,000 playouts since they have close execution times (see table 3).

To determine the gain we have parallelizing with 8 slaves over not parallelizing at all, we can compare the results of 12,500 playouts with 1 slave (47.5%) to the results of 100,000 playouts with 8 slaves (70.0%). In this case it is very beneficial.

Another interesting conclusion we can draw from the table is that the interest of parallelizing starts to decrease at 32 slaves. For example 100,000 playouts with 32 slaves wins 65.0% when 50,000 playouts with 16 slaves wins 65.5%. So going from 16 slaves to 32 slaves does not help much.

Therefore, our algorithm is very beneficial until 16 slaves, but it is much less beneficial to go from 16 slaves to 32 or 64 slaves.

Table 3. Time in sec. of the first move.

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
1 slave per computer							
100,000 playouts	65.02	32.96	16.78	8.23	4.49	—	—
50,000 playouts	32.45	17.05	8.56	4.08	2.19	—	—
2 slaves per computer							
100,000 playouts	—	35.29	17.83	9.17	4.61	3.77	—
50,000 playouts	—	16.45	9.23	4.61	2.25	1.74	—
4 slaves per computer							
100,000 playouts	—	—	20.48	13.13	5.47	3.77	3.61
50,000 playouts	—	—	10.33	6.13	2.82	1.83	1.75

Table 3 gives the mean over 11 runs of the time taken to play the first move of a 9x9 game, for different numbers of total slaves, different numbers of slaves per computer, and different numbers of playouts. The values were computed on an homogeneous network of dual cores. The associated variances are very low.

We define the speedup for n slaves as the division of the time for playing the first move with one slave by the time for playing the first move with n slaves.

Table 4 gives the speedup for the different configurations and 100,000 playouts, calculated using table 3. Table 5 gives the corresponding speedups for 50,000 playouts. The speedups are almost linear until 8 slaves with one slave per computer. They start to decrease for 16 slaves (the speedup is then roughly 14), and stabilize near to 18 for more than 16 slaves.

Table 4. Time-ratio of the first move for 100,000 playouts.

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
1 slave per computer	1.00	1.97	3.87	7.90	14.48	—	—
2 slaves per computer	—	1.84	3.65	7.09	14.10	17.25	—
4 slaves per computer	—	—	3.17	4.95	11.89	17.25	18.01

Table 5. Time-ratio of the first move for 50,000 playouts.

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
1 slave per computer	1.00	1.90	3.79	7.95	14.82	—	—
2 slaves per computer	—	1.97	3.52	7.04	14.42	18.65	—
4 slaves per computer	—	—	3.14	5.29	11.51	17.73	18.54

Another conclusion we can draw from these tables is that it does not make a large difference running one slave per computer, two slaves per computer or four slaves per computer (even if processors are only dual cores).

In order to test if the decrease in speedup comes from the client or from the server, we made multiple tests. The first one consists in not playing playouts in the slaves, and sending a random value instead of the result of the playout. It reduces the time processing of each slave to almost zero, and only measures the communication time between the master and the slaves, as well as the master processing time.

The results are given in table 6. We see that the time for random results converges to 3.9 seconds when running on 32 slaves, which is close to the time taken for the slaves playing real playouts with 32 or more slaves. Therefore the 3.9 seconds limit is due to the communications and to the master processing time and not to the time taken by the playouts.

Table 6. Time in sec. of the first move with random slaves.

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves
1 slave per computer						
100,000 playouts	25.00	12.50	6.25	4.44	4.10	—
2 slaves per computer						
100,000 playouts	—	12.49	6.94	4.47	4.48	3.93
4 slaves per computer						
100,000 playouts	—	—	6.26	4.72	4.07	3.93

In order to test the master processing time, we removed the communication. We removed the send command in the master, and replaced the reception command with a random value. In this experiment the master is similar to the previous experiment, except that it does not perform any communication. Results are given in table 7. For 100,000 playouts the master processing time is 2.60 seconds, it accounts for 78% of the 3.3 seconds limit we have observed in the previous experiment.

Further speedups can be obtained by optimizing the master part, and from running the algorithm on a shared memory architecture to reduce significantly the communication time.

Table 7. Time in sec. of the random master.

100,000 playouts	2.60
50,000 playouts	1.30

Table 8 gives the time of the parallel algorithm for various numbers of slaves, with random slaves and various fixed playout times. In this experiment, a slave sends back a random evaluation when the fixed playout time is elapsed. The first column of the table gives the fixed playout time in milliseconds. The next columns gives the mean time for the first move of a 9x9 game, the numbers under parenthesis give the associated variance, each number corresponds to ten measures.

We see in table 8 that for slow playout times (greater than two milliseconds) the speedup is linear even with 32 slaves. For faster playout times the speedup degrades as the playouts get faster. For one millisecond and half a millisecond, it is linear until 16 slaves. The speedup is linear until 8 slaves for playout time as low as 0.125 milliseconds. For faster playout times it is linear until 4 slaves.

Slow playouts policies can be interesting in other domains than Go, for example in General Game Playing. Concerning Go, we made experiments with a fast playout policy, and we succeeded parallelizing it playing multiple playouts at each leaf. For 19x19 Go, playouts are slower than for 9x9 Go, therefore our algorithm should better apply to 19x19 Go.

Table 8. Time of the algorithm with random slaves and various playout times.

time	1 slave	4 slaves	8 slaves	16 slaves	32 slaves
10	1026.8 (6.606)	256.2 (0.103)	128.1 (0.016)	64.0 (0.030)	32.0 (0.020)
2	224.9 (0.027)	56.3 (0.162)	28.1 (0.011)	14.0 (0.005)	7.0 (0.002)
1	125.0 (0.081)	31.2 (0.006)	15.6 (0.001)	7.8 (0.006)	4.3 (0.035)
0.5	75.0 (0.026)	18.8 (0.087)	9.4 (0.001)	4.8 (0.034)	4.0 (0.055)
0.25	50.0 (0.005)	12.5 (0.024)	6.2 (0.001)	4.1 (0.019)	3.9 (0.049)
0.125	37.5 (0.021)	9.4 (0.001)	4.7 (0.007)	4.1 (0.222)	3.9 (0.055)
0.0625	25.0 (0.012)	6.6 (0.013)	4.4 (0.013)	4.0 (0.023)	3.8 (0.016)
0.03125	25.0 (0.007)	6.3 (0.004)	4.5 (0.110)	4.2 (0.0025)	4.0 (0.054)
0.01	25.0 (0.007)	6.3 (0.004)	4.5 (0.110)	4.5 (0.025)	4.0 (0.054)

The last experiment tests the benefits of going from 8 slaves to 16 slaves assuming linear speedups. Results are given in table 9. There is a decrease in winning percentages as we increase the number of playouts.

Table 9. Results of the 8-slaves program against 16-slaves program.

8-slaves with 50,000 playouts against 16-slaves with 100,000 playouts	33.50%
8-slaves with 25,000 playouts against 16-slaves with 50,000 playouts	27.00%
8-slaves with 12,500 playouts against 16-slaves with 25,000 playouts	21.50%

5 Conclusion

We have presented a parallel Monte-Carlo tree search algorithm. Experimental results against GNUGO 3.6 show that the improvement in level is efficient until 16 slaves. Using 16 slaves, our algorithm is 14 times faster than the sequential algorithm. On a cluster of computers the speedup varies from 4 to at least 32 depending on the playout speed. Using 5 seconds per move the parallel program improves from 40.5% with one slave to 70.5% with 16 slaves against GNUGO 3.6.

References

1. B. Bruegmann. Monte Carlo Go. Technical Report, http://citeseer.ist.psu.edu/637134.html, 1993.
2. Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep blue. *Artif. Intell.*, 134(1-2):57–83, 2002.
3. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Computer Games Workshop 2007*, pages 93–101, Amsterdam, The Netherlands, June 2007.
4. R. Coulom. Efficient selectivity and back-up operators in Monte-Carlo tree search. In *Computers and Games 2006*, Volume 4630 of LNCS, pages 72–83, Torino, Italy, 2006. Springer.
5. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
6. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.
7. Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, pages 273–280, 2007.
8. L. Kocsis and C. Szepesvàri. Bandit based Monte-Carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.