On the Parallelization of UCT

Tristan Cazenave[1] and Nicolas Jouandeau[2]

[1] Dept. Informatique cazenave@ai.univ-paris8.fr
[2] Dept. MIME n@ai.univ-paris8.fr
LIASD, Université Paris 8, 93526, Saint-Denis, France

Abstract. We present three parallel algorithms for UCT. For 9×9 Go, they all improve the results of the programs that use them against GNU GO 3.6. The simplest one, the single-run algorithm, uses very few communications and shows improvements comparable to the more complex ones. Further improvements may be possible sharing more information in the multiple-runs algorithm.

1 Introduction

Works on parallelization in games are mostly about the parallelization of the Alpha-Beta algorithm. We address here different approaches to the parallelization of the UCT algorithm.

Monte-Carlo Go has recently improved to compete with the best Go programs [3–5, 7]. We show that it can be further improved using parallelization.

Section 2 describes related work. Section 3 presents three parallel algorithms. 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 [3]. 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 [6], and it has been applied with success to Monte-Carlo Go in the program MOGO [4, 5, 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 $\mu_i + C \times \sqrt{log(t)/s}$. 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 t, and the number of games that start with move c_i is s.

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 parallel virtual machine that we have used to implement the parallel algorithms. Then we present in three separate subsections the three parallel algorithms.

3.1 The Parallel Virtual Machine

To improve search, we choose message passing as parallel programming model, which is implemented in the standard MPI, also supported by LAM/MPI [2]. Our virtual parallel computer, constituted with classical personal computer, sets up a fully connected network of computers. 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 two 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. All gtp read and write commands are realized from and to the master. Slaves satisfy computing requests. The maximum time taken by any slave task is specified during each computing request. Therefore, the communication time is added. According to time limits, the maximum time spent over all computing loops is defined by the sum of all slowest answers. We use a synchronous communication mode for data transmission, with time-constrained computing sequences. In the UCT context, as the algorithm is anytime, it is naturally well-adapted for synchronous programming.

```
MASTER_PART:
singleRunParallelUCTMove(goban[ ], color, ko, time)
    best ← −1;
    (wins[ ], games[ ]) ← initialParallelUCTMove(goban[ ], color, ko, time);
    for j ← 0 to goban.size()
        best ← max(best, wins[j]/games[j]);
    return best;

initialParallelUCTMove(goban[ ], color, ko, time)
    for i ← 0 to goban.size()
        wins[i] ← 0;
        games[i] ← 0;
    broadcast(goban[ ], color, ko, time);
    for i ← 0 to nbSlaves
        receiveUCTSequences(newWins[ ], newGames[ ]);
        for j ← 0 to goban.size()
            wins[j] ← wins[j] + newWins[j];
            games[j] ← games[j] + newGames[j];
    return (wins[ ], games[ ]);

SLAVE_PART:
singleRunParallelUCTMoveSlaveLoop()
    while(true)
        if(SingleQueryUCTSlaveLoop() == END_GAME) break;
    return;

SingleQueryUCTSlaveLoop()
    if(receive(goban[ ], color, ko, time) == END_GAME) return END_GAME;
    for i ← 0 to goban.size()
        wins[i] ← 0;
        games[i] ← 0;
    (wins[ ], games[ ]) ← playUCTSequences(goban[ ], color, ko, time);
    send(wins[ ], games[ ]);
    return CONTINUE;
```

ALG. 1: Single-Run Parallel Algorithm.

3.2 Single-Run Parallelization

The single-run parallelization consists in running multiple UCT algorithms in parallel without communication between the processes. Each process has a different seed for the random-number generator, so they do not develop the same UCT tree. When the time is over, or when the maximum number of random games is reached, each slave sends back to the master the number of games and the number of wins for all the moves at the root node of the UCT tree. The master process then simply adds the number of games and the number of wins of the moves for all the slave processes.

The master part and the slave part of the single-run parallelization are given in algorithm 1.

3.3 Multiple-Runs Parallelization

On the contrary of the single-run parallelization algorithm, the multiple-runs parallelization algorithm shares information between the processes. It consists in updating the number of games and the number of wins for all the moves at the root of the shared UCT tree every fixed amount of time. The master process starts with sending the goban, the color to move, the ko intersection and the initial thinking time to the slaves, then all the slaves start computing their UCT trees, and after the initial thinking time is elapsed, they all send the number of wins and the number of games for the root moves to the master process. Then the master process adds all the results for all the moves at the root, and sends back the information to the slaves. The slaves then initiate a new UCT computation with the same shared root moves information. The communication from the slaves to the master, the update of the master root moves information, the update of the slaves root moves information and the slaves computations are then run until the overall thinking time is elapsed. It is important to notice that at line 5 of the `multipleQuerySlaveLoop` function, the `newWins` and `newGames` arrays contain the difference between the number of wins (resp. games) after the UCT search and the number of wins (resp. games) before the UCT search.

Another important detail of the algorithm is that in the slaves, the number of wins and the number of games of the root moves are divided by the number of slaves. During the experiments of the multiple-runs algorithm, we tried not to divide, and the results were worse than the non-parallel algorithm. Dividing by the number of slaves makes UCT develop its tree in the slaves in a similar way as it would without sharing information, however the average scores of the root moves are more accurate than without sharing information. The improvement comes from the improved average scores.

The master part and the slave part of the multiple-runs parallelization are given in algorithm 2.

3.4 At-the-leaves Parallelization

At-the-leaves parallelization consists in replacing the random game at a leaf of the UCT tree with multiple random games run in parallel on the slave processes. This type of parallelization costs much more in communication time than the two previous approaches since communications between the master and the slaves occur for each new leaf of the UCT tree.

In the at-the-leaves parallelization algorithm, the master is the only one to develop the UCT tree. For each new leaf of the UCT tree, it sends to the slaves the sequence that leads from the root of the tree to the leaf. Then each slave plays a pre-defined number of random games that start with the sequence, and returns the average score of these random games. The master collects all the averages of the slaves and computes the average of the averages.

```
MASTER_PART:
multipleRunsParallelUCTMove(goban[ ], color, ko, time)
best ← −1;
(wins[ ], games[ ]) ← initialParallelUCTMove(goban[ ], color, ko,
                                              initialPassTime);
for j ← 0 to goban.size()
    | best ← max(best, wins[j]/games[j]);
time ← time − initialPassTime;
while(runPassTime < time)
    | (wins[ ], games[ ]) ← runParallelUCTMove(wins[ ], games[ ],
    |                                          runPassTime);
    | time ← time − runPassTime;
for j ← 0 to goban.size()
    | best ← max(best, wins[j]/games[j]);
return best;

runParallelUCTMove(wins[ ], games[ ], time)
broadcast(wins[ ], games[ ], time);
for i ← 0 to nbSlaves
    | receiveUCTSequences(newWins, newGames);
    | for j ← 0 to goban.size()
          wins[j] ← wins[j] + newWins[j];
          games[j] ← games[j] + newGames[j];
return (wins[ ], games[ ]);

SLAVE_PART:
multipleRunsParallelUCTMoveSlaveLoop()
while(true)
    | if(SingleQueryUCTSlaveLoop() == END_GAME) break;
    | state ← CONTINUE;
    | while(state == CONTINUE)
          state ← multipleQueryUCTSlaveLoop();
return;

multipleQueryUCTSlaveLoop()
if(receive(wins[ ], games[ ], time) == END_LOOP) return END_LOOP;
for i ← 0 to goban.size()
    | wins[i] ← wins[i]/nbSlaves;
    | games[i] ← games[i]/nbSlaves;
(newWins[ ], newGames[ ]) ← continueUCTSequences(time);
send(newWins[ ], newGames[ ]);
return CONTINUE;
```

ALG. 2: Multiple-Runs Parallel Algorithm.

```
    MASTER_PART:
AtLeavesParallelUCTMove(goban[ ], color, ko, time)
    best ← −1;
    broadcast(goban[ ], color, ko);
    while(moreTime(time)))
        | sequence[ ] ← getUCTSequence()
        | newWins ← runParallelImproveAtLeaves(sequence[ ]);
        | for nodeId in sequence[ ]
              wins[nodeId] ← wins[nodeId] + newWins;
              games[nodeId] ← games[nodeId] + 1;
    for j ← 0 to goban.size()
        | best ← max(best, wins[j]/games[j]);
    return best;

runParallelImproveAtLeaves(sequence[ ])
    broadcast(sequence[ ]);
    improvedWins ← 0;
    for i ← 0 to nbSlaves
        | receive(nodeWins);
        | improvedWins ← improvedWins + nodeWins;
    return improvedWins/nbSlaves;

    SLAVE_PART:
atLeavesParallelSlaveLoop()
    while(true)
    if(receive(goban[ ], color, ko) == END_GAME) break;
        | state ← CONTINUE;
        | while(state == CONTINUE)
              state ← atLeavesQuerySlaveLoop();
    return;

atLeavesQuerySlaveLoop()
    if(receive(sequence[ ]) == END_LOOP) return END_LOOP;
    for i ← 0 to sequence.size()
        | playMove(sequence[i]);
    nodeWins ← 0
    for i ← 0 to nbGamesAtLeaf
        | newNodeWins ← playRandomGame();
        | nodeWins ← nodeWins + newNodeWins;
    send(nodeWins/nbGameAtLeaf);
    return CONTINUE;
```

ALG. 3: At-The-Leaves Parallel Algorithm.

The master part and the slave part of the at-the-leaves parallelization are given in algorithm 3.

4 Experimental Results

Tests are run on a simple network of computers running LINUX 2.6.18. The network includes 100 Mb switches. The BogoMips rating of each node is approximately 6000.

In our experiments, UCT uses $\mu_i + \sqrt{\frac{log(t)}{10 \times s}}$ to explore moves.

The random games are played using the same patterns as in MOGO [7] near the last move. If no pattern is matched near the last move, the selection of moves is the same as in CRAZY STONE [3].

Table 1 gives the results (% of wins) of 200 9×9 games (100 with black and 100 with white) for the single-run parallel program against GNU GO 3.6 default level. The parallel algorithm has been tested with either 3,000 simulations (random games) for each UCT search, or 10,000 simulations. The single-run parallelization improves the result bringing them from 27.5% for 1 CPU and 3,000 games to 53.0% for 16 CPUs and 3,000 games per CPU. Concerning the experiments with 10,000 games per CPU, the results increase from 45.0% for 1 CPU to 66.5% for 16 CPUs. For the single-run parallelization, the communication time is very small compared to the computation time. The single-run parallelization successfully improves the UCT algorithm.

Table 1. Results of the single-run program against GNU GO 3.6.

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
3,000 simulations	27.5%	40.0%	48.0%	55.5%	53.0%
10,000 simulations	45.0%	62.0%	61.5%	65.0%	66.5%

Table 2 gives the results of 200 9×9 games for the multiple-runs parallel program against GNU GO 3.6 default level. In these experiments, the multiple-runs algorithms updates the shared information every 250 simulations. The results are similar to the results of the single-run parallelization. They are slightly better with 10,000 simulations.

Table 2. Results of the multiple-runs program against GNU GO 3.6.

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
3,000 simulations	20.0%	32.0%	48.0%	53.5%	56.0%
10,000 simulations	49.0%	58.5%	72.0%	72.0%	68.0%

Table 3 gives the results of 200 9×9 games for the at-the-leaves parallel program against GNU GO 3.6 default level. We can see an improvement when the number of

CPUs increases from 1 to 8. However increasing the number to more than 8 does not improve much as can be seen from the second line of the table, where the slaves all play 8 games per leaf. Playing 8 games per leaf is equivalent to having 8 times more CPUs with one game per leaf.

Concerning the 10,000 simulations experiments, the percentage of wins also increases until 8 CPUs.

Table 3. Results of the at-the-leaves parallel program against GNU GO 3.6.

	nbGamesAtLeaf	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
3,000 simulations	1	21.0%	35.0%	42.0%	46.0%	45.0%
3,000 simulations	8	54.5%	48.5%	49.5%	47.5%	51.0%
10,000 simulations	1	47.0%	53.5%	53.5%	69.5%	62.0%

Table 4 shows the communication overhead for the at-the-leaves parallel program. For one CPU, the slave process runs on the same machine as the master process, so the communication time is small and the time used to find the first move on an empty 9×9 goban can be used as a reference for a non-parallel program. We see in the next columns that the communication time progressively increases, doubling the thinking time for 8 CPUs.

Table 4. Times for the first move for the at-the-leaves parallel program.

	nbGamesAtLeaf	1 CPU	2 CPUs	4 CPUs	8 CPUs
10,000 simulations	1	6.21s.	10.85s.	11.56s.	13.07s.

The communication time for the at-the-leaves parallel program is significantly higher than the two previous algorithms. Given that it gives similar improvements in level, it is preferable to use the single-run or multiple-runs algorithms. The at-the-leaves parallelization could be of interest to multiple CPUs machines with a shared memory where the communication costs are less of a problem.

5 Conclusion

We have presented three parallel algorithms that improve UCT search. They all give similar improvements. The single-run parallelization is the most simple one and also the one that uses the fewest communications between the processes.

The at-the-leaves parallelization currently costs too much communications, however it could still be interesting on a multiple CPUs machine.

We believe that the multiple-runs algorithm can be further improved to share more information and then may become the best algorithm.

References

1. Bruegmann, B.: Monte Carlo Go. ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z (1993)
2. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium. (1994) 379–386
3. Coulom, R.: Efficient selectivity and back-up operators in monte-carlo tree search. In: Proceedings Computers and Games 2006, Torino, Italy (2006)
4. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in Monte-Carlo go. Technical Report 6062, INRIA (2006)
5. Gelly, S., Wang, Y.: Exploration exploitation in go: UCT for Monte-Carlo go. In: NIPS-2006: On-line trading of Exploration and Exploitation Workshop, Whistler Canada (2006)
6. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: ECML-06. Number 4212 in LNCS, Springer (2006) 282–293
7. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo go. In: CIG 2007, Honolulu, USA (2007) 175–182