A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm

Markus Enzenberger and Martin Müller

Department of Computing Science, University of Alberta
Corresponding author: mmueller@cs.ualberta.ca

Abstract. With the recent success of Monte-Carlo tree search algorithms in Go and other games, and the increasing number of cores in standard CPUs, the efficient parallelization of the search has become an important issue. We present a new lock-free parallel algorithm for Monte-Carlo tree search which takes advantage of the memory model of the IA-32 and Intel-64 CPU architectures and intentionally ignores rare faulty updates of node values. We show that this algorithm significantly improves the scalability of the Fuego Go program.

1 Introduction

1.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) has proved to be a successful search method in two-player board games for which it is difficult to create a good heuristic evaluation function. In the game of Go, it was first used by the programs Crazy Stone [1] and MoGo [2], and has led to programs that for the first time have reached human master level, especially on smaller board sizes.

In MCTS, a search tree is stored in memory and expanded incrementally. Each simulated game starts with an *in-tree phase*, in which moves are selected following a sequence of nodes from the root node and choosing a child in each node based on the current value of the child and potentially additional exploration bonuses. After a node with no children is reached, it is expanded, and the game is finished using a *playout phase* in which moves are generated by a more or less randomized move generation policy. After the playout, there is an *update phase*, in which the values of the nodes used in the game are updated with the game result, such that they store the average result of all games in which the move was chosen. The simulation process is repeated until a resource limit is exceeded, for example the number of simulations, time used, or memory.

1.2 Parallel Monte-Carlo Tree Search

MCTS has the additional advantage that it is easier to parallelize than traditional programs based on alpha-beta search [3–7]. The most common methods have been classified by Chaslot et al. [5] as *leaf parallelization*, *root parallelization*, and *tree parallelization*. On shared memory systems, tree parallelization is the natural method that takes full advantage of the available bandwidth for communicating game results. In this method, several threads run simulations in parallel and share a common tree in memory. Usually, a global mutex is used for protecting access and modification of the tree during the in-tree and the update phase, whereas the playout phase proceeds independently in each thread and does not require locking. How well tree parallelization scales with the number of threads depends, among other things, on the ratio of the time spent in the unlocked playout phase to the total time for a game.

Chaslot et al. [5] have shown that in their Go program Mango, tree parallelization only scales well up to four threads. Their experiments were done in 9×9 Go, in which the playout phase is shorter and the in-tree phase longer than in 19×19 Go. They also tried a more fine-grained locking algorithm, which reduced the overhead of the global mutex at the price of increasing the node size in the tree by adding a local mutex to each node. The problem here is that, due to the selectivity of MCTS, usually a large number of nodes are shared in the in-tree move sequences of the different threads; at least one node, the root node, is always shared. Therefore using local mutexes is not necessarily an improvement. However, they showed that the addition of a *virtual loss* improved the scaling of tree parallelization in Mango.

In messages to the Computer Go mailing list [8], Coulom reports strong results with a lock-free transposition table in Crazy Stone. Another possible approach, also implemented in Crazy Stone, uses *spinlocks* [8], which are a form of busy waiting. Spinlocks avoid the overhead of other locking approaches. For a relatively small number of threads, the speed of spinlocks might be comparable to the lock-free approach presented here. However, an extra variable per node is needed to hold the spinlock.

1.3 The Fuego Go Program

Fuego is an open-source Go program developed by the Computer Go group at the University of Alberta [9]. On the Bayes-Elo ranking of the Computer Go Server [10] from January 15 2009 (16:03 UCT for 19×19; 18:19 UCT for 9×9), it is ranked as the 3rd-strongest program ever on 9×9 with a rating of 2664 Elo. On 19×19 it is the 2nd-strongest program ever with a rating of 2290 Elo. The program is written in C++ and separated into different libraries. The MCTS implementation is in the game-independent SmartGame library and is also used by external projects for other games, such as MoHex, a Hex program by the Games Group at the University of Alberta [11].

The main Go player in Fuego uses full-board MCTS with a number of common enhancements to the basic MCTS algorithm. Move generation in the playout phase is similar to the one originally used by MoGo; it uses patterns, liberty and locality heuristics. In the in-tree phase, children are selected based on the Rapid Action Value Estimation (RAVE) heuristic [2], which combines the current value of a move with the average value of this move in the subtree of the corresponding node. When a node is expanded, the values and counts of new children are initialized based on a static heuristic evaluation. Parallel search is supported using tree parallelization.

2 Lock-free Multithreaded MCTS

The basic idea of Fuego's lock-free multithreaded MCTS algorithm is to share a tree between multiple threads without using any locks. Because of specific requirements on the memory model of the hardware platform, this lock-free mode is an optional feature of the base MCTS class in Fuego and needs to be enabled explicitly.

2.1 Modifying the Tree Structure

The first change to make the lock-free search work is in the handling of concurrent changes to the structure of the tree. Fuego never deletes nodes during a search; new nodes are created in a pre-allocated memory array. In the lock-free algorithm, each thread has its own memory array for creating new nodes. Only after the nodes are fully created and initialized, are they linked to the parent node. This can cause some memory overhead, because if several threads expand the same node only the children created by the last thread will be used in future simulations. It can also happen that some of the children that are lost already received value updates; these updates will be lost.

The child information of a node consists of two variables: a pointer to the first child in the array, and the number of children. To avoid that another thread sees an inconsistent state of these variables, all threads assume that the number of children is valid if the pointer to the first child is not null. Linking a parent to a new set of children requires first writing the number of children, then the pointer to the first child. The compiler is prevented from reordering the writes by declaring these variables using the C++ type qualifier *volatile*.

2.2 Updating Values

The move and RAVE values are stored in the nodes as counts and mean values. The mean values are updated using an incremental algorithm. Updating them without protection by a mutex can cause updates of the mean to be lost with or without increment of the count, as well as updates of the mean occurring without increment of the count. It could also happen that one thread reads the count and mean while they are written by another thread, and the first thread sees an erroneous state that exists only temporarily. In practice, these faulty updates occur with a low probability and will have only a small effect on the counts and mean values. They are intentionally ignored.

The only problematic case is if a count is zero, because the mean value is undefined if the count is zero, and this case has a special meaning at several places in the search. For example, the computation of the values for the selection of children in the in-tree phase distinguishes three cases: if the move count and RAVE count is non-zero, the value will be computed as a weighted linear combination of both mean values, if the move count is zero, only the RAVE mean is used, and if both counts are zero, a configurable constant value, the *first play urgency*, is used. To avoid this problem, all threads assume that a mean value is only valid if the corresponding count is non-zero. Updating a value requires first writing the new mean value, then the new count. Again, *volatile* is used to protect the order of writes.

2.3 Platform Requirements

There are some requirements on the memory model of the platform to make the lock-free search algorithm work. Writes of the basic types *size_t, int, float* and *pointer* must be atomic. Writes by one thread must be seen by other threads in the same order. The IA-32 and Intel-64 CPU architectures, which are used in most modern standard computers, guarantee these assumptions. They also synchronize CPU caches after writes [12].

3 Experiments

The experiments compare how well locked and lock-free searches scale with the number of threads in Fuego in 9×9 and 19×19 Go. The comparison is against the ideal case represented by running the singlethreaded program n times longer.

3.1 Setup

The version of Fuego was 0.3, which was released on 17 December 2008. The hardware was an Intel Xeon E5420 2.5 GHz dual quadcore system with 8 GB main memory and a 64-bit version of the GNU/Linux operating system. On this hardware, Fuego achieves about 11,400 simulations per second per core if a search is started on an empty 9×9 board. About 53 percent of the simulation time is spent in the playout phase. On 19×19, the program achieves 2750 simulations on an empty board and spends about 69 percent of the simulation time in playouts. The maximum tree size was set to 20,000,000 nodes. Although Fuego implements the virtual loss enhancement [5] as an option, it is disabled by default and was not used in the experiment.

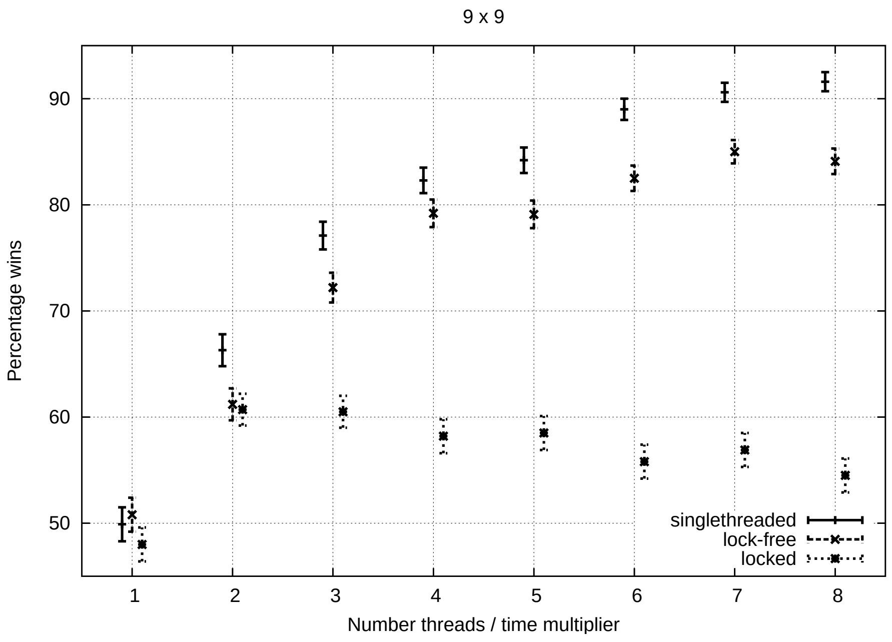
The self-play experiments were performed against a fixed opponent, the singlethreaded version set to 1 sec per move. Three series of runs measured the percentage of wins against the standard version using a single-threaded version with n times more time per move, as well as locked and lock-free multithreaded versions with 1 sec per move and n threads. A total of 1000 games with Chinese rules and alternating player colors were played for each data point. The opening book was disabled. The games were played using the gogui-twogtp program included in the GoGui distribution [13]. GNU Go version 3.6 [14] was used as a referee for determining the result of a game.

3.2 Results

The results of the experiment are shown in Fig. 1. The error bars in the figure correspond to one standard error.

The version using a global mutex does not scale beyond two threads on 9×9 and three on 19×19. On 9×9, this is even less than what was reported by Chaslot et al. for their program Mango, which still showed an improvement in playing strength with up to four threads.

The lock-free version scales up to seven threads on both board sizes. The playing strength with eight threads is slightly less than with seven, although one cannot say with high confidence whether this is a real effect given the statistical error of the experiment.



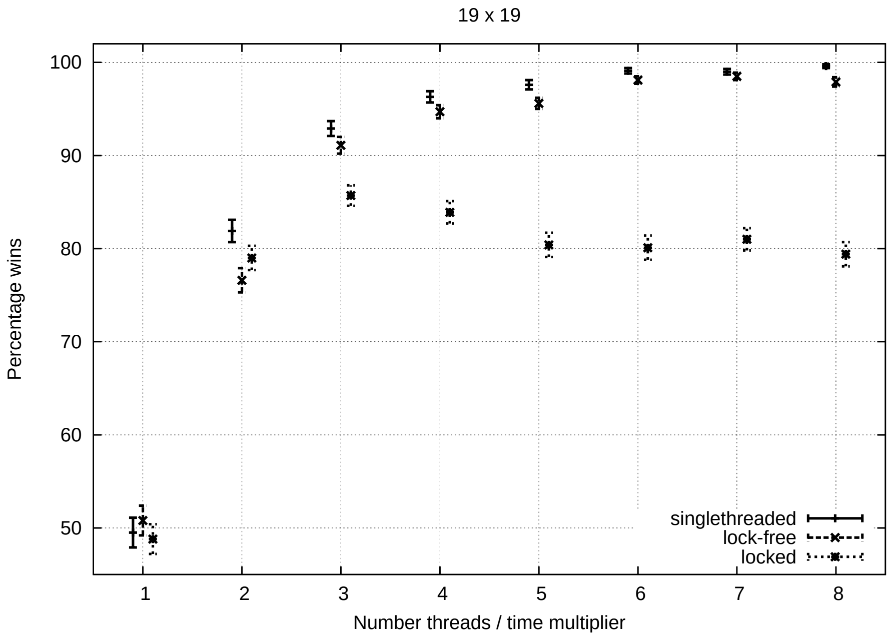


Fig. 1. Self-play performance of locked and lock-free multithreading in comparison to a single-threaded search (1 s per move)

4 Conclusion and Future Work

A lock-free multithreaded algorithm for MCTS can significantly improve the scaling of MCTS in 9×9 and 19×19 Go. Future experiments should investigate scaling with more than eight cores, and applications to other games.

Modifications of the algorithm are possible that allow it to be used on more CPU architectures. If an architecture does not guarantee that writes by one thread are seen by other threads in the same order and caches are synchronized after writes, then it might still be better to use explicit memory barriers at the places with write order dependencies than to use a global mutex for the whole in-tree and update phases.

References

1. R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In J. van den Herik, P. Ciancarini, and H. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630/2007 of *Lecture Notes in Computer Science*, pages 72–83, Turin, Italy, June 2006. Springer.
2. S. Gelly. *A Contribution to Reinforcement Learning; Application to Computer-Go*. PhD thesis, Université Paris-Sud, 2007.
3. T. Cazenave and N. Jouandeau. A parallel Monte-Carlo tree search algorithm. In J. van den Herik, X. Xu, Z. Ma, and M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 2008.
4. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Computer Games Workshop*, pages 93–101, Amsterdam, 2007.
5. G. Chaslot, M. Winands, and J. van den Herik. Parallel Monte-Carlo tree search. In *Proceedings of the 6th International Conference on Computer and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008.
6. S. Gelly, J. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. On the parallelization of Monte-Carlo planning. In *Icinco*, pages 198–203, Madeira, Portugal, 2008.
7. H. Kato and I. Takeuchi. Parallel Monte-Carlo Tree Search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, 2008.
8. R. Coulom. Lockless hash table and other parallel search ideas. `http://computer-go.org/pipermail/computer-go/2008-March/014537.html` and `http://computer-go.org/pipermail/computer-go/2008-March/014547.html`, 2008. Date retrieved: April 28, 2009.
9. M. Enzenberger and M. Müller. Fuego – an open-source framework for board games and go engine based on monte-carlo tree search. Technical Report TR09-08, University of Alberta, Edmonton, 2009. 16 pages.
10. D. Dailey. Computer Go Server. `http://cgos.boardspace.net/`, 2008. Date retrieved: January 19, 2009.
11. B. Arneson, R. Hayward, and P. Henderson. Wolve wins Hex tournament. `http://www.cs.ualberta.ca/~hayward/papers/rptBeijing.pdf`, 2008. To appear in ICGA Journal.
12. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3A: System Programming Guide, Part 1*, 2008. Order Number: 253668-029US.
13. M. Enzenberger. GoGui. `http://gogui.sf.net/`, 2009. Date retrieved: January 2, 2009.
14. Free Software Foundation. GNU Go. `http://www.gnu.org/software/gnugo/`, 2009. Date retrieved: January 2, 2009.