Parallel Monte-Carlo Tree Search

Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik

Games and AI Group, MICC, Faculty of Humanities and Sciences,
Universiteit Maastricht, Maastricht, The Netherlands
{g.chaslot,m.winands,herik}@micc.unimaas.nl

Abstract. Monte-Carlo Tree Search (MCTS) is a new best-first search method that started a revolution in the field of Computer Go. Parallelizing MCTS is an important way to increase the strength of any Go program. In this article, we discuss three parallelization methods for MCTS: *leaf parallelization*, *root parallelization*, and *tree parallelization*. To be effective tree parallelization requires two techniques: adequately handling of (1) *local mutexes* and (2) *virtual loss*. Experiments in 13×13 Go reveal that in the program MANGO root parallelization may lead to the best results for a specific time setting and specific program parameters. However, as soon as the selection mechanism is able to handle more adequately the balance of exploitation and exploration, tree parallelization should have attention too and could become a second choice for parallelizing MCTS. Preliminary experiments on the smaller 9×9 board provide promising prospects for tree parallelization.

1 Introduction

For decades, the standard method for two-player games such as chess and checkers has been $\alpha\beta$ search. Nevertheless, in 2006 Monte-Carlo Tree Search (MCTS) [7, 4, 12, 8, 10, 6] started a revolution in the field of Computer Go. At this moment (January 2008) the best MCTS 9×9 Go programs are ranked 500 rating points higher than the traditional programs on the Computer Go Server [2]. On the 19×19 Go board, MCTS programs are also amongst the best programs. For instance, the MCTS program MOGO won the Computer Olympiad 2007 [9], and the MCTS program CRAZY STONE has the highest rating amongst programs on the KGS Go Server [1].

MCTS is not a classical tree search followed by a Monte-Carlo evaluation, but rather a best-first search guided by the results of Monte-Carlo simulations. Just as for $\alpha\beta$ search [11], it holds for MCTS that the more time is spent for selecting a move, the better the game play is. Moreover, the law of diminishing returns that nowadays has come into effect for many $\alpha\beta$ chess programs, appears to be less of an issue for MCTS Go programs. Hence, parallelizing MCTS seems to be a promising way to increase the strength of a Go program. Pioneer work has been done by Cazenave and Jouandeau [3] by experimenting with two parallelization methods: leaf parallelization and root parallelization (original called single-run parallelization).

In this article we introduce a third parallelization method, called tree parallelization. We compare the three parallelization methods (leaf, root, and tree) by using the *Games-Per-Second (GPS) speedup measure* and *strength-speedup measure*. The first measure corresponds to the improvement in speed, and the second measure corresponds to the improvement in playing strength. The three parallelization methods are implemented and tested in our Go program MANGO [5] (mainly designed and constructed by Guillaume Chaslot), running on a multi-core machine containing 16 cores. An earlier version of the program - using more modest hardware - participated in numerous international competitions in 2007, on board sizes 13×13 and 19×19. It was ranked in the first half of the participants at all these events [9, 1].

The article is organized as follows. In Sect. 2 we present the basic structure of MCTS. In Sect. 3, we discuss the different methods used to parallelize an MCTS program. We empirically evaluate the three parallelization methods in Sect. 4. Section 5 provides the conclusions and describes future research.

2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [7, 12] is a best-first search method that does not require a positional evaluation function. It is based on randomized explorations of the search space. Using the results of previous explorations, the algorithm gradually grows a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves.

MCTS consists of four strategic phases, repeated as long as there is time left. The phases are as follows. (1) In the *selection step* the tree is traversed from the root node until it selects a leaf node L that is not added to the tree yet.[1] (2) Subsequently, the *expansion strategy* is called to add the leaf node L to the tree. (3) A *simulation strategy* plays moves in a self-play mode until the end of the game is reached. The result R of such a "simulated" game is $+1$ in case of a win for Black (the first player in Go), 0 in case of a draw, and -1 in case of a win for White. (4) A *backpropagation strategy* propagates the results R through the tree, i.e., in each node traversed the average result of the simulations is computed. The four phases of MCTS are shown in Fig. 2.

When all the time is consumed, the move played by the program is the root child with the highest visit count. It might be noticed that MCTS can be stopped anytime. However, the longer the program runs, the stronger the program plays. We show in Sect. 4 that the rating of our program increases nearly linearly with the logarithm of the time spent.

[1] Examples of such a strategy are UCT, OMC, BAST, etc. [4, 7, 12, 6]. All experiments have been performed with the UCT algorithm [12] using a coefficient C_p of 0.35.

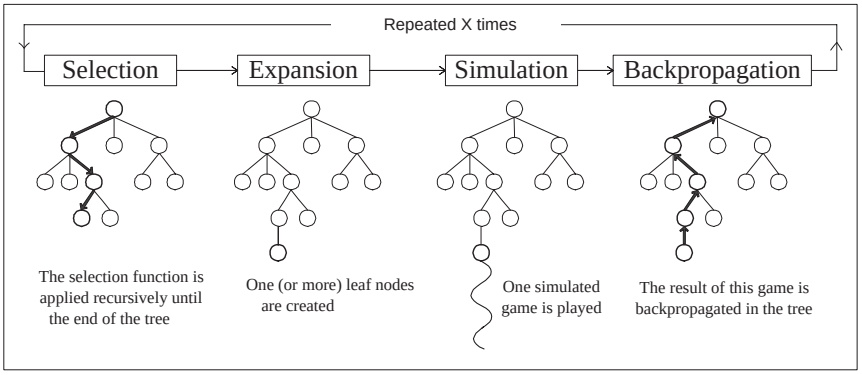

Fig. 1. Scheme of Monte-Carlo Tree Search

3 Parallelization of Monte-Carlo Tree Search

In this article, we consider parallelization for a symmetric multiprocessor (SMP) computer. We always use one processor thread for each processor core. One of the properties of a SMP computer is that any thread can access the central (shared) memory with the same (generally low) latency. As a consequence, parallel threads should use a mutual exclusion (mutex) mechanism in order to prevent any data corruption, due to simultaneous memory access. This could happen when several threads are accessing the MCTS tree (i.e., in phase 1, 2 or 4). However, the simulation phase (i.e., phase 3) does not require any information from the tree. There, simulated games can be played completely independently from each other. This specific property of MCTS is particularly interesting for the parallelization process. For instance, it implies that long simulated games make the parallelization easier. We distinguish three main types of parallelization, depending on which phase of the Monte-Carlo Tree Search is parallelized: leaf parallelization, root parallelization, and tree parallelization.

3.1 Leaf Parallelization

Leaf parallelization introduced by Cazenave and Jouandeau [3] is one of the easiest ways to parallelize MCTS. To formulate it in machine-dependent terms, only one thread traverses the tree and adds one of more nodes to the tree when a leaf node is reached (phase 1 and 2). Next, starting from the *leaf* node, independent simulated games are played for each available thread (phase 3). When all games are finished, the result of all these simulated games is propagated backwards through the tree by one single thread (phase 4). Leaf parallelization is depicted in Fig. 2a.

Leaf parallelization seems interesting because its implementation is easy and does not require any mutexes. However, two problems appear. First, the time required for a simulated game is highly unpredictable. Playing n games using n

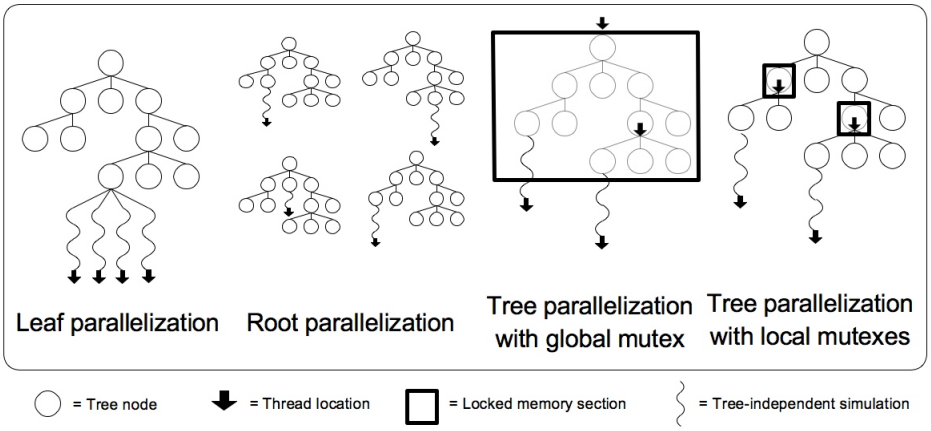

Fig. 2. (a) Leaf parallelization (b) Root parallelization (c) Tree parallelization with global mutex (d) and with local mutexes

different threads takes more time in average than playing one single game using one thread, since the program needs to wait for the longest simulated game. Second, information is not shared. For instance, if 16 threads are available, and 8 (faster) finished games are all losses, it will be highly probable that most games will lead to a loss. Therefore, playing 8 more games is a waste of computational power. To decrease the waiting time, the program might stop the simulations that are still running when the results of the finished simulations become available. This strategy would enable the program to traverse the tree more often, but some threads would be idle. Leaf parallelization can be performed inside an SMP environment, or even on a cluster using MPI (Message Passing Interface) communication.

3.2 Root Parallelization

Cazenave proposed a second parallelization under the name "single-run" parallelization [3]. In this article we will call it *root parallelization* to stress the part of the tree for which it applies. The method works as follows. It consists of building multiple MCTS trees in parallel, with one thread per tree. Similar to leaf parallelization, the threads do not share information with each other. When the available time is spent, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. The best move is selected based on this grand total. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy, even on a cluster. Root parallelization is depicted in Fig. 2b.

3.3 Tree Parallelization

In this article we introduce a new parallelization method called *tree parallelization*. This method uses one shared tree from which several simultaneous games are played. Each thread can modify the information contained in the tree; therefore mutexes are used to lock from time to time certain parts of the tree to prevent data corruption. There are two methods to improve the performance of tree parallelization: (1) mutex location and (2) "virtual loss".

Mutex location. Based on the location of the mutexes in the tree, we distinguish two mutex location methods: (1) using a *global mutex* and (2) using several *local mutexes*.

The global mutex method locks the whole tree in such a way that only *one* thread can access the search tree at a time (phase 1, 2, and 4). In the meantime several other processes can play simulated games (phase 3) starting from *different* leaf nodes. This is a major difference with leaf parallelization where all simulated games start from the *same* leaf node. The global mutex method is depicted in Fig. 2c. The potential speedup given by the parallelization is bounded by the time that has to be spent in the tree. Let x be the average percentage of time spent in the tree by one single thread. The maximum speedup in terms of games per second is $100/x$. In most MCTS programs x is relatively high (say between 25 to 50%), limiting the maximum speedup substantially. This is the main disadvantage of this method.

The local mutexes method makes it possible that *several* threads can access the search tree simultaneously. To prevent data corruption because two (or more) threads access the same node, we lock a node by using a local mutex when it is visited by a thread. At the moment a thread departs the node, it is unlocked. Thus, this solution requires to frequently lock and unlock parts of the tree. Hence, fast-access mutexes such as spinlocks have to be used to increase the maximum speedup. The local mutexes method is depicted in Fig. 2d.

Virtual loss. If several threads start from the root at the same time, it is possible that they traverse the tree for a large part in the same way. Simulated games might start from leaf nodes, which are in the neighborhood of each other. It can even happen that simulated games begin from the same leaf node. Because a search tree typically has millions of nodes, it may be redundant to explore a rather small part of the tree several times. Coulom[2] suggests to assign one "virtual loss" when a node is visited by a thread (i.e., in phase 1). Hence, the value of this node will be decreased. The next thread will only select the same node if its value remains better than its siblings' values. The virtual loss is removed when the thread that gave the virtual loss starts propagating the result of the finished simulated game (i.e., in phase 4). Owing to this mechanism, nodes that are clearly better than others will still be explored by all threads, while nodes for which the value is uncertain will not be explored by more than

[2] Personal Communication

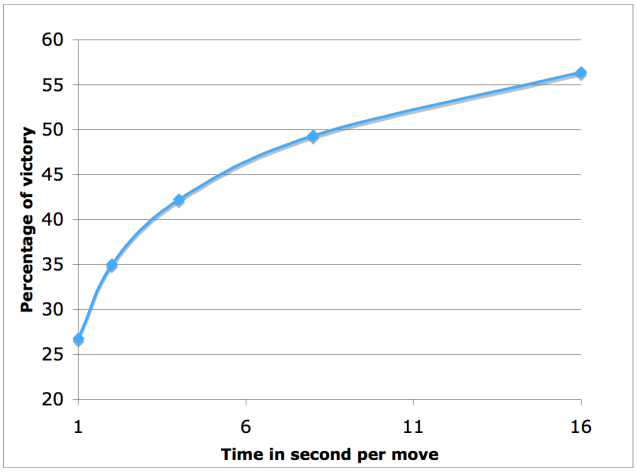

Fig. 3. Scalability of the strength of MANGO with time

one thread. Hence, this method keeps a certain balance between exploration and exploitation in a parallelized MCTS program.

4 Experiments

In this section we compare the different parallelization methods with each other. Subsection 4.1 discusses the experimental set-up. We show the performance of leaf parallelization, root parallelization, and tree parallelization in Subsection 4.2, 4.3, and 4.4, respectively. An overview of the results is given in Subsection 4.5. Root parallelization and tree parallelization are compared under different conditions in Subsection 4.6.

4.1 Experimental Set-up

The aim of the experiments is to measure the quality of the parallelization process. We use two measures to evaluate the speedup given by the different parallelization methods. The first measure is called the *Games-Per-Second (GPS) speedup*. It is computed by dividing the number of simulated games per second performed by the multithreaded program, by the number of games per second played by a single-threaded program. However, the GPS speedup measure might be misleading, since it is not always the case that a faster program is stronger. Therefore, we propose a second measure: called *strength-speedup*. It corresponds to the *increase of time needed to achieve the same strength*. For instance, a multithreaded program with a strength-speedup of 8.5 has the same strength as a single-threaded program, which consumes 8.5 times more time.

In order to design the strength-speedup measurement, we proceed in three steps. First, we measure the strength of our Go program MANGO on the 13×13 board against GNU GO 3.7.10, level 0, for 1 second, 2 seconds, 4 seconds, 8

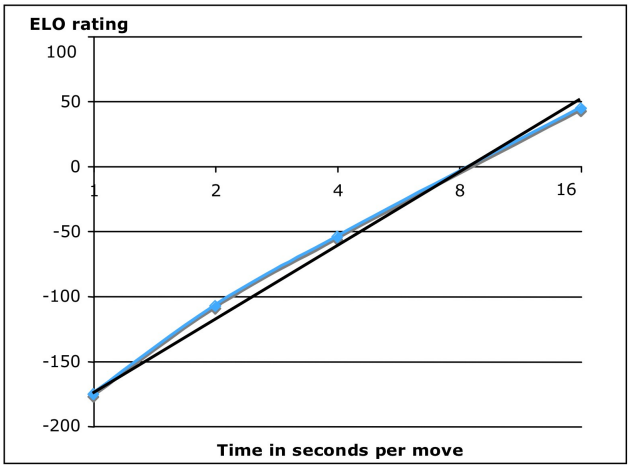

Fig. 4. Scalability of the rating of MANGO vs. GNU GO with time. The curve represents the data points, and the line is a trend-line for this data

seconds, and 16 seconds. For each time setting, 2,000 games are played. Figure 3 reports the strength of MANGO in terms of percentage of victory. In Fig. 4, the increase in strength in term of rating points as a function of the logarithmic time is shown. This function can be approximated accurately by linear regression, using a correlation coefficient $R^2 = 0.9922$. Second, the linear approximation is used to give a theoretical Go rating for any amount of time. Let us assume that E_t is the level of the program in rating points, T is the time in seconds per move. Linear regression gives us $E_t(T) = A \cdot log_2 T + B$ with $A = 56.7$ and $B = -175.2$. Third, the level of play of the multithreaded program is measured against the same version of GNU GO, with one second per move. Let E_m be the rating of this program against GNU GO. The strength-speedup S is defined by: $S \in \mathbb{R} | E_t(S) = E_m$.

The experiments were performed on the supercomputer Huygens, which has 120 nodes, each with 16 cores POWER5 running at 1.9 GHz and having 64 Gigabytes of memory per node. Using this hardware the single-threaded version of MANGO was able to perform 3,400 games per second in the initial board position of 13×13 Go. The time setting used for the multithreaded program was 1 second per move.

4.2 Leaf Parallelization

In the first series of experiments we tested the performance of plain leaf parallelization. We did not use any kind enhancement to improve this parallelization method as discussed in Subsection 3.1. The results regarding winning percentage, GPS speedup, and strength-speedup for 1, 2, 4, and 16 threads are given

Table 1. Leaf parallelization

Number of threads	Winning percentage	Number of games	Confidence interval	GPS Speedup	Strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	26.8 %	2000	2.0 %	1.8	1.2
4	32.0 %	1000	2.8 %	3.3	1.7
16	36.5 %	500	4.3 %	7.6	2.4

in Table 1. We observed that the GPS speedup is quite low. For instance, when running 4 simulated games in parallel, finishing all of them took 1.15 times longer than finishing just 1 simulated game. For 16 threads, it took two times longer to finish all games compared by finishing just one. The results show that the strength-speedup obtained is rather low as well (2.4 for 16 processors). So, we may conclude that plain leaf parallelization is not a good way for parallelizing MCTS.

4.3 Root Parallelization

In the second series of experiments we tested the performance of root parallelization. The results regarding winning percentage, GPS speedup, and strength-speedup for 1, 2, 4, and 16 threads are given in Table 2.

Table 2 indicates that root parallelization is a quite effective way of parallelizing MCTS. One particularly interesting result is that, for four processor threads, the strength-speedup is significantly higher than the number of threads used (6.5 instead of 4). This result implies that, in our program Mango, it is more efficient to run four independent MCTS searches of one second than to run one large MCTS search of four seconds. It might be that the algorithm stays for quite a long time in local optima. This effect is caused by the UCT coefficient setting. For small UCT coefficients, the UCT algorithm is able to search more deeply in the tree, but also stays a longer time in local optima. For high coefficients, the algorithm escapes more easily from the local optima, but the resulting search is shallower. The optimal coefficient for a specific position can only be determined experimentally. The time setting also influences the scalability of the results. For a short time setting, the algorithm is more likely to spend too much time in local optima. Hence, we believe that with higher time settings, root parallelization will be less efficient. In any case, we may conclude that root parallelization is a simple and effective way to parallelize MCTS.

Table 2. Root parallelization

Number of threads	Winning Percentage	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7 %	2000	2.2 %	1	1.0
2	38.0 %	2000	2.2 %	2	3.0
4	46.8 %	2000	2.2 %	4	6.5
16	56.5 %	2000	2.2 %	16	14.9

Table 3. Tree parallelization with global mutex

Number of threads	Percentage of victory	Number of games	Confidence interval	GPS speedup	strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	31.3 %	2000	2.2 %	1.8	1.6
4	37.9 %	2000	2.2 %	3.2	3.0
16	36.5 %	500	4.5 %	4.0	2.6

4.4 Tree Parallelization

In the third series of experiments we tested the performance of tree parallelization. Below, we have a closer look at the *mutexes location* and *virtual loss*.

Mutexes location. First, the global mutex method was tested. The results are given in Table 3. These results show that the strength-speedup obtained up to 4 threads is satisfactory (i.e., strength-speedup is 3). However, for 16 threads, this method is clearly insufficient. The strength-speedup drops from 3 for 4 threads to 2.6 for 16 threads. So, we may conclude that the global mutex method should not be used in tree parallelization.

Next, we tested the performance for the local mutexes method. The results are given in Table 4. Table 4 shows that for each number of threads the local mutexes has a better strength-speed than global mutex. Moreover, by using local mutexes instead of global mutex the number of games played per second is doubled when using 16 processor threads. However, the strength-speedup for 16 processors threads is just 3.3. Compared to the result of root parallelization (14.9 for 16 threads), this result is quite disappointing.

Using virtual loss. Based on the previous results we extended the global mutexes tree parallelization with the virtual loss enhancement. The results of using virtual loss are given in Table 5.

Table 5 shows that the effect of the virtual loss when using 4 processor threads is moderate. If we compare the strength-speedup of Table 4 we see an increase from 3.0 to 3.6. But when using 16 processor threads, the result is more impressive. Tree parallelization with virtual loss is able to win 49.9% of the games instead of 39.9% when it is not used. The strength-speedup of tree

Table 4. Tree parallelization with local mutex

Number of threads	Percentage of victory	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	32.9 %	2000	2.2 %	1.9	1.9
4	38.4 %	2000	2.2 %	3.6	3.0
16	39.9 %	500	4.4 %	8.0	3.3

Table 5. Using virtual loss for tree parallelization

Number of threads	Winning percentage	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	33.8 %	2000	2.2 %	1.9	2.0
4	40.2 %	2000	2.2 %	3.6	3.6
16	49.9 %	2000	2.2 %	9.1	8.5

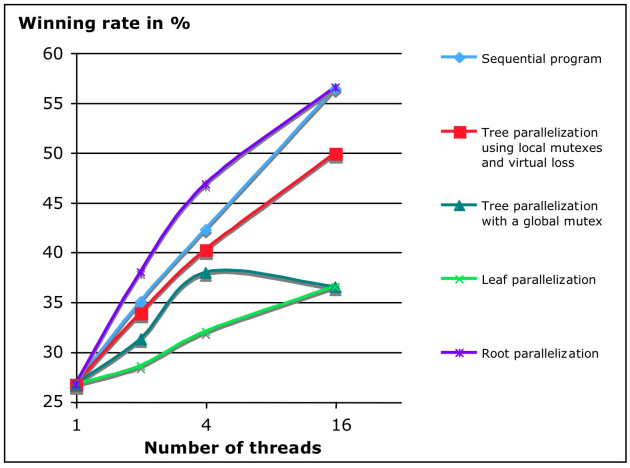

Fig. 5. Performance of the different parallelization methods

parallelization increases from 3.3 (see Table 4) to 8.5. Thus, we may conclude that virtual loss is important for the performance of tree parallelization when the number of processor threads is high.

4.5 Overview

In Fig. 5 we have depicted the performance of leaf parallelization, root parallelization, and tree parallelization with global mutex or with local mutexes. The x-axis represents the logarithmic number of threads used. The y-axis represents the winning percentage against GNU Go. For comparison reasons, we have plotted the performance of the default (sequential) program when given more time instead of more processing power. We see that root parallelization is superior to all other parallelization methods, performing even better than the sequential program.

4.6 Root Parallelization vs. Tree Parallelization Revisited

In the previous subsection we saw that on the 13×13 board root parallelization outperformed all other parallelization methods, including tree parallelization. It

Table 6. 9×9 results for root and tree parallelization using 4 threads

Time (s)	Winning percentage	
	Root parallelization	Tree parallelization
0.25	60.2 %	63.9 %
2.50	78.7 %	79.3 %
10.0	87.2 %	89.2 %

appears that the strength of root parallelization lies not only in an more effective way of parallelizing MCTS, but also in preventing that MCTS stays too long in local optima. The results could be different for other board sizes, time settings, and parameter settings. Therefore, we switched to a different board size (9×9) and three different time settings (0.25, 2.5, and 10 seconds per move). Using 4 processor threads, root, and tree parallelization played both 250 games against the same version of GNU GO for each time setting. The results are given in Table 6. For 4 threads, we see that root parallelization and tree parallelization perform equally well now. Nevertheless, the number of games played and the number of threads used is not sufficient to give a definite answer which method is better.

5 Conclusions and Future Research

In this article we discussed the use of leaf parallelization and root parallelization for parallelizing MCTS. We introduced a new parallelization method, called tree parallelization. This method uses one shared tree from which games simultaneously are played. Experiments were performed to assess the performance of the parallelization methods in the Go program MANGO on the 13×13 board. In order to evaluate the experiments, we propose the strength-speedup measure, which corresponds to the time needed to achieve the same strength. Experimental results indicated that leaf parallelization was the weakest parallelization method. The method led to a strength-speedup of 2.4 for 16 processor threads. The simple root parallelization turned out to be the best way for parallelizing MCTS. The method led to a strength-speedup of 14.9 for 16 processor threads. We saw that tree parallelization requires two techniques to be effective. First, using local mutexes instead of global mutex doubles the number of games played per second. Second, virtual loss increases both the speed and the strength of the program significantly. By using these two techniques, we obtained a strength-speedup of 8.5 for 16 processor threads.

Despite the fact that tree parallelization is still behind root parallelization, it is too early to conclude that root parallelization is the best way of parallelization. It transpires that the strength of root parallelization lies not only in an more effective way of parallelizing MCTS, but also in preventing that MCTS stays too long in local optima. Root parallelization repairs (partially) a problem in the UCT formula used by the selection mechanism, namely handling the issue of balancing exploitation and exploration. For now, we may conclude that root parallelization lead to excellent results for a specific time setting and specific program parameters. However, as soon as the selection mechanism is able to handle more adequately the balance of exploitation and exploration, we believe

that tree parallelization could become the choice for parallelizing MCTS. Preliminary experiments on the smaller 9×9 board suggest that tree parallelization is at least as strong as root parallelization.

In this paper we limited the tree parallelization to one SMP-node. In future research, we will focus on tree parallelization and determine under which circumstances tree parallelization outperforms root parallelization. We believe that the selection strategy, the time setting, and the board size are important factors. Finally, we will test tree parallelization for a cluster with several SMP-nodes.

Acknowledgments. The authors thank Bruno Bouzy for providing valuable comments on an early draft of this paper. This work is financed by the Dutch Organization for Scientific Research (NWO) for the project Go for Go, grant number 612.066.409. The experiments were run on the supercomputer Huygens provided by the Nationale Computer Faciliteiten (NCF).

References

1. KGS Go Server Tournaments. http://www.weddslist.com/kgs/past/index.html.
2. Computer Go Server. http://cgos.boardspace.net, 2008.
3. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M.P.D. Schadd, editors, *Proceedings of the Computer Games Workshop 2007 (CGW 2007)*, pages 93–101. Universiteit Maastricht, Maastricht, The Netherlands, 2007.
4. G.M.J-B. Chaslot, J-T. Saito, B. Bouzy, J.W.H.M. Uiterwijk, and H.J. van den Herik. Monte-Carlo Strategies for Computer Go. In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–90, 2006.
5. G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.
6. P.-A. Coquelin and R. Munos. Bandit algorithms for tree search. In *To appear in the proceedings of Uncertainty in Artificial Intelligence*, Vancouver, Canada, 2007.
7. R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630 of *Lecture Notes in Computer Science (LNCS)*, pages 72–83. Springer-Verlag, Heidelberg, Germany, 2007.
8. S. Gelly and Y. Wang. Exploration Exploitation in Go: UCT for Monte-Carlo Go. In *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*, 2006.
9. S. Gelly and Y. Wang. Mogo wins 19×19 go tournament. *ICGA Journal*, 30(2):111–112, 2007.
10. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.
11. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
12. L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Artificial Intelligence*, pages 282–293, 2006.