More Trees or Larger Trees: Parallelizing Monte Carlo Tree Search

Erik Steinmetz and Maria Gini, *Fellow, IEEE*

***Abstract*—Monte Carlo Tree Search (MCTS) is being effectively used in many domains, but acquiring good results from building larger trees takes time that can in many cases be impractical. In this paper we show that parallelizing the tree building process using multiple independent trees (root parallelization) can improve results when limited time is available, and compare these results to other parallelization techniques and to results obtained from running for an extended time. We obtained our results using MCTS in the domain of computer Go which has the most mature implementations. Compared to previous studies, our results are more precise and statistically significant.**

***Index Terms*—game of Go, Monte Carlo Tree search, parallelization.**

I. Introduction

Monte Carlo Tree Search (MCTS) is a method of building a search tree by successively adding single nodes to a tree, each representing some state, and randomly choosing from available actions beginning at the newly added node in a "playout," until a terminal state is reached. Each playout from a node is used to help score the goodness of that leaf node and all its predecessor nodes in the tree. MCTS is used when traditional tree search methods are insufficient, such as with extremely large state spaces and where no reasonable static evaluation function is available. It has been applied to games, scheduling, constraints, physics simulations, security testing, and other domains [1].

MCTS was a breakthrough in the Go domain after it was introduced over a decade ago, and in 2016 achieved a milestone by helping to beat the best human professional Go players, first the European Go champion Fan Hui [2] and then the world champion Lee Sedol in the "Google DeepMind Challenge Match". The program which accomplished this, called AlphaGo, used a combination of distributed MCTS along with deep convolution neural networks to achieve victory. A successor to AlphaGo called AlphaGo Zero [3] performed even better training solely on self-play, but no longer uses Monte Carlo playouts and involves large amounts of training, sometimes on specialized hardware to reduce the training time. This latter method is therefore applicable only in situations where sufficient training and testing times are available.

Recent applications of MCTS in real-time scenarios such as active sensing in robots [4], cooperative trajectory planning for automated vehicles [5], and control of radiation therapy [6] are cases where such an investment of time and effort is not tenable. Using MCTS thus remains a robust and easy-to-implement solution in many domains.

E. Steinmetz is with the Department of Computer Science, Augsburg University, Minneapolis, MN 55454 USA Email: steinmee@augsburg.edu

Maria Gini is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA Email: gini@umn.edu)

Manuscript received ....

With computer processors becoming not much faster, but more numerous along with Moore's Law, we believe that the way forward with MCTS is through efficient parallelization. Time is a critical factor in many of the domains listed above, so the only hardware improvements available will be to add more processors and more memory. In this paper we try to outline the most efficient way of using CPU power.

Our experiments were run with the mature implementations of MCTS which are available in the Go game domain. Although there are similar comparisons available in the literature, they show much variability in their results due to the use of small sample sizes which lead to large confidence intervals.

II. Background and Related Work

A. Description of the Game of Go

Go is a two-player, perfect information game played by placing black or white stones on a 19 by 19 grid of lines, with the black and white players alternating turns. The objective of the game is to surround or control the territory on the board. Once placed, the stones are not moved around on the board as are chess pieces, but remain in place until the end of the game, or until they are removed by capture. A player may elect not to place a stone by passing on their turn. The game ends when both players have passed in succession and the winner is the player who controls the most territory. There are two scoring systems, Chinese and Japanese, but they both produce equivalent results, with a possible one stone variance [7].

B. Monte Carlo Tree Search

Monte Carlo methods are used to evaluate possible actions from a given state. They are based on executing a large number of playouts through the state space, choosing random actions until a terminal state is reached, and its value is determined. In an unbiased Monte Carlo evaluation, the actions chosen in these playouts are selected entirely at random from the possible actions available in the current state. The candidate actions from the original root state are then scored by some metric based on the value of the terminal states reached by playouts through that candidate. In win/loss domains, each candidate will have a simple count. Other methods of scoring a candidate move involve the sum of the scores of the terminal states of each playout through that candidate. For example, the scale of the win or loss can be used [8].

In its simplest form, also known as flat Monte Carlo, all possible actions from the current state are evaluated with an equal number of playouts [9]. Flat Monte Carlo evaluation

does produce results, but is handicapped by a number of shortcomings. Because so many playouts are spent on sub-optimal actions, it does not scale well. Additionally the lack of an opponent model in adversarial situations makes it more likely to choose an incorrect action as the number of playouts increases [10].

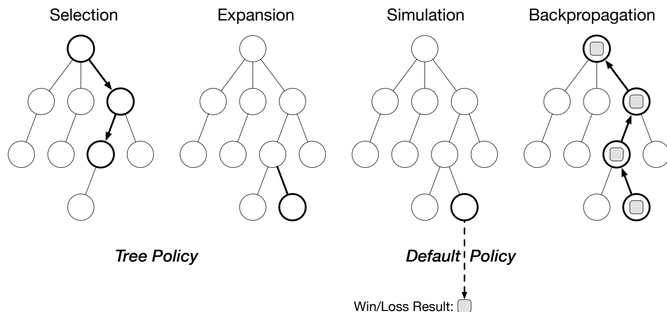

Fig. 1. MCTS Algorithm Phases

In Monte Carlo Tree Search (MCTS), the actions of a playout are given nodes containing the newly reached state and added to a search tree [11]. As this tree gets created the algorithm follows these basic steps until a resource limit (such as time or number of iterations) has been reached: selection, expansion, simulation, and backpropagation.

The *selection* phase starts at root, choosing the action or move represented by a node in the tree according to some selection policy. It then recursively descends the tree always choosing a child node according to the selection policy. When it reaches a node which has unvisited children and which represents a non-terminal state, the selection phase stops. The *expansion* phase creates a new node based as a child node of the selected node and adds it to the tree. Then the third phase begins: *simulation* plays legal actions (moves) randomly until an end position is reached. The win or loss from this simulation, also called a playout, is determined. This is added to the statistics of the newly created node and all of its parents up to the root node during the *backpropagation* phase. These steps are shown in Figure 1.

A major improvement to this basic MCTS algorithm has an extra variable to assist choosing the in-tree moves [12] during the selection phase. This method, called Upper Confidence Bounds Applied to Trees (UCT), chooses moves by iterating through the scoring of candidate moves as with the normal Monte Carlo algorithm but also uses the number of times successor nodes in the tree are encountered along with their win rate as a modification to the basic in-tree selection policy. The selection of an action a depends on maximizing $\overline{X}_a + C\sqrt{\frac{2\ln n}{n_a}}$ where $\overline{X}_a$ is the average reward for having selected action a, n_a is the number of times a has been selected, and n is the total number of times the current node has been selected. The first term represents how successful choosing the action has been so far and is known as the *exploitation factor*. This term grows or shrinks depending on the scores of playouts through the choice of action a, while the second term represents the confidence in the score, and is known as the *exploration factor*. As the number of times this action has been chosen from its parent rises in comparison to all the other children of that parent, this term will decrease. If an action has only been tried a few times, its scored value will have a low confidence, and so in order to increase the confidence of the winning rate for that action, the odds of selecting these actions are increased by the exploration factor so that they will be chosen more often. If an action has not yet been chosen at all, this second term will go to infinity ensuring this action will be chosen before other actions from the same parent state that have already been selected once.

In domains where the ordering of actions is not always critical to their value, there is a modification called RAVE, for rapid action value estimation [13]. This formula scores actions not just based on choosing them at the current state (of the node being scored), but also considers choosing that action at any point during the sequence of actions. This is known as the all-moves-as-first (AMAF) heuristic [14].

Other modifications include integrating domain dependent knowledge [15] and heuristics [16] along with an agent-like system for polling different agents (or algorithms) for action choices [17]. Biasing in-tree node selection based on learned patterns when they are available [18] or from expert play [19] has improved play in the domain of Go. In the Go domain, a system of modifying action choices in the simulation phase using a lookup table based on individual action sequences from successful playouts [20] worked well and was improved by removing from the table those sequences which were subsequently chosen in an unsuccessful playout [21].

C. Parallelization of MCTS

There are generally three ways to parallelize search in an MCTS tree [22] [23]: (1) at the leaves of the search tree, (2) throughout the entire tree, and (3) at its root.

In *leaf parallelization*, after a node is selected in the tree for expansion, multiple playouts originating at that node are conducted in parallel. When all the playouts have finished, the combined score from those playouts is propagated back up the tree. Depending on the number of parallel threads this can greatly increase the speed at which the win-rate statistics are gathered, however since each playout may take a different length of time, some efficiency is lost waiting for the longest playout to complete. Some implementations have moved these playouts to a GPU [24]. Unfortunately many of the playouts in leaf level parallelization are wasted due to their simultaneous nature: if the first eight playouts all are losses or very low-scoring, it is unlikely that the next eight will do any better, leading to an inherent limitation of this technique.

In *tree parallelization*, multiple threads perform all four phases of MCTS (descend through the search tree, add nodes, conduct playouts, and propagate statistics) at the same time. In order to prevent data corruption from simultaneous memory access, mutexes (locks) are placed on nodes that are currently in use by a thread. These locks lead to scalability problems since many threads may want to be exploiting a particular node in a tree. One solution to this mutex problem has been to introduce virtual losses [22]. When a thread descends through a node in the tree, it adds a virtual loss to the statistics of

that node, making it less appealing to other threads. When the thread updates statistics, it removes the virtual loss. Another implementation [25] has implemented a lock-free version of tree parallelization utilizing the virtual loss system. Without locks this depends on a large enough virtual loss to deter other threads to the point that overwriting of old data will occur very rarely. Using the synchronization properties of the atomic variables, atomic operations, and atomic variants of the builtin types in C++ [26] the ordering of memory accesses by the threads can be enforced [27], creating a lock-free system. The results show better playout speedup compared to other synchronization methods, but still no better performance than root parallelization, unless the number of tasks and the exploration factor are high. Similarly, separately tracking statistics about the number of threads currently exploring through a node [28] has successfully avoided mutex delays. Tree parallelization can be created on shared memory systems or on distributed systems with very fast interconnects (clusters). Clusters usually suffer from the communications overhead, but one implementation using distributed depth-first UCT [29] avoids some of this by not propagating results to root at every playout.

In *root parallelization*, multiple independent trees are built by separate threads with no information communicated between them while the trees are being built [23]. This can happen either on a shared memory machine or in a cluster. All the trees are created through the end of the time limit on the machine or machines, and then the scores for the top layer nodes, that is to say the nodes which represent the immediate choice from the root node, are combined to determine which action will be chosen. A variant of this, dubbed "slow root parallelization" shares this top-level information periodically throughout the search [23], [30]. Although the trees have been created starting from the same node, the stochastic element of MCTS means that each tree will be formed differently from the others. When the information from the different trees is combined, two methods of combining the values from the trees are commonly used. The first is to add up all the scores for each possible action from all the trees. In this case the combined score for an action is the sum of each tree's score for that action. The second method is to choose the action which "won" the contest in each of the trees. These two methods are called "average" voting and "majority" voting respectively.

D. Comparisons of Parallelization of MCTS

Chaslot, Winands, and van den Herik [22] compared all three of these parallelization methods. They created a new measure of scalability which was based on the increase in winning rates as the amount of time per move increased, naming it the "strength speedup" measure. They concluded that the best increase in strength was achieved by using root parallelization, with average voting. They did not test the majority voting method. This was tested on their (relatively weak) Mango program. These experiments used very short time limits per move starting at 1 second as the baseline.

Soejima, Kishimoto, and Watanabe [31] explored root parallelization, comparing a majority voting system versus an average voting system, and found that majority voting was superior. They compared results over 9×9, 19×19 with both self play using Fuego and against MoGo. They also tested move selection on particular board problems against an "oracle" version of the software which was Fuego running for 80 seconds, which is 8 times their baseline move time of 10 seconds in other experiments, but for the move selection experiments they ranged from 1 to 64 seconds.

Schaefers and Platzner [32] analyze the effects of using a parallel transposition table for tree parallelization along with dedicating some compute nodes to broadcast operations to help scaling to large numbers of machines.

Świechowski and Mańdziuk [33] measured the performance of their "Limited Hybrid Root-Tree Parallelization" on General Game Playing. The system combines tree parallelization with root parallelization by building each of the separate trees in a root parallel system using a tree-parallelized player. A similar hybrid approach was also utilized as part of Steinmetz's Ph.D. thesis [34] and as part of the current work.

III. Methodology and Experiments

We compared the benefits of root parallelization to tree parallelization and measured both against a baseline of building a larger tree utilizing more time. We ran our experiments in playing the game of Go because the MCTS software for computer Go has become quite mature. Additionally the game offers a look at problems of different inherent difficulty by varying the size of the game board. The game of Go was the first to seriously develop MCTS, and offers a selection of open-source programs with which to experiment, including Pachi [35] and Fuego [36], two of the strongest computer Go programs. In addition, in order to look at parallelization in a domain without using the all-moves-as-first effects, we include data from running a computer Go program with the no RAVE implementation of MCTS.

We ran and looked at 3 different MCTS setups:

1) More time: as our baseline measurement, we increased the time available to a single thread thereby allowing a larger tree to be created.
2) More threads: we increased the number of threads operating on a single tree, so a larger tree can be created.
3) More compute nodes: we increased the number of compute nodes on which to run MCTS, and so the number of trees that would contribute to the choice of action.

There are two highly ranked open source MCTS Go engines, Pachi and Fuego. We used Pachi 11 and Fuego revision 1983 to conduct our experiments.

Our Go experiments consisted of running tournaments between these two programs and recording the win/loss rates using Chinese scoring rules. We used Fuego and our modifications of Fuego as the variable program, modifying it and its parameters. These various incarnations of Fuego were then played against Pachi running always with the same set of parameters. All of our results are stated in terms of the percentage of games won by Fuego against Pachi. Pachi was played with all default parameters except that it was run as a single thread with a time limit of 15 seconds per turn with no

opening book. Pachi plays with a RAVE component. Fuego was run with no opening book, at the time limit specified in the experiment, and on the number of threads specified in the experiment. Fuego also runs with a RAVE component.

In order to observe the effects of parallelization in a situation where the all-moves-as-first (AMAF) heuristic was not applicable, we also ran experiments with a version of Fuego which did not use the RAVE statistics.

In the game of Go two different board sizes are used to play games, especially between computers. A game on a 9×9 board usually lasts on average about 60 moves before it is decided or one side resigns. A game on a 19×19 board lasts about 260 moves, and so involves a much larger state space and takes from four to five times as long to play. These two board sizes were used to observe effects of parallelization on the algorithm at two different levels of problem difficulty.

The tournaments were run on a cluster of HP blade servers, where each compute node contained 2 quad-core 2.8 GHz Xeon "Nehalem EP" processors sharing 24 GB of memory.

Tournaments between the computer opponents were moderated by the "gogui-twogtp" script [37] which communicates with each program using the simple Go Text Protocol and records the results of the games played.

In order to study the trend of performance in the winning rates in an adversarial game situation we use the binomial confidence interval $p \pm 1.96\sqrt{p(1-p)/n}$ where p is the probability of a win and n is the number of trials in the sample. This means that 95% of the time an experiment is run the actual value sought will be within the confidence interval of the value seen in the sample by the experiment. Given this formula a reasonably large number of games must be played to detect an improvement in the software's ability. If the winning rate is near 50% for example, the confidence interval for 500 games is a bit below 5%.

Where many of the previous studies used tournament sizes in the hundreds, we chose to run tournaments of 1000 games in order to get confidence intervals close to $\pm 3\%$. When comparing different win rates, we use the two-proportions z-test with a null hypothesis that the better win rate is equal or less than the smaller win rate. The p-value from this test is thus the likelihood that the better win rate is not actually larger.

A. More Time for a Larger Tree

We increased the time available to a single thread, thereby allowing a larger tree to be created. The longer an MCTS algorithm is run, the larger the search tree will become and this usually improves the quality of the results. All trees eventually run into memory limitations, and need to have their least promising branches pruned to free up the memory to build out the higher quality parts of the tree. Nonetheless, measured by the total number of playouts conducted, the size of the tree explored at some point, even if it has been subsequently pruned, rises close to linearly with the time allotted.

We allowed an MCTS program to run for longer than a typical amount of time in our domain, doubling and redoubling the time allowed, in order to see the effect this would have on the quality of the resultant decisions.

We chose a standard time allotment for our domain as the baseline amount of time. In the case of computer Go, using 15 second turns for decisions is quite common. We picked this as 1 unit of time, and then kept doubling the time allowed based on that unit. This avoids the problems of showing large improvements from an unrealistically small baseline time unit. We measured the quality by comparing the winning rate of a computer Go program playing against an opponent that was only allowed to use the standard amount of time.

For the *more time* experiment in Go, we played a tournament on a 9×9 board with Fuego running a single thread for 15 seconds available per move against Pachi running a single thread for 15 seconds per move. We then repeated this tournament five more times, allowing Fuego 30 seconds per move, then 60, 120, 240, and finally 480 seconds per move. We then repeated this experiment with the no RAVE version of Fuego along with a tournament on a 19×19 board.

The results from the tournaments between copies of Fuego with increasing amounts of time allotted for each decision and Pachi with the fixed 15 seconds of time per turn are shown in Table I. As the amount of time allowed increases, the winning rate of Fuego increases until it is playing with eight times the base time, i.e., 120 s, after which no significant advantage is seen. This is also seen on the 19×19 board. The results from the no RAVE Fuego with increasing time limits show essentially no gain from increasing the time available when it is using the default urgency value.

TABLE I
FUEGO WIN RATES WITH INCREASING TIME PER MOVE LIMITS FOR A 9×9 GO BOARD

Time	15 s	30 s	60 s	120 s	240 s	480 s
Fuego vs Pachi 9×9	53.2%	57.5%	63.8%	70.3%	68.6%	70.7%
Fuego (no RAVE) vs Pachi	8.3%	9.5%	10.3%	8.7%	8.7%	7.9%
Fuego vs Pachi 19×19	42.1%	53.7%	59.7%	59.6%	56.0%	60.6%

B. More Threads for a Larger Tree

Another way to increase the size of the search tree is to allow more than one thread to work on the same tree at a time: tree parallelization. The scalability of this approach is limited by the number of cores that access the same memory on a machine with a shared memory architecture. Current high-end consumer-grade processors are four-core, but act as though they have eight cores by using hyper-threading technology.

We ran an MCTS program capable of conducting lock-free tree parallelization on a single machine with an increasing number of threads and recorded the quality of the win rate results as the number of threads increased.

For the *more threads* experiment, we played four 9×9 tournaments matching up Fuego against Pachi, allowing Fuego to play with 1, 2, 4, and 8 threads respectively. Pachi was kept to a single thread. We also ran these four tournaments with the no RAVE version of Fuego against Pachi, and another four tournaments on a 19×19 board.

Tournaments between copies of Fuego set to run with varying number of threads versus Pachi on a single thread produced the results seen in Table II. As the number of threads

increases, the strength of the program appears to increase almost linearly up to the maximum of eight, the number of processors in our shared-memory systems.

TABLE II
FUEGO WIN RATES WITH INCREASING NUMBER OF THREADS

Number of Threads	1	2	4	8
Fuego vs Pachi 9×9	53.2%	59.0%	65.1%	74.2%
Fuego (no RAVE) vs Pachi 9×9	8.3%	10.5%	12.4%	18.9%
Fuego vs Pachi 19×19	42.1%	57.5%	65.9%	75.5%

C. More Compute Nodes for More Trees

We increased the number of compute nodes over which to run MCTS, and so the number of trees that would be built and contribute to the choice of action. We implemented a parallel version of the MCTS algorithm with root level parallelism. Because of the success seen in [31], we used a system of majority voting to determine the next action to be taken.

At initialization, N copies of the MCTS program are created on different machines in a cluster. Each compute node keeps track of the current board state. When asked to produce an action choice, the compute node runs an MCTS search without sharing information with the other nodes. When the decision has been reached, each node reports its decision to the head node, node 0, but does not implement that decision: it does not change the state. For each possible action, compute node 0 tallies up the number of nodes on which that action was chosen. The action with the largest number of votes, one vote per node, is considered the winner. This decision is then reported back to each node, which implements that winning action and changes the state accordingly. At this point, each compute node should have a copy of the current state and be ready for the next request.

We ran this majority vote root-parallel version of MCTS using an increasing number of compute nodes and recorded the changes in the winning rates.

In the experiment with *more compute nodes* to create multiple trees we played six tournaments on a 9×9 board. In each tournament both Fuego and Pachi were run on single threads with 15 second time limits. The number of compute nodes Fuego was allowed to use, and hence the number of trees it built, was varied with 1, 8, 16, 32, 64 and 128 nodes respectively in the tournaments. We then repeated these tournaments using the no RAVE version of Fuego against Pachi on a 9×9 board and using Fuego against Pachi with the games played on a 19×19 board.

Tournaments played between the root parallelized versions of Fuego with an increasing number of compute nodes against the single-threaded Pachi are shown in Table III. In each case, the major jump in strength is from one to eight nodes.

IV. COMPARING PARALLELIZATION TECHNIQUES

We compare the effects of these techniques in our Fuego vs Pachi tournaments in Fig. 2. As the time or number of threads or number of trees increases, shown in log scale on the x-axis, all these techniques increase their strength from a

TABLE III
FUEGO WIN RATES WITH INCREASING NUMBER OF COMPUTE NODES FOR A 9×9 GO BOARD

Number of Nodes	1	8	16	32	64	128
Fuego vs Pachi 9×9	53.2%	62.8%	66.1%	69.0%	68.5%	69.7%
Fuego (no RAVE) vs Pachi	8.3%	22.8%	25.8%	28.4%	22.8%	24.1%
Fuego vs Pachi 19×19	42.1%	62.3%	65.8%	64.5%	67.9%	66.8%

baseline value. Increasing the number of threads using a lock-free implementation works the best (at 8× level compared to time the p-value is 0.02887), but is limited by the number of CPUs in a shared memory architecture. Increasing the number of compute nodes scales almost as well as increasing the time, but both of these see diminishing returns between the 8 and 32 multiplier levels.

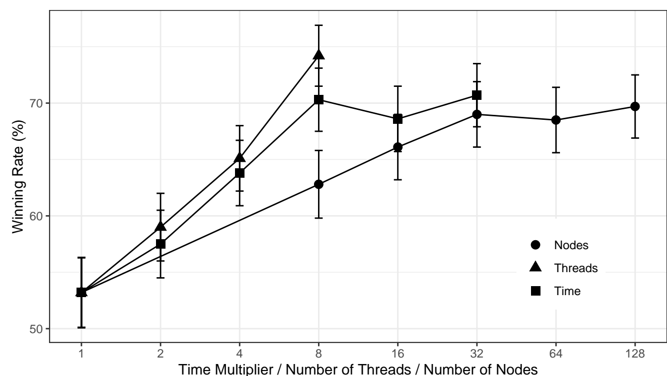

Fig. 2. Comparison of Fuego winning rates vs Pachi for increasing numbers of compute nodes, numbers of threads, and increased time MCTS for a 9×9 Go board. Notice that the x-axis is in log scale.

We compare these effects again in our Fuego (no RAVE) vs Pachi tournaments shown in Fig. 3. In this case the best result is achieved by increasing the number of compute nodes (at 8× level compared to threads the p-value is 0.01823), but appears to maximize at 32 nodes.

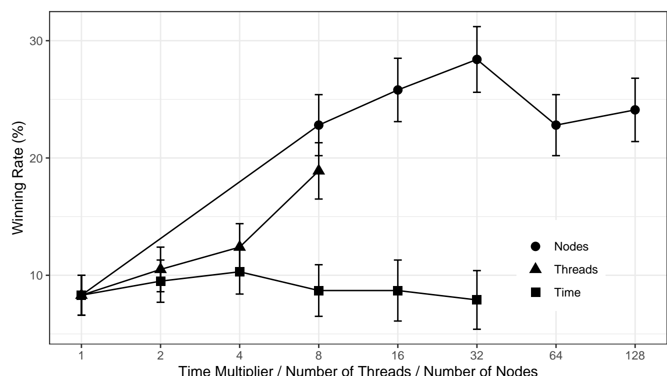

Fig. 3. Comparison of Fuego (no RAVE) winning rates vs Pachi for increasing numbers of compute nodes, numbers of threads, and increased time MCTS for a 9×9 Go board. Notice that the x-axis is in log scale.

Finally we compare these three techniques when used on the larger and more difficult 19×19 board (see Fig. 4). They

appear similar to the 9×9 results except the *more time* results stop improving after the 4 multiple rather than the 8 multiple.

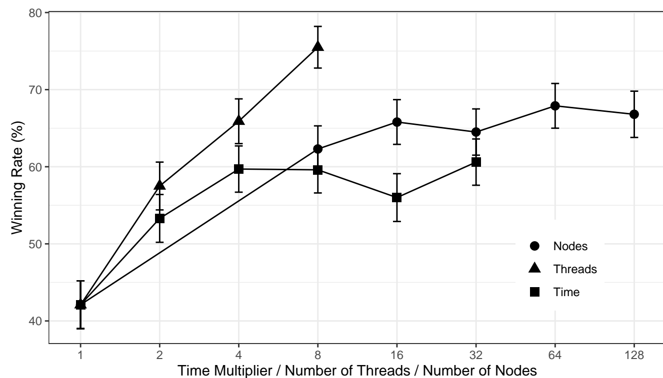

Fig. 4. Comparison of Fuego winning rates vs Pachi for increasing numbers of compute nodes, numbers of threads, and increased time MCTS for a 19×19 Go board. Notice that the x-axis is in log scale.

V. Discussion and Future Work

We surveyed the effects of building larger search trees with more time or more threads along with building more search trees using root level parallelism. Each of these increases performance, but have different limitations. Using *more time* appears to be limited by memory available to the single-threaded process. Using *more threads* appeared to be most effective but cannot be scaled past the number of cores that can share memory. Using *more trees* requires a minimum of eight nodes in order to have enough votes to pick a winning move: using fewer trees often results in each answer receiving one vote. This leaves no way to pick the best action.

The increasing win rates with time appear to run into a performance ceiling at earlier multiples than previous research [22] where it was used as a baseline to measure parallelization improvements. Here increasing time did not provide any meaningful improvement after the 4 or 8 factor increase. Given our starting unit of 15-second play, realistic in the domain, we hypothesize that memory constraints restricted the ability to find better moves. As search trees get large enough to consume all allocated memory, some search nodes end up being pruned from the tree. The rules for pruning depend on the implementation, but involve the least likely to succeed branches of the tree. The odd results in our Fuego no RAVE experiment appear to bear this out. Without RAVE the scores in the search nodes are so unreliable (they have such a high variability) that when pruning takes place the algorithm is unable to adequately distinguish which branches to remove to free up space.

One interesting result using root parallelization was that although in 9×9 Go performance gains appeared to level off at 32 compute nodes, in 19×19 Go the gains appeared to continue through 64 compute nodes before leveling off. We believe this requirement for more compute nodes may be due to the larger number of good choices available in the larger game. As the number of voting compute nodes increases, the likelihood of picking the best move increases, but this may plateau at some multiple of the available good moves. For example, if there are only three reasonable moves in a 9×9 game having 64 instead of 32 voters choosing may not increase the quality of the pick, but if there are ten reasonable moves available it may require 64 or more nodes to find one of the best as many of the votes will be spread out over the wider selection of reasonable moves. Observing the number of disparate moves chosen by the nodes at various parts of the game and with different numbers of nodes available could help answer this question, and we will include this in our future research.

VI. Conclusions and Future Work

Compared to previous results which have found that parallel algorithms under-performed against simply running the algorithm for a longer period of time, we have shown that parallel algorithms keep pace with or may exceed the performance gained by increasing the amount of time in the domain of Go.

We believe this is due to both the maturity of the algorithms now available, and the use of a reasonable baseline time unit for measurement. Weaker programs have more variability in their playing strength, and it is deceptively easy to get a jump in performance when starting from an unrealistically low time scale. If a quality-improving measure such as RAVE is available, multiple threads outperform multiple compute nodes, but the opposite is true when such measures are not applicable. Future research will explore if this is true in domains other than Go

Additionally as more cores become inexpensive on shared-memory machines, it will be possible to determine the number of lock-free threads at which multiple threading stops providing gains. Other future work will include looking at the specific impact of tree pruning on performance and at exploring possibilities for heterogeneous algorithms in root parallelism.

Acknowledgements

We gratefully acknowledge the Minnesota Supercomputing Institute (MSI) for the use of their computing facilities, and Daniel Boley for his support through the project.

References

[1] C. Browne *et al.*, “A survey of Monte Carlo Tree Search methods,” *IEEE Trans Comput Intell AI Games*, vol. 4, no. 1, pp. 1–43, 2012.

[2] D. Silver, A. Huang *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 01 2016.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] G. Best, O. M. Cliff, T. Patten, R. R. Mettu, and R. Fitch, “Decentralised Monte Carlo Tree Search for active perception,” in *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, K. Goldberg, P. Abbeel, K. Bekris, and L. Miller, Eds. Springer International Publishing, 2020, pp. 864–879.

[5] K. Kurzer, C. Hörtnagl, and M. Zöllner, “Parallelization of Monte Carlo Tree Search in continuous domains,” arXiv:2003.13741v1 [cs.LG], 2020.

[6] P. Dong, H. Liu, and L. Xing, “Monte Carlo Tree Search-based non-coplanar trajectory design for station parameter optimized radiation therapy (SPORT),” *Physics in Medicine & Biology*, vol. 63, no. 13, p. 135014, 2018.

[7] S. Masayoshi, *A Journey in Search of the Origins of Go*. Yutopian Enterprises, 2005.

[8] H. Yoshimoto, K. Yoshizoe, T. Kaneko, A. Kishimoto, and K. Taura, "Monte Carlo Go has a way to go," in *21st Nat'l Conf. on Artificial Intelligence (AAAI-06)*, 2006, pp. 1070–1075.

[9] B. Brügmann, "Monte Carlo Go," October 1993, http://www.ideanest.com/vegos/MonteCarloGo.pdf.

[10] C. Browne, "The dangers of random playouts," *ICGA Journal*, vol. 34, no. 1, pp. 25–26, 2011.

[11] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, ser. Lecture Notes in Computer Science, vol. 4630. Springer, 2006, pp. 72–83.

[12] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006, 17th European Conf. on Machine Learning*, ser. LNCS, vol. 4212. Springer, 2006, pp. 282–293.

[13] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[14] B. Bouzy and B. Helmstetter, "Monte-Carlo Go Developments," in *Advances in Computer Games conference (ACG-10)*, E. A. Heinz, H. J. van den Herik, and H. Iida, Eds. Kluwer Academic Publishers, 2003.

[15] B. Bouzy, "Associating domain-dependent knowledge and Monte Carlo approaches within a Go program," *Information Sciences, Heuristic Search and Computer Playing IV*, vol. 175, no. 4, pp. 247–257, 2005.

[16] P. Drake and S. Uurtamo, "Heuristics in Monte Carlo Go," in *Proceedings 2007 Int'l Conf. on Artificial Intelligence (IJCAI)*, 2007.

[17] L. S. Marcolino and H. Matsubara, "Multi-agent Monte Carlo Go," in *Proc. Int'l Conf, on Autonomous Agents and Multiagent Systems*, 2011, pp. 21–28.

[18] M. Michalowski, M. Boddy, and M. Neilsen, "Bayesian learning of generalized board positions for improved move prediction in computer Go," in *Proc. 25th Conf. on Artificial Intelligence (AAAI)*, 2011.

[19] E. S. Steinmetz and M. L. Gini, "Mining expert play to guide Monte Carlo search in the opening moves of Go," in *Proc. of the Twenty-Fourth Int'l Joint Conf. on Artificial Intelligence, IJCAI*, 2015, pp. 801–807.

[20] P. Drake, "The last-good-reply policy for Monte-Carlo Go," *Int'l Computer Games Association Journal*, vol. 32, no. 4, pp. 221–227, 2009.

[21] H. Baier and P. D. Drake, "The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go," *IEEE Trans Comput Intell AI Games*, vol. 2, no. 4, pp. 303–309, Dec 2010.

[22] G. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," in *Proc. Computers and Games, 6th Int'l Conf, (CG)*, 2008, pp. 60–71.

[23] T. Cazenave and N. Jouandeau, "On the parallelization of UCT," *Proceedings of CGW07*, pp. 93–101, 2007.

[24] K. Rocki and R. Suda, "Large-scale parallel Monte Carlo Tree Search on GPU," in *IEEE Int'l Symp on Parallel and Distributed Processing, (IPDPS)*, 2011, pp. 2034–2037.

[25] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo tree search algorithm," in *Advances in Computer Games, 12th International Conference (ACG)*, 2009, pp. 14–20.

[26] A. Williams, *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series, 2012.

[27] S. A. Mirsoleimani, J. van den Herik, A. Plaat, and J. Vermaseren, "Lock-free algorithm for parallel MCTS," in *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART)*, vol. 2, 2018, pp. 589–598.

[28] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu, "Watch the Unobserved: A simple approach to parallelizing Monte Carlo Tree Search," in *International Conference on Learning Representations*, 2020.

[29] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa, "Scalable distributed Monte-Carlo tree search," in *Proc. Fourth Annual Symposium on Combinatorial Search (SOCS)*, 2011.

[30] A. Bourki *et al.*, "Scalability and parallelization of Monte-Carlo tree search," in *Computers and Games*. Springer, 2010, pp. 48–58.

[31] Y. Soejima, A. Kishimoto, and O. Watanabe, "Evaluating root parallelization in Go," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 2, no. 4, pp. 278–287, 2010.

[32] L. Schaefers and M. Platzner, "Distributed Monte Carlo tree search: A novel technique and its application to computer Go," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 7, no. 4, pp. 361–374, 2015.

[33] M. Świechowski and J. Mańdziuk, "A hybrid approach to parallelization of Monte Carlo Tree Search in general game playing," in *Challenging Problems and Solutions in Intelligent Systems*. Springer International Publishing, 2016, pp. 199–215.

[34] E. S. Steinmetz, "Computer go and Monte Carlo tree search: Opening book and parallel solutions," Ph.D. dissertation, University of Minnesota, May 2016.

[35] P. Baudiš and J. Gailly, "Pachi: State of the art open source Go program," in *Advances in Computer Games*, ser. Lecture Notes in Computer Science, vol. 7168. Springer, 2011, pp. 24–38.

[36] M. Enzenberger, M. Müller, B. Arneson, and R. Segal, "Fuego - an open-source framework for board games and Go engine based on Monte Carlo tree search," *IEEE Trans Comput Intell AI Games*, vol. 2, no. 4, pp. 259–270, Dec 2010.

[37] M. Enzenberger, "Gogui," http://gogui.sourceforge.net, 2012.

Erik Steinmetz received the M.S. degree in computer science (1999) and the Ph.D. degree in computer science (2016) from the University of Minnesota. His dissertation was devoted to the game of Go, of which he is a passionate player. In 2013 he was a National Science Foundation / Japanese Society for the Promotion of Science summer research fellow.

He is currently an Assistant Professor at Augsburg University in Minneapolis, Minnesota, where he enjoys teaching a variety of courses and advising undergraduate projects. From 2000 to 2006 he helped run Iterativity, Inc., a small research company, and from 1997 to 2000 was a research scientist at Honeywell International.

Maria Gini is a College of Science & Engineering Distinguished Professor in the Department of Computer Science and Engineering at the University of Minnesota. She works on distributed decision making for autonomous agents in many application domains, ranging from swarm robotics to task allocation, exploration of unknown environments, navigation in dense crowds, and conversational agents. She is a Fellow of the ACM, IEEE, and AAAI. She has published more than 60 journal articles and more than 300 conference papers and book chapters. She is Editor in Chief of Robotics and Autonomous Systems, and is on the editorial board of numerous journals, including Artificial Intelligence, Autonomous Agents and Multi-Agent Systems, Current Robotics Reports, and Integrated Computer-Aided Engineering.