Evaluating Root Parallelization in Go

Yusuke Soejima Email: soejima1024@gmail.com
Akihiro Kishimoto Email: kishimoto@is.titech.ac.jp
Osamu Watanabe Email: watanabe@is.titech.ac.jp

***Abstract*—Parallelizing Monte-Carlo tree search has been considered to be a way to improve the strength of computer Go programs. In this paper, we analyze the performance of two root parallelization methods – the standard strategy based on average selection and our new strategy based on majority voting. As a starting code base, we used Fuego, which is one of the best programs available. Our experimental results with 64 CPU cores show that majority voting outperforms average selection. Additionally, we show through an extensive analysis that root parallelization has limitations.**

***Index Terms*—Monte-Carlo tree search, computer Go, root parallelization, majority voting, tree parallelization**

I. Introduction

A. *Motivations*

Computer Go has been an extremely difficult challenge for game researchers. Since the invention of the UCT algorithm [1], there have been many computer Go programs based on variations of Monte-Carlo Tree Search (MCTS), including UCT. Examples of strong Go programs leveraging MCTS are MoGo [2], Crazy Stone [3], and Fuego [4], [5]. The best programs are currently reaching the level of strong human professional players in 9×9 Go, and have won against a professional player with a 7 stone handicap in 19×19 Go.

MCTS consists of Monte Carlo sampling called *playout* and tree search. In a playout at position P, MCTS keeps selecting a move for each player almost randomly except for one filling in an eye point of that player until it reaches a terminal position Q. The result of Q (i.e. either a win, loss, or draw) is used to compute the winning ratio at P as an evaluation criterion.

The tree search engine of MCTS selects the most promising path defined by the upper confidence bound (UCB) values [1] to find a leaf node. MCTS then expands the leaf and performs a playout. Next, it updates the UCB values of all affected branches along the path back to the root. It continues these procedures until the time has come to select the next move to play.

Performance improvements for MCTS are achieved by building a larger search tree and increasing the number of playouts. A larger tree enables MCTS to concentrate on a more promising part of the search space, while more playouts result in more accurate evaluations of positions.

Parallel computing has been widely used to improve the playing strength of MCTS computer programs. Two of the main approaches that have been extensively studied are tree parallelization and root parallelization. Tree parallelization shares a search tree among processors [2] and is a representative approach incorporated into strong computer Go programs (e.g., [6], [7]).

While tree parallelization can build a larger search tree, it suffers from an extra overhead incurred by tree sharing. On the other hand, in root parallelization each process performs local MCTS without sharing the search tree [8]. Clearly, root parallelization can linearly increase the number of playouts per second, as the number of CPU cores increases. Additionally, each processor in root parallelization often generates a search tree that has different characteristics and that is constructed by the different outcomes of playouts performed individually.

In fact, Chaslot et al. reported that root parallelization not only is more effective than tree parallelization up to 16 CPU cores but also often achieved super-linear speedups [9] in improving the strength of their program. Their hypothesis for explaining this phenomenon is that tree parallelization suffers from staying at local optima, while root parallelization escapes from such traps. However, they do not provide any evidence explaining why root parallelization performs better than tree parallelization. This paper tries to elucidate the pros and cons of root parallelization.

B. *Contributions of this Paper*

When a parallel algorithm achieves super-linear speedups, it often indicates that there is still enough room to improve its sequential counterpart. The parallel version of the program used for the experiments performed by Chaslot et al. in [9] was developed on top of Mango, which was fifth (tied with one program) out of ten programs in 9×9 Go and fourth (tied with two programs) out of eight participants in 19×19 Go at the 12th Computer Olympiad[1]. Although Mango incorporates progressive pruning strategies [10] to further enhance performance, using a stronger sequential program as a baseline for experiments is preferred so that a parallel algorithm cannot easily exploit the weakness of its sequential counterpart. Additionally, Chaslot et al. measured the performance of root parallelization up to only 16 cores. Investigating the scalability of root parallelization with a larger number of CPU cores remains an unanswered question.

This paper evaluates the effectiveness of root parallelization with a larger number of CPU cores than in [9] by using a recent version of Fuego as a baseline for experiments. Fuego is not only one of the best programs in the recent Computer Olympiad but also includes several important enhancements including the rapid action value estimation (RAVE) algorithm [11], which greatly improves the strength of sequential MCTS.

Our contributions are summarized as follows:

[1]See http://www.grappa.univ-lille3.fr/icga/tournament.php?id=169 and http://www.grappa.univ-lille3.fr/icga/tournament.php?id=167.

- Measurement of the performance of root parallelization with 64 CPU cores.
- A new algorithm based on majority voting, which experimentally outperforms the standard strategy of root parallelization.
- Experimental results showing a limitation of root parallelization compared with tree parallelization.
- Detailed analysis of the properties of moves chosen by root parallelization and tree parallelization.
- Estimation of the scalability of majority voting with 512 cores.

C. Structure of this Paper

The rest of the paper is organized as follows: Sections II and III deal with sequential and parallel MCTS. Section IV gives an overview of Fuego. Section V presents root parallelization based on majority voting. Section VI analyzes the experimental results. Section VII concludes by briefly summarizing and mentioning future work.

II. Sequential MCTS

MCTS performs best-first search combined with playouts to evaluate positions. The UCT algorithm [1], which is a representative method of MCTS and is incorporated into many computer Go programs, manages a search tree that is built incrementally and saved in memory. Let i be an edge (i.e. move) leading to a child $child_i$. UCT leverages the UCB value for i, which is computed as

$$\text{UCB}(i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln s}{n_i}}$$

where c is a constant, w_i is the number of wins computed by playouts at $child_i$ and its descendants, n_i is the number of visits to $child_i$, and s is the number of visits to $child_i$ and its siblings.

Move i must be selected if i's winning ratio $\frac{w_i}{n_i}$ is large (*exploitation*). However, if the winning ratio is small due to inaccuracy caused by small n_i and w_i, then performing search and playouts below $child_i$ is still important (*exploration*). The UCB value provides a good trade-off for such an exploitation-exploration dilemma.

Starting at the root, MCTS keeps selecting a move with the largest UCB value until it reaches a leaf node l that has not yet been expanded. MCTS then expands l and performs a playout there. A playout consists of a game in which each player almost randomly selects legal moves until reaching a terminal position where the winner of the game can be easily determined by counting the players' territories. The result (i.e., a win, draw, or loss) of the playout is propagated back to the root by updating the UCB values on the moves that are on the path to l from the root. In other words, MCTS increments n_i (and w_i according to the playout result) of such moves. MCTS continues this procedure until it is time to play a move.

When playing a move, typical computer Go programs select not a move with the largest UCB value but one leading to a child with the largest number of visits. A move with a large UCB value is sometimes unreliable due to a much smaller number of visits. In practice, it is more important to select the move with higher reliability.

Two typical enhancements are incorporated into MCTS. MCTS leverages local patterns based on Go dependent knowledge to assign non-uniform probabilities to moves when performing playouts [2], [3]. RAVE quickly estimates a UCB value for each move by reusing the playout results [11].

III. Related Work on Parallel MCTS

A. Obstacles to Parallel MCTS

When search algorithms are parallelized, the parallel search usually suffers from an extra *overhead* that never occurs in a sequential algorithm. As similarly discussed in many papers on parallel game-tree search (e.g., [12], [13], [14]), there are mainly three kinds of overhead in parallel MCTS:

- *Search overhead* is a useless part of a tree built only by parallel MCTS.
- *Synchronization overhead* is the idle time incurred when some processors must wait for others to finish their computations. An example of synchronization overhead includes a synchronization point caused by acquiring mutual exclusion (mutex) locks to access shared data in a shared-memory environment when more than one processor tries to enter such a critical section.
- *Communication overhead* is caused by the communication delay when processors exchange information over the network. Communication overhead is incurred mainly in a distributed-memory environment.

In principle, parallel algorithms achieve best performance by minimizing these overheads. However, in practice, it is difficult to find the best condition because these overheads depend on one another. Moreover, it is still an open question which factor is more important: building a larger tree, or increasing the number of playouts. For example, leaf parallelization [8] increases the number of playouts by performing playouts in parallel at leaf nodes without increasing the search tree size compared with sequential MCTS. However, this approach suffers from performance degradation caused by severe synchronization overhead, because it waits for all the processors to finish their playouts at each leaf. As a result, leaf parallelization is less effective than tree parallelization and root parallelization[2]. Although Kato et al. removed the synchronization points of leaf parallelization [15], their approach may perform playouts at less promising leaf nodes. Cazenave and Jouandeau's parallel master-slave algorithm can perform playouts at different leaf nodes and scales well up to 16 processors [16]. However, it suffers from the overhead caused by the master process and communications.

B. Tree Parallelization

Tree parallelization performs parallel MCTS while sharing a UCT tree among processes (or threads). Tree parallelization is a natural extension based on existing research on traditional

[2]The terms root parallelization, tree parallelization, and leaf parallelization were first introduced by Chaslot et al. [9]. Tree parallelization and root parallelization are explained later in this section.

parallel game-tree search [17], because tree parallelization splits a shared search tree, thus building a larger tree and increasing the playout size. In particular, in a shared-memory environment, tree parallelization has been incorporated into state-of-the-art programs including Fuego, MoGo, and Crazy Stone [6], [7].

1) Tree Parallelization in Shared-Memory Environments: In the case of tree parallelization in a shared-memory environment, each thread traverses a shared tree to find a leaf node. Playouts are performed independently without any interactions among threads. However, in a shared-memory environment, each thread must lock a node of the shared tree when accessing that node; this lock is necessary to ensure consistent results. Tree parallelization therefore incurs synchronization overhead when more than one thread accesses the same node at a time. According to [9], Coulom suggested *virtual loss*, which assumes the outcome of a playout to be a loss until that playout finishes and returns its result. This approach enables threads to search different parts of the shared tree, thus reducing the synchronization overhead caused by locks.

Enzenberger and Müller's lock-free tree parallelization implemented in Fuego removes mutex operations by leveraging the volatile declaration in C++ and the features of IA-32 and Intel 64-bit CPU [6]. While lock-free tree parallelization does not degrade the performance caused by synchronization overhead, some information may be lost when a tree is updated. However, they conclude that the problem of losing information does not affect the strength of Fuego because such a scenario rarely occurs in practice.

2) Tree Parallelization in Distributed-Memory Environments: Tree parallelization has the notoriously difficult issue of tree sharing in a distributed-memory environment, because tree sharing requires network communications among processes. In order to reduce the communication overhead, Gelly et al.'s approach periodically broadcasts only the important part of the search tree among machines where memory is distributed, as in the global transposition table in parallel game-tree search [18]. On the other hand, their approach leverages the aforementioned tree parallelization technique inside a machine where memory is shared [7]. Cazenave and Jouandeau's multiple-runs parallelization algorithm [8] can be considered as a special case of the algorithm in [7] by regarding the children of the root as an important part of the search tree and running processes separately even when memory can be shared.

C. Root Parallelization

In root parallelization [8], a process does not share any information about its search tree with the other processes, and each process performs completely independent MCTS with a different random seed. When root parallelization determines the next move to play, one process collects from the other processes the number of visits to the root's children and computes the total sum of visits for each child. It then selects a move leading to a child with the largest number of visits.

While root parallelization is simple to implement, each process does not build a search tree larger than sequential MCTS. In other words, there may be many duplicate trees searched by processes and these trees may be a cause of a large search overhead. However, since playouts performed at the same node with different seeds often return different outcomes, each process has a local tree that contains characteristics that are different from the others. Moreover, in root parallelization, communication overhead is incurred only at the start and end of MCTS. Because root parallelization suffers from neither synchronization nor communication overhead, it allows more playouts per second with more CPU cores.

Chaslot et al. compared performance in a shared-memory environment with 16 CPU cores, and showed that root parallelization not only performed better than tree parallelization but also often achieved super-linear speedups in improving the strength of their program [9]. They conjecture that this is because the shallower search performed by root parallelization makes it easier for MCTS to escape from local optima than the deeper search performed by sequential MCTS. However, neither theoretical nor empirical evidence was provided to support their conjecture.

IV. Overview of Fuego

Fuego is being developed mainly by people at the GAMES Group of the University of Alberta [4], [5]. At present, it is one of the best programs. It won first prize in 9×9 Go and second prize in 19×19 Go [19] at the Computer Olympiad in 2009. More importantly, because Fuego is an open source program, researchers can implement new techniques on top of Fuego for free.

Fuego's search engine is based on the UCT algorithm with several state-of-the-art enhancements explained in Section II. Additionally, as described in Section III, the source code of Fuego includes lock-free tree parallelization in shared-memory environments.

V. Root Parallelization Based on Majority Voting

In order to measure the performance of root parallelization, we implemented not only the standard strategy based on the maximal number of the sum of visits to select the next move, but also a modified strategy based on majority voting. We call these approaches *average selection* and *majority voting*, respectively. This section describes majority voting.

One drawback of average selection is that it select a move that is better than the other moves merely *on average*. Let m_1 and m_2 be moves. Assume that m_1 is visited by several processes frequently but visited by the other processes much less frequently. If m_2 is the second best move for each process but the total number of visits to m_2 is larger than that to m_1, average selection selects m_2. However, from the viewpoint of each process, m_2 is less reliable than the move that is locally best for that process.

In majority voting, each process independently performs MCTS with a unique random seed as in average selection. However, when selecting the best move, each process votes for a move that it wants to play. Each process selects the best move with the largest number of visits at the root node.

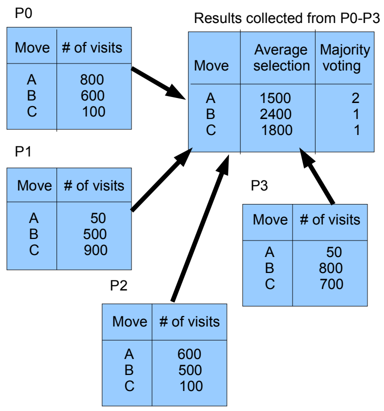


Fig. 1. Example of root parallelization based on average selection/majority voting

We assume that each process runs on a CPU core with the same speed to perform a similar number of playouts among processes. One process collects the result of voting from all the processes, and selects the move with the largest number of votes. If there is more than one move with the largest number of votes, the move having the largest total number of visits is selected.

The idea behind majority voting has been successful not only in machine learning [20] but also in improving the strength of a strong computer shogi (Japanese chess) program [21]. In majority voting each process ignores the locally second best move for which winning ratio is less reliable than that of the locally best move. Although this is a very small difference from average selection, the next section will show the superiority of majority voting to average selection.

An example illustrating selection strategies of average selection and majority voting is shown in Figure 1. Average selection selects move B because the number of total visits to B is $600 + 500 + 500 + 800 = 2400$. On the other hand, majority voting selects move A. While P_0 and P_2 voted A, only P_3 voted B.

VI. Experimental Results

A. Setup

Average selection and majority voting were implemented on top of Fuego version 0.3.2[3] with MPI [22]. Experiments were conducted in both 9×9 and 19×19 Go with komi of 6.5 points and a time limit of 10 seconds per move. While experiments in previous papers were sometimes run with shorter time limits such as 1 second per move (e.g., [6] and Figure 5 in [9]), we did our best to measure the performance under conditions that are as close as possible to real tournaments. When measuring the strengths of two programs, we held a 200-game match to compute the winning percentage. Because the inclusion of its opening book causes Fuego to often play almost identical games, we disabled the opening book for the experiments. In this way, we obtained a variety of games per match, because MCTS has randomness for playouts.

The performance was measured on a cluster of eight machines. Each machine had a dual quad-core 2.33GHz Xeon E5410 with 6MB L2 cache and 16 GB memory (i.e. 8 CPU cores per machine and a total of 64 CPU cores for the cluster).

B. Performance Comparison between Average Selection and Majority Voting

Figures 2 and 3 show the winning percentage of average selection and majority voting against the sequential version of Fuego and MoGo in 19×19 and 9×9 Go respectively. When N CPU cores are available for root parallelization, N processes are invoked without sharing any information even inside a machine, where memory can be shared. The best parameters for Fuego were used for both sequential and parallel versions. Each program has its own playing style and the winning percentage is often influenced by accidentally exploiting weaknesses in the playing style. Experiments were therefore conducted on both self-play games (i.e. Fuego plus root parallelization versus sequential Fuego) and games against sequential MoGo[4].

Average selection and majority voting tend to improve the strength of Fuego in both 19×19 and 9×9 Go. However, our results also imply that root parallelization might not scale well with a large number of processes. This phenomenon is observed more clearly for average selection (see Figure 2(a) and (b) and Figure 3(b)), while the phenomenon also seems to occur in majority voting (see Figure 2(b)). For example, in average selection the winning percentage with 64 cores seems to be lower than that with 32 cores in 19×19 Go (65% with 64 cores versus 70% with 32 cores in self-play and 29% versus 34% against MoGo respectively), although our results with 200 games may contain some statistical noise that makes it difficult to draw a firm conclusion.

Majority voting tends to perform better than average selection especially in 19×19 Go. In the self-play experiment with 64 processes in 19×19 Go, the winning percentage of majority voting was 87%, while that of average selection was 65%. The difference in winning percentage became smaller between majority voting and average selection when they played against MoGo (e.g., 37% for majority voting and 29% for average selection with 64 processes). However, majority voting was still more effective than average selection.

In 9×9 Go, while majority voting was superior to average selection in self-play games, majority voting performed similarly to average selection in games against MoGo.

One interesting note is that the results obtained in this paper are close to those in [7], despite different sequential and parallel algorithms.

Table I shows the winning percentage of sequential Fuego with a longer time limit against sequential Fuego with 10 seconds per move. This table and Figures 2 and 3 indicate that the actual speedup in strength (hereinafter *strength speedup*) is

[3]Because the latest distribution of Fuego is version 0.4.1, our baseline is an older version. Version 0.3.2 was the latest version when we started implementing root parallelization. It was not possible to re-measure the performance of root parallelization with version 0.4.1 because it took a few *months* to measure all the experiments shown in this paper. On the other hand, several state-of-the-art enhancements are already included in version 0.3.2.

[4]The executable code of MoGo is available at http://www.lri.fr/~gelly/MoGo_Download.htm. Release version 3 was used for the experiments.

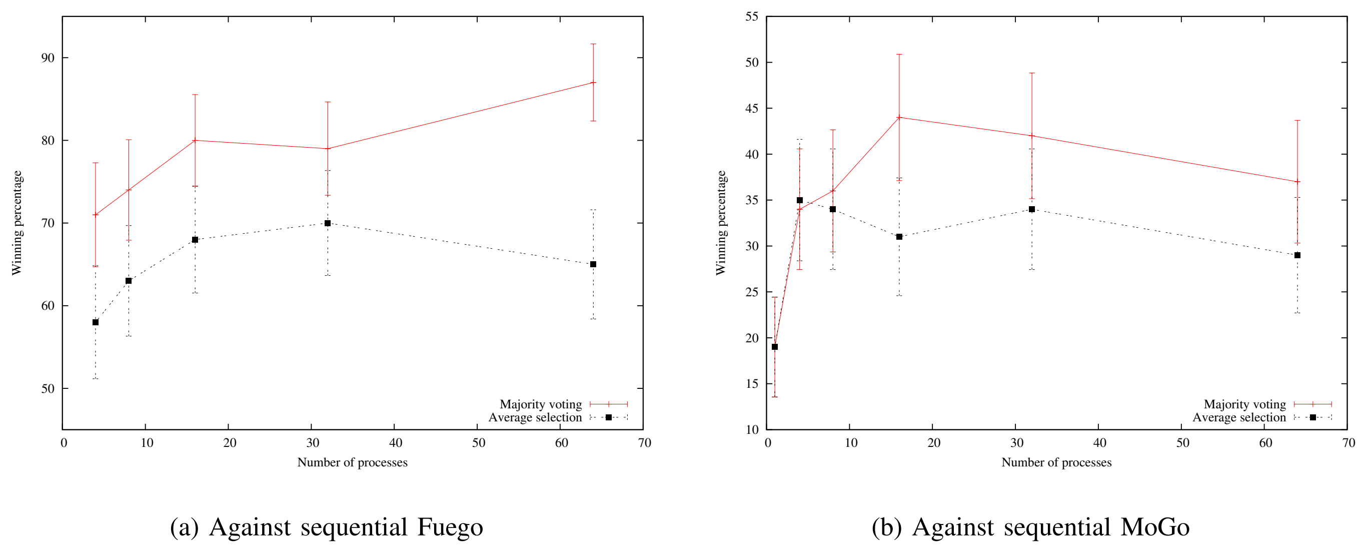


Fig. 2. Winning percentage of average selection and majority voting (19 × 19 Go, 200 games)

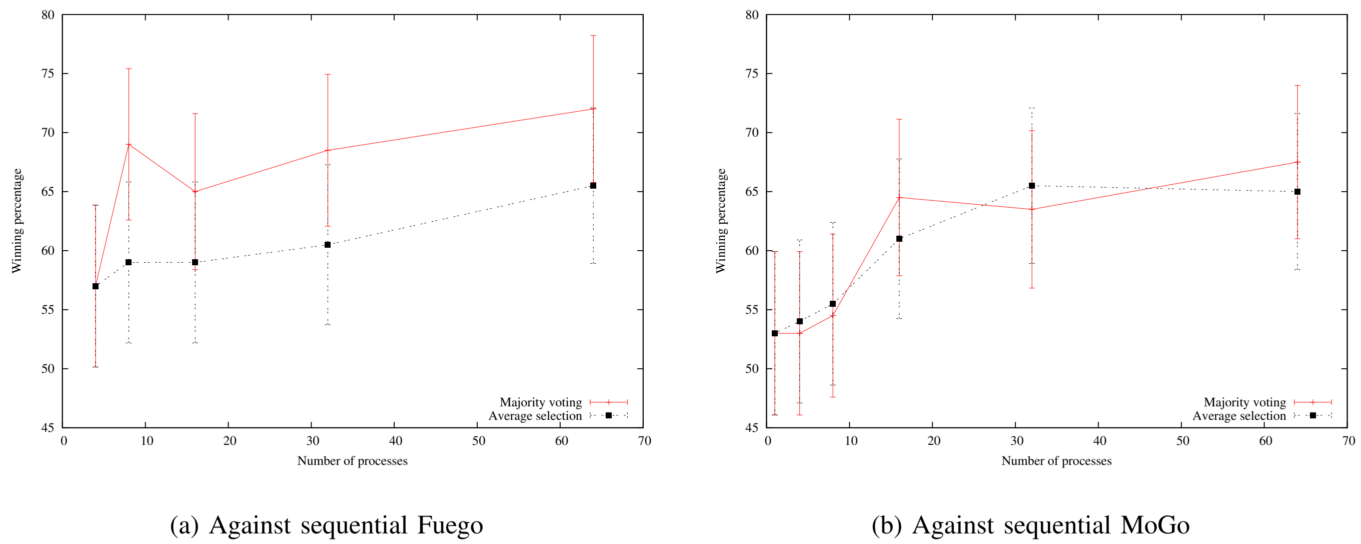


Fig. 3. Winning percentage of average selection and majority voting (9 × 9 Go, 200 games)

TABLE I
WINNING PERCENTAGE OF SEQUENTIAL FUEGO WITH DIFFERENT TIME LIMITS AGAINST FUEGO WITH 10 SECONDS PER MOVE (200 GAMES)

Time limit (seconds)	20	40	80	160
19 × 19	81.5%	94.5%	99.5%	100.0%
9 × 9	66.5%	77.5%	91.5%	92.0%

estimated to be between 2 and 4 even for majority voting with 64 processors in both 9 × 9 and 19 × 19 Go. Our results are different from Chaslot et al.'s findings that root parallelization achieved 6.5-fold and 14.9-fold strength speedups respectively with 4 and 16 processors, although it is not possible to directly compare numbers due to several differences such as sequential programs, board sizes, and time settings. One important difference could be related to time settings. While our time limit was set to 10 seconds per move, Chaslot et al. set it to 1 second. Chaslot et al. describe in [9] that they believe that with longer time limits root parallelization will be less efficient. Additionally, according to Brockington discussion of the importance of selecting a high-performance sequential algorithm as a baseline for measuring the performance of parallel game-tree search algorithms [23], it is important to select a stronger sequential program to measure the strength speedup of root parallelization. For this purpose, we chose Fuego, which is one of the best publicly available computer Go programs. One hypothesis is that if a sequential MCTS algorithm is less efficient, parallel MCTS may have a higher chance of exploiting the weaknesses of the sequential algorithm, resulting in better strength speedup values.

The experiments performed by us and by Chaslot et al. are based on root parallelization in which each processor does not communicate with any others. If processors frequently exchanged information about the root node's children (e.g., a multiple-run variant in [8] and a special case of [7]), root parallelization could further improve the strength speedup.

Combining majority voting with such an improvement is one direction for future work.

We analyzed the moves selected by average selection and majority voting for details to investigate the differences in their playing styles. 200 games were used for this analysis and these games were played by majority voting with 64 processes and sequential Fuego.

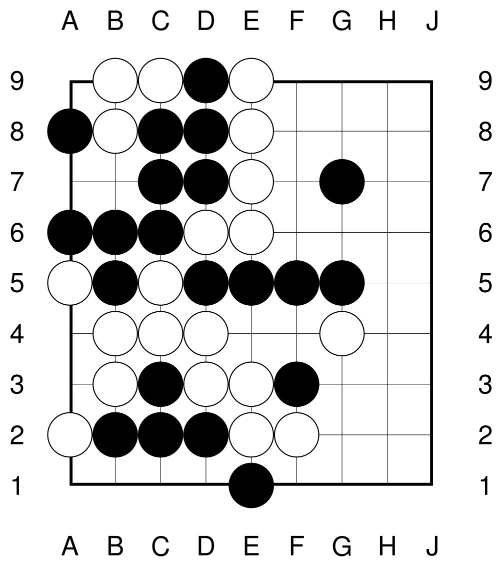


Fig. 4. Example where majority voting selected a better move than average selection (White to play)

The only difference between majority voting and average selection in the implementations was the move selection scheme after each process had performed local MCTS. Therefore, when majority voting selects a move on a position, it is easy to check the move that would be selected by average selection in the same position.

In 9×9 Go, 97% of the moves selected by majority voting were identical to ones chosen by average selection, while this number was 91% in 19×19 Go. In other words, in 9×9 Go majority voting played only one or two moves per game that were different from ones played by average selection. On the other hand, in 19×19 Go, 10-20 moves per game played by majority voting typically did not match those of average selection. Because there is more room in 19×19 Go for root parallelization to improve performance than in 9×9 Go, it is reasonable to observe majority voting behaving similarly to average selection in 9×9 Go but achieving better results in 19×19 Go.

We analyzed positions where majority voting selected different moves from ones chosen by average selection with 64 cores. We prepared F_{tree}, which is a version of lock-free tree parallelization of Fuego with eight threads and a time limit of 30 seconds[5], and checked whether the moves selected by F_{tree} matched ones played by majority voting or average selection for these positions. In 9×9 Go, majority voting matched 40% of the moves played by F_{tree}, while average selection agreed with 23% of F_{tree}'s moves. In 19×19 Go, these numbers were 26% for majority voting and 6% for average selection, respectively. These results provide further evidence of the superiority of majority voting to average selection.

A position where majority voting selected a correct move but average selection played incorrectly with 64 processes is shown in Figure 4. In this example, move A7 selected by majority voting kills the black blocks at the top-left. On the other hand, move A4 chosen by average selection enables Black to play A7 to connect. White loses the capturing race and five white stones at the top are captured.

Table II shows the five best moves computed by each program in the position shown in Figure 4. While A4 was visited most frequently by average selection, only nine processes vote for A4 in majority voting. By considering only the move that was locally best for each process and ignoring the rest, majority voting avoided the case where the number of visits to a move was large merely on average.

C. Performance Comparison against Tree Parallelization

Figure 5 shows the winning percentage of lock-free tree parallelization already implemented in Fuego versus majority voting with 64 processes in 19×19 and 9×9 Go. We set the best parameters such as turning on virtual losses.

Chaslot et al. reported that root parallelization outperformed tree parallelization leveraging local locks and virtual losses with a small number of CPU cores [9]. On the other hand, we confirmed that lock-free tree parallelization outperformed root parallelization even with 64 CPU cores and with majority voting which performed better than average selection. The strength of root parallelization with 64 cores was similar to that of tree parallelization with only 4 CPU cores in 9×9 Go. Moreover, in 19×19 Go lock-free tree parallelization with 4 cores won against majority voting by a large margin. One reason why tree parallelization performed better than root parallelization is explained in the next subsection.

As observed in [6], the winning percentage of lock-free parallelization with 8 cores against majority voting is lower than that with 6 cores in 9×9 Go. Enzenberger and Müller concluded that they could not say with high confidence whether this was a real effect given the statistical error of the experiment. We cannot conclude whether this is caused by statistical error or other reasons either, although in theory faulty updates of the values used to compute RAVE may occur with a low probability[6].

TABLE III
WINNING PERCENTAGE OF 8×8 ROOT PARALLELIZATION BASED ON MAJORITY VOTING (%)

Board size	vs Tree parallelization (8 cores)	vs Sequential MoGo
19×19	76.0	62.5
9×9	67.0	77.0

[5] F_{tree} is stronger than root parallelization. See the next subsection.

[6] According to personal communication with Martin Müller, lock-free tree parallelization scales well up to 16 CPU cores. Therefore, the slight performance degradation with eight cores may be unrelated to these faulty updates.

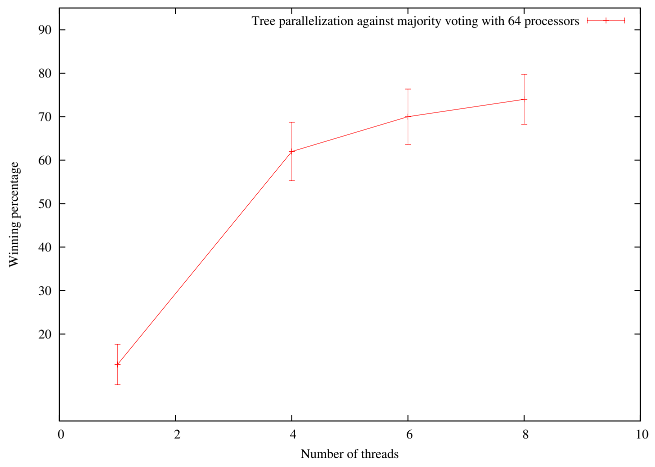
TABLE II
BEST FIVE MOVES IN AVERAGE SELECTION AND MAJORITY VOTING FOR THE POSITION SHOWN IN FIGURE 4

Rank	Moves	Number of visits on average
1	A4	29,416
2	H5	25,984
3	A7	25,917
4	H4	20,460
5	H6	2,618

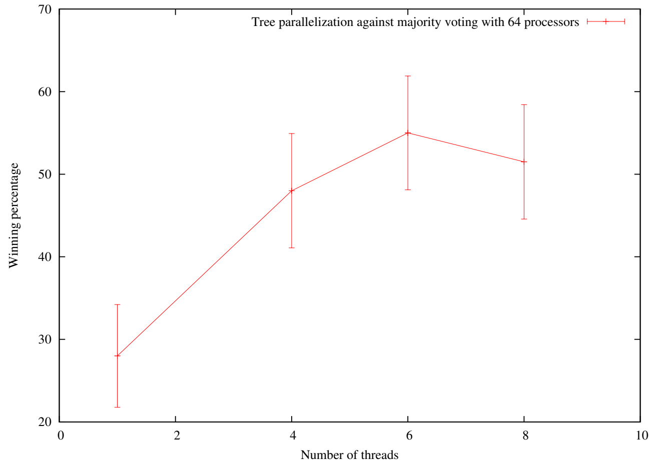
(a) Average selection

Rank	Moves	Number of visits	Number of votes
1	A7	25,917	28
2	H4	20,460	14
3	H5	25,984	13
4	A4	29,416	9
5	H6	2,618	0

(b) Majority voting



(a) 19×19 Go



(b) 9×9 Go

Fig. 5. Winning percentage of lock-free tree parallelization using different number of threads versus majority voting with 64 CPU cores

One way to recover from the weakness of root parallelization is to combine tree parallelization with root parallelization. Lock-free tree parallelization is adapted to build a larger tree that can be shared among eight cores inside a shared-memory machine, while root parallelization is leveraged among eight machines, where memory is distributed and communication is required to exchange information located at different machines. We call this hybrid approach 8×8 root parallelization. Note that a similar approach is presented in [7] to implement tree parallelization on a cluster of machines, although we did not intend to directly compare the performance with their approach. We aimed to clarify the characteristics of root parallelization.

Table III shows the winning percentage of 8×8 root parallelization with a total of 64 cores versus lock-free tree parallelization with 8 cores. Compared with incorporating only root parallelization, the hybrid approach improves the strength of Fuego in both 19×19 and 9×9 Go. While the winning percentages of majority voting with 64 processes against lock-free tree parallelization with 8 cores were 26% in 19×19 Go and 49.5% in 9×9 Go, respectively, 8×8 root parallelization clearly achieved much better results for both sizes (see Table III). Moreover, considering the winning percentage of majority voting with 64 processes versus sequential MoGo (37% in 19×19 Go and 67.5% in 9×9 Go according to Figures 2 and 3), we observed the superiority of 8×8 root parallelization to the simple majority voting algorithm with 64 processes (see Table III again).

D. Detailed Analysis of Move Selection

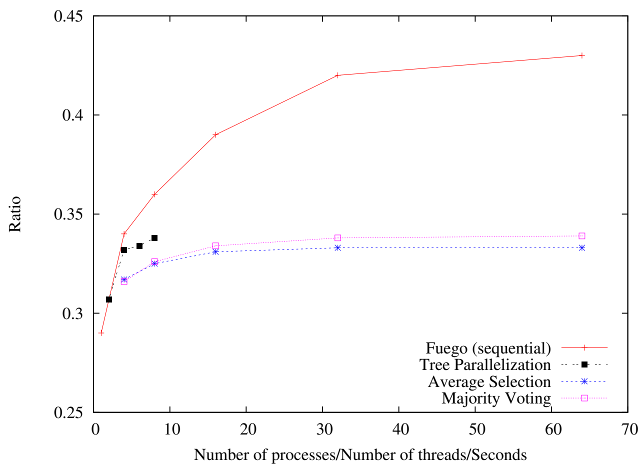
We analyzed the moves selected by root parallelization and lock-free tree parallelization to estimate the scalability of these algorithms. We prepared 400 game records for each board size and extracted from these games 46,193 positions for 19×19 Go and 10,591 positions for 9×9 Go. We also prepared F_{80}, which is sequential Fuego with a time limit of 80 seconds. We regarded the moves selected by F_{80} for these positions as "oracles" and checked how many moves selected by each method matched the oracles. The time limit for sequential Fuego was varied from 1 second to 64 seconds. The time limit for root parallelization and tree parallelization was always set to 1 second[7] but the number of CPU cores was varied from 4 to 64 processes[8] for root parallelization and 2–8 threads for tree parallelization, respectively.

The ratio of the number of selected moves that match the oracles[9] is shown in Figure 6. While 33.8% in 19×19 Go and 61% in 9×9 Go of the moves selected by tree parallelization matched the oracles even with eight threads,

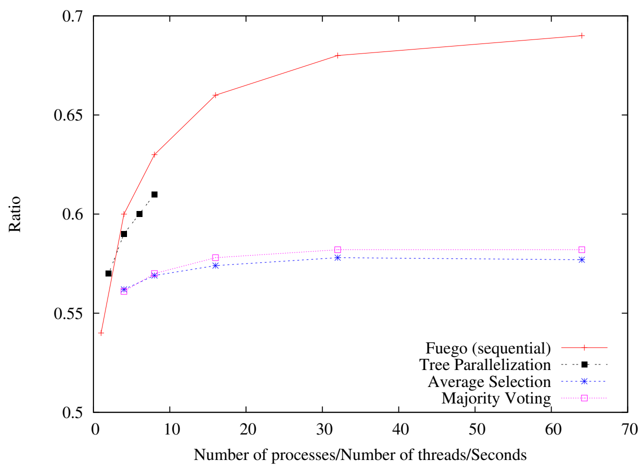
[7]Although the time limits in real tournaments are ideally desirable, limiting the time to 1 second is our best possible condition. Otherwise the time limit for sequential Fuego used as an oracle would need to be much longer than 80 seconds per position, so experiments would take much longer.

[8]Because at least three processes are required for majority voting, we started not with two processes but with four processes.

[9]Although these metrics are apparently different, they can be regarded as the same metric by considering the total time spent by all the CPU cores used by each method. In this way, the lines drawn in each graph can be compared directly.

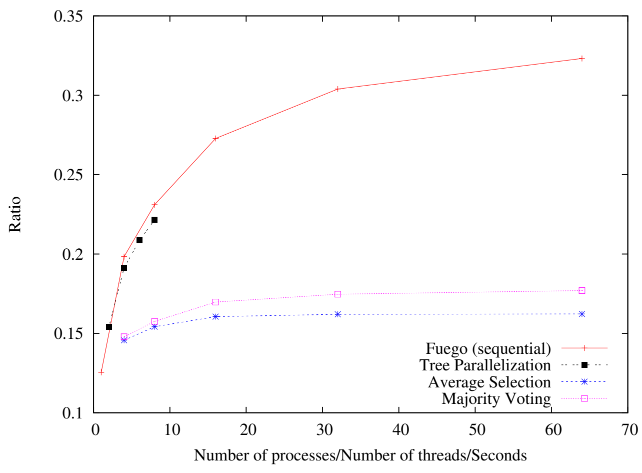


(a) 19×19 Go

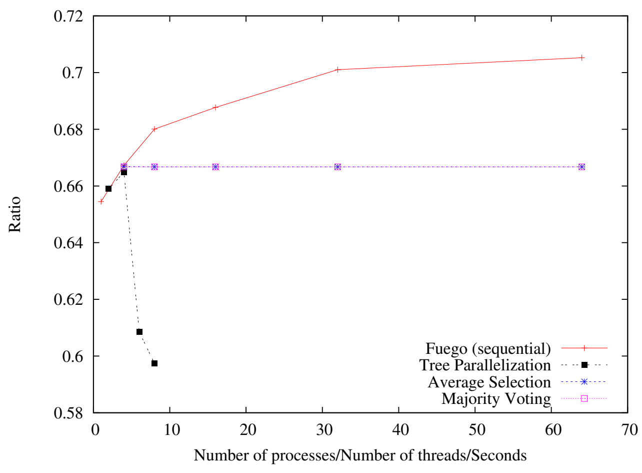


(b) 9×9 Go

Fig. 6. Ratio of the number of moves, which were selected by each method and agreed with oracles, to the total number of positions (4–64 CPU cores for root parallelization, up to 8 cores for tree parallelization, and 1–64 seconds for sequential Fuego)



(a) $0.2 \leq R_{unique} < 0.4$



(b) $0.8 \leq R_{unique} \leq 1.0$

Fig. 7. Ratio of the number of moves, which were selected by each method and agreed with oracles, to the total number of positions (categorized by $R{\cdot}unique$ in 19×19 Go)

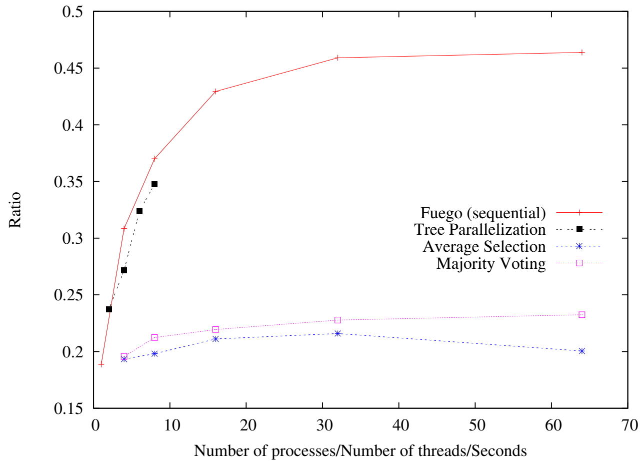
diminishing returns with additional processes were clearly observed for root parallelization for both majority voting and average selection, implying a limitation of root parallelization. The ratio converged to around 0.34 in 19×19 Go and 0.58 in 9×9 Go. We believe that this is evidence explaining why lock-free tree parallelization performs better than root parallelization.

Because 91-97% of the moves selected by majority voting and average selection were identical, the ratio for majority voting was better than that for average selection in both 19×19 and 9×9 Go only slightly in Figure 6 (e.g., 0.34 versus 0.33 in 19×19 Go and 0.582 versus 0.577 in 9×9 Go with 64 processes). However, this slight difference is an important factor to improve the strength of Fuego (see Section VI-B again).

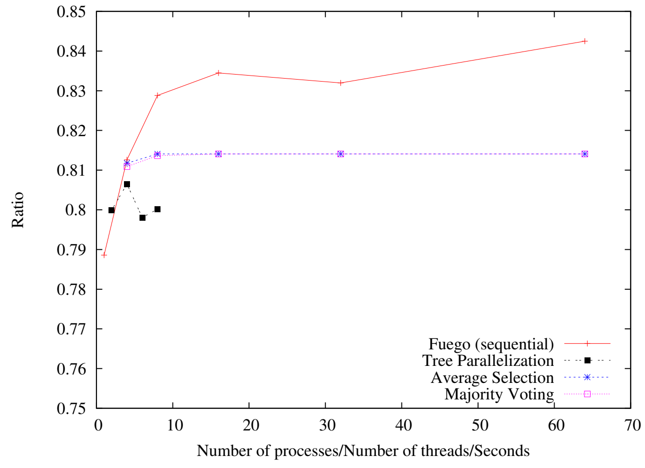
Even if playout contains some randomness, MCTS often selects the same move for a position with different random seeds. We ran sequential Fuego 512 times with a time limit of 1 second for the aforementioned 9×9 and 19×19 positions. Let $V(m)$ be the number of processes that select m as the best move. We define the move uniqueness ratio R_{unique} for a position P in the following way:

$$R_{unique} = \frac{\max_{m \in \text{Legal moves}} V(m)}{512}.$$

On the basis of R_{unique}, we classify the positions used to compare against F_{80} in Figure 6. Figures 7 and 8 show the cases of $0.2 \leq R_{unique} < 0.4$ and $0.8 \leq R_{unique} \leq 1.0$ in 19×19 and 9×9 Go respectively. When R_{unique} was large, the number of moves that matched oracles of F_{80} did not increase with over eight processes in the case of root



(a) $0.2 \leq R_{unique} < 0.4$



(b) $0.8 \leq R_{unique} \leq 1.0$

Fig. 8. Ratio of the number of moves, which were selected by each method and agreed with oracles, to the total number of positions (categorized by $R_{\cdot}unique$ in 9×9 Go)

parallelization. In other words, once a process has mistakenly selected an incorrect move, it is hard for root parallelization to fix the mistake, because only a few processes select the correct move. On the other hand, with small R_{unique}, not only did root parallelization improve the performance but also we observed the superiority of majority voting. Again, this is where majority voting is different from average selection.

While the total ratio for lock-free tree parallelization with 8 cores is similar to that for majority voting with 64 cores in Figure 6(a), tree parallelization won against majority voting by a large margin in 19×19 Go (see Figure 5(a) again). Because the ratio for tree parallelization with additional threads was decreased in the case of $0.8 \leq R_{unique} \leq 1.0$ (see Figure 7(b)), the total ratio for tree parallelization in Figure 6(a) was not very drastically increased. The quality of move selection in the case of $0.2 \leq R_{unique} < 0.4$ shown in Figure 7(a) seems to be a more important factor for tree parallelization to outperform root parallelization.

Lock-free tree parallelization decreased the ratio with additional threads in the case of $0.8 \leq R_{unique} \leq 1.0$. Many moves categorized in the case of $0.8 \leq R_{unique} \leq 1.0$ were ones for playing the endgame. For the game records used to measure experiments, there are many possible alternative moves for accurately playing the endgame. Lock-free tree parallelization tended to select different moves from the other three algorithms.

E. Detailed Analysis of Move Selections with a Large Number of Processes

Figure 9 shows the ratio of moves that were selected by majority voting with up to 512 cores and matched the oracles of F_{80}. The ratio for majority voting stayed at around 0.34 with over 64 processes, clearly indicating that majority voting is unlikely to scale well.

Assume that we have an algorithm $\mathcal{A}$ that can select an oracle if the move selected by at least one process of root

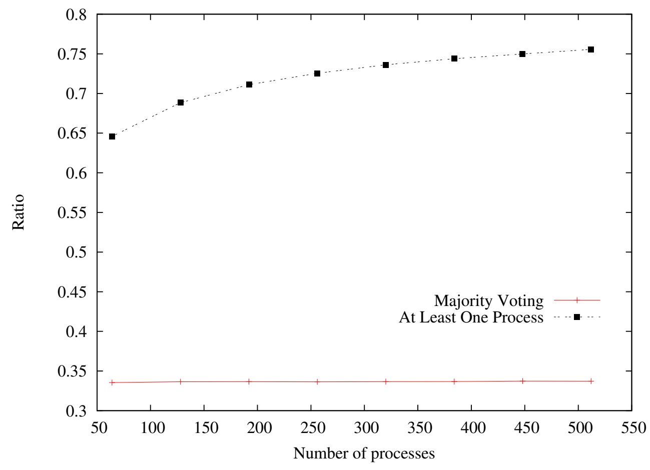


Fig. 9. Ratio of the number of moves that were selected by each method and agreed with oracles in 19×19 Go (up to 512 cores)

parallelization matches the oracle of F_{80}. We can then compute the ratio of the number of moves that match oracles as well as being selected by $\mathcal{A}$. "At least one process" in Figure 9 is the case of $\mathcal{A}$, which is the best scenario for root parallelization. $\mathcal{A}$ indicates that root parallelization still has room to improve the strength of computer Go programs, although developing an algorithm that is close to $\mathcal{A}$ remains an important direction for future work.

VII. Conclusions and Future Work

This paper not only presented a new root parallelization algorithm based on majority voting, but also evaluated the performance of both average selection and majority voting on top of Fuego. Majority voting performed better than average selection, which is encouraging. However, lock-free tree parallelization outperformed majority voting and root parallelization has limited scalability, which are discouraging.

The speedups obtained in our experiments are more modest than Chaslot et al.'s findings in [9], where root parallelization achieved super-linear speedups with fewer CPU cores. We therefore made an extensive analysis to understand the behavior of root parallelization. In this sense, this paper includes important new insights. Besides, because it has extra an overhead, tree parallelization is also expected to eventually reach a point where dramatic performance improvement can no longer be achieved. In this case, majority voting could be a candidate for exploiting orthogonal parallelism to tree parallelization, as we did in our 8×8 root parallelization experiment.

A few extensions to this paper are possible, such as:

- As is shown in Section VI-E, there is still room for improvement in the performance of root parallelization. One extension is to develop a better move selection algorithm that includes an approach based on voting schemes with weights.
- Gelly et al. developed a tree parallelization algorithm in a distributed environment [7], but they showed a limitation on scalability in 9×9 Go. Because we observed that majority voting improved the performance on top of lock-free tree parallelization, majority voting could exploit different aspects of parallelism than their technique in a distributed-memory environment. Combining their approach with majority voting would therefore be an interesting extension, especially to prepare for the time when their approach does not scale well.

Acknowledgment

We would like to thank Adi Botea, Tomoyuki Kaneko, and Kazuki Yoshizoe for their constructive comments on this paper. This research is supported by the JST PRESTO program "High-Performance Artificial Intelligence Systems Based on Massive Parallelism" and the JSPS Global COE program "Computationism as a Foundation for the Sciences".

References

[1] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proceedings of the 17th European Conference on Machine Learning*, ser. Lecture Notes in Computer Science, vol. 4212. Springer, 2006, pp. 282–293.

[2] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte-Carlo Go," INRIA, Tech. Rep. 6062, 2006.

[3] R. Coulom, "Computing Elo ratings of move patterns in the game of Go," *ICGA Journal*, vol. 30, no. 4, pp. 198–208, 2007.

[4] M. Enzenberger and M. Müller, "Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search," University of Alberta, TR 09-08, 2009.

[5] 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 Transactions on Computational Intelligence and AI in Games*, 2010.

[6] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo tree search algorithm," in *Advances in Computer Games 12*, ser. Lecture Notes in Computer Science, vol. 6048, 2010, pp. 14–20.

[7] S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian, "The parallelization of Monte-Carlo planning," in *Proceedings of the 5th International Conference on Informatics in Control, Automation, and Robotics*, 2008, pp. 198–203.

[8] T. Cazenave and N. Jouandeau, "On the parallelization of UCT," in *Proceedings of the Computer Games Workshop*, 2007, pp. 93–101.

[9] G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo tree search," in *Proceedings of the 6th International Conference on Computer and Games*, ser. Lecture Notes in Computer Science, vol. 5131, 2008, pp. 60–71.

[10] 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*, vol. 4, no. 3, pp. 343–357, 2008.

[11] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *Proceedings of the 24th International Conference on Machine Learning*, 2009, pp. 273–280.

[12] T. A. Marsland and F. Popowich, "Parallel game-tree search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, no. 4, pp. 442–452, 1985.

[13] J. Schaeffer, "Distributed game-tree searching," *Journal of Parallel and Distributed Computing*, vol. 6, no. 2, pp. 90–114, 1989.

[14] M. Brockington and J. Schaeffer, "APHID game-tree search," in *Advances in Computer Chess*, vol. 8, 1997, pp. 69–91.

[15] H. Kato and I. Takeuchi, "Parallel Monte-Carlo tree search with simulation servers," in *Proceedings of the 13th Game Programming Workshop*, 2008, pp. 31–38.

[16] T. Cazenave and N. Jouandeau, "A parallel Monte-Carlo tree search algorithm," in *Proceedings of the 6th International Conference on Computer and Games*, ser. Lecture Notes in Computer Science, vol. 5131, 2008, pp. 72–80.

[17] M. G. Brockington, "A taxonomy of parallel game-tree search algorithms," *ICCA Journal*, vol. 19, no. 3, pp. 162–174, 1996.

[18] R. Feldmann, "Spielbaumsuche auf massiv parallelen systemen," Ph.D. dissertation, University of Paderborn, 1993, English translation titled *Game Tree Search on Massively Parallel Systems* is available.

[19] M. Müller, "Fuego at the Computer Olympiad in Pamplona 2009: a tournament report," University of Alberta, TR 09-09, 2009.

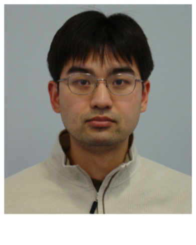
[20] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[21] T. Obata, M. Hanawa, and T. Ito, "Consultation algorithm in brain game - effect of simple majority system (in Japanese)," in *Information Processing Society of Japan (IPSJ) SIG Game Informatics Colloquium*, vol. 22, 2009.

[22] M. Snir and W. Gropp, *MPI: The Complete Reference*. MIT Press, 1998.

[23] M. G. Brockington, "Asynchronous parallel game-tree search," Ph.D. dissertation, Department of Computing Science, University of Alberta, 1998.

Yusuke Soejima has been working for Nintendo Co., Ltd. since 2010. He received the M.Sc. degree from the Department of Mathematical and Computing Sciences at Tokyo Institute of Technology in 2010. He is interested in AI in games especially computer Go.



Akihiro Kishimoto is an Assistant Professor in the Department of Mathematical and Computing Sciences at Tokyo Institute of Technology as well as a researcher at Japan Science and Technology Agency. He received the B.Sc. degree from the University of Tokyo in 1997 and the M.Sc. and Ph.D. degrees from the University of Alberta in 2001 and 2005, respectively. His research interests include artificial intelligence and parallel computing. Particularly he is interested in developing high-performance game-playing programs and planning systems.

Osamu Watanabe received in 1989 B.Sc., in 1982 M.Sc., and Dr. of Engineering in 1986, all from Tokyo Institute of Technology. Presently, he is with Tokyo Institute of Technology, a Professor at Department of Information and Computing Sciences. His current interests are randomness in computation, design and analysis of randomized algorithms, and computational complexity. Members of IEICE, IPSJ (fellow), and EATCS.