Parameter-Free Tree Style Pipeline in Asynchronous Parallel Game-Tree Search

Shu Yokoyama[1], Tomoyuki Kaneko[1⋆], and Tetsuro Tanaka[2]

[1] Graduate School of Arts and Sciences, The University of Tokyo
[2] Information Technology Center, The University of Tokyo

Abstract. Asynchronous parallel game-tree search methods are effective in improving playing strength by using many computers connected through relatively slow networks. In game position parallelization, the master program manages a game-tree and distributes positions in the tree to workers. Then, each worker asynchronously searches the best move and evaluation for its assigned position. We present a new method for constructing an appropriate master tree that provides more important moves with more workers on their sub-trees to improve playing strength. Our contribution introduces two advantages: (1) being parameter free in that users do not need to tune parameters through trial and error, and (2) efficiency suitable even for short-time matches, such as one second per move. We implemented our method in chess with a top-level chess program (Stockfish) and evaluated playing strength through self-plays. We confirmed that playing strength improves with up to sixty workers.

Keywords: game-tree search, distributed computing, chess, parallel search

1 Introduction

Parallelization of game-tree search has been extensively studied to improve game programs' playing strength, especially in chess and its variants. Several efficient methods have been developed in hardware parallelization [2, 4], in thread-level parallelization [14], and for tightly connected computers [5, 1, 12, 18].

Recently, as grid computing has become popular, new approaches utilizing computational resources placed in different locations connected through wide-area networks have been proposed. Game position parallelization (GPP) [16] is one such method actually showing steady improvements in playing strength. In that method, a local computing unit (we call it a *worker*, even though it could be a cluster of tightly connected computing nodes) assumes a position assigned to it. Each worker runs its own game-tree search independently, and then, the master integrates workers' results to reach a decision.

This study presents pipeline GPP (P-GPP), which extends both Optimistic Pondering [8] and GPP, by improving worker management. In P-GPP, positions are assigned to workers on the basis of realization probabilities [17] automatically

⋆ A part of this work was supported by JSPS KAKENHI Grant Number 25330432.

acquired from game records and a playing program. This automation frees users from the need to tune heuristic parameters, whereas existing methods need many configuration parameters. Experiments demonstrate P-GPP's effectiveness with up to sixty workers, with results comparable to those shown in the literature [8]. Therefore, P-GPP deserves further study of both its effectiveness and usability in terms of being parameter free.

The remainder of this paper is organized as follows. The next section reviews related research. The third section introduces the GPP framework, and the fourth section presents the P-GPP method. The fifth section presents our experimental results in chess and improvements in playing strength through self-play. The last section provides our concluding remarks.

2 Related Work

2.1 Parallelization of Alpha–Beta Pruning

State-of-the-art sequential algorithms on game-tree search have been built upon alpha–beta pruning [13] with many enhancements. Basically, playing strength improves when a program searches more deeply, assuming that adequate evaluation functions have been provided. Therefore, various parallel search algorithms have been developed to improve strength by exploring game-trees in a shorter time by using more processors. The best solution depends on users' environments, because there is a well-known trade-off in the design of parallelization; an increase in shared information among processors increases pruning effectiveness, while communication to share information inevitably incurs overheads that degrade efficiency. State-of-the-art sequential algorithms prune branches aggressively (e.g., late move reductions[3]) by utilizing information of the tree explored already, e.g., transposition tables, $\alpha\beta$-windows, killer moves, and history tables.

In parallel search methods in shared memory environments, (e.g., Principal Variation Splitting [14], Dynamic Tree Splitting[4]), transposition tables are naturally shared. However, in effective parallelization in distributed environments, only a part of a transposition table is shared [3, 8] because the cost for full sharing is not beneficial overall. Still, in major approaches including YBWC and its enhancements [5, 18], APHID [1], and TDSAB [12], $\alpha\beta$-windows or equivalent information are shared in frequent communication. Therefore, they work more effectively in a network with higher quality, e.g., Infiniband, than with an ordinary one. Also, many practical systems incorporated hybrid parallelization including hardware (e.g., Deep Blue [2] and Hydra [4]).

2.2 Integration of Computing Resources through the Internet

Recently, accessing computational resources placed elsewhere has become easy, if one permits relatively high latency and limited bandwidth to reach them, e.g.,

[3] http://www.glaurungchess.com/lmr.html (Last access: February 2015).
[4] https://www.cis.uab.edu/hyatt/search.html (Last access: February 2015).

through ordinary Ethernet or wide-area networks. Hence, several new methods for utilizing such resources have been developed to improve playing strength further. Owing to network limitations, these methods are designed to work with little inter-node communication. For example, majority voting requires only communication regarding a position to search and vote for a move [15].

In Optimistic Pondering [7, 9, 8] and GPP [16] which were developed independently, workers are assigned distinct positions and search independently without sharing information. In Optimistic Pondering, the goal is to increase "ponder-hit" rate as well as to begin pondering as many plies earlier as possible. Pondering gains additional thinking time to deepen the search when pondering hits, i.e., the position assigned to a worker is actually realized in the game. In GPP, the goal is to conduct minimax search cooperatively by integrating search trees explored by hundreds of workers [16].

A notable advantage of these approaches is that they lend themselves well to combination with existing parallelization methods. The effectiveness of Optimistic Pondering in integrating workers running in YBWC mode was demonstrated in GridChess [8]. A combination of majority voting, GPP, and shared memory parallelization on each worker is used for playing shogi in Akara [10].

3 Game Position Parallelization

This section introduces the details of Game Position Parallelization (GPP), on which our work, P-GPP, is constructed. GPP is based on a master/worker model, with a typical worker being a universal chess interface (UCI) [5] chess engine.

3.1 Master Tree

GPP conducts minimax search by integrating the results obtained locally by workers. The master constructs *a master tree* for task assignments for workers. The root of a master tree corresponds to the current position, and the number of nodes of the master tree must be the number of workers available. Suppose that we have six equivalent workers A, B, C, D, E, and F. The master constructs a game-tree rooted at the current position, which has five leaves. Fig. 1 shows an example of a master tree. A node depicted in a rounded rectangle corresponds to a position similar to those in usual game-trees. A leaf "others" enclosed in a rectangle is a special position, in which the position is the same as that of the parent but for which moves to be searched are limited, excluding moves already covered by its siblings. For example, at position "d4" in the figure, the worker C does not search moves `Nf6` and `e6` because workers A and B are working on them, respectively. Therefore, at each internal node, the children cover all moves without duplication.

Each worker then independently execute game-tree search for a node and periodically reports the best move (more precisely, a sequence of best moves, the

[5] http://www.shredderchess.com/chess-info/features/uci-universal-chess-interface.html (Last access: February 2015).

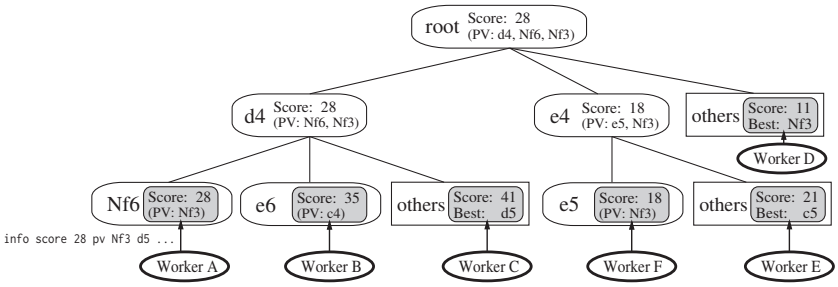


Fig. 1: A master tree with the initial position at the root. Each leaf has a worker assigned. For an internal node "d4", workers A, B, and C work on it cooperatively. Workers A and B assume responsibility for two child positions that are assumed to be the best ("Nf6") and second best ("e6") successors, and C assumes responsibility for the remaining moves. The best move and its score with respect to the max player are kept at each leaf shown in a gray rounded rectangle. The scores and best moves of internal nodes are computed, as in minimax search.

principal variation, PV) with its evaluation score. The master then integrates the best moves and scores. Communication required in this process is supported by standard game protocols, UCI for chess and universal shogi interface (USI)[6] for shogi. For example, the command `go searchmoves` restricts the moves searched at an "others" leaf. The use of a text protocol improves software modularity.

GPP aims to achieve steady improvements in playing strength with hundreds of workers in slow networks [16]. Therefore, neither transposition tables nor $\alpha\beta$ windows are shared. If the network quality is sufficient for them to be shared, the methods introduced in Section 2.1 are preferable to GPP.

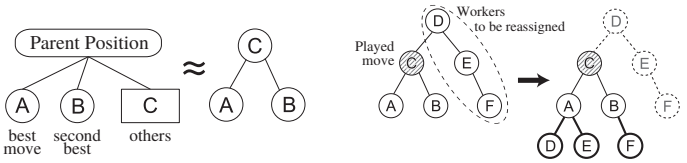
3.2 Tree Growth and Tree Style Pipeline

To improve playing strength in GPP, a master tree must be carefully constructed such that a more relevant move should have more workers than less important moves. The problem here is the difficulty in identifying relevant moves in advance.

Having the master tree of the previous position available while playing a game provides two advantages: Relevant branches can be estimated from the information stored in the tree, and workers working on common nodes between the previous and new trees can continue searching without interruption. In this study, we call this idea *tree style pipeline*. The idea has already been adopted in two different methods: tree of pondering pipeline in Optimistic Pondering [9], and a recent GPP method [11].

In tree style pipeline, nodes in the previous master trees that remain effective in the new tree are preserved. The remaining nodes are discarded, and their

[6] http://www.glaurungchess.com/shogi/usi.html (Last access: February 2015).



(a) Simplified expression (b) Tree style pipeline

Fig. 2: Simplified notation (Fig. 2a): Merging a node for other moves into the parent, makes the right tree equivalent to the left one. Either tree represents a portion of the master tree shown in Fig. 1, where "Parent Position" corresponds to the position "d4". Tree-style pipeline (Fig. 2b): The left tree is the simplified notation for Fig. 1. When the board situation changes, the master tree grows. Workers working on positions that are not descendants of the new root (D, E, and F) are collected and assigned to the newly created leaves.

corresponding workers are assigned to new expanded leaves, as shown in Fig. 2b. As a simple illustration, note that we draw a tree merging an "others" leaf to its parent, as explained in Fig. 2a. An advantage in tree style pipeline is that a worker on a shaded node in Fig. 2b continues searching without interruption in the transition from the previous tree to the new one. In GPP, a normal leaf sometimes becomes an "others" leaf, when its child is expanded. Node A in Fig. 2b is in such a situation. A worker working on such a node is stopped and immediately restarted with restriction of moves already expanded (i.e., D and E in this example). This restart's negative effect must be very limited because the contents in a transposition table are preserved in each worker. Workers outside the common tree are collected and used to grow relevant branches of the tree. In expanding a leaf of the tree, the current best move at the leaf is a promising candidate to expand. However, it is still not clear which leaves (including "others" nodes) are to be expanded and how many new workers each leaf is worth. This study's primary contribution is to present a new criterion and procedure for this problem.

3.3 Comparison with Similar Systems

Table 1 summarizes related work similar to our work. GridChess combines Optimistic Pondering and tree style pipeline. The main difference between that work and ours is minimax integration. Optimistic Pondering focuses on pondering and does not perform minimax backup in a master tree. Consequently, tasks assigned to workers are not disjoint in Optimistic Pondering, i.e., it does not have any "others" leaf in GPP, such as worker C in Fig. 2a. Lacking minimax backup can create a problem. Suppose that a move at the root seems promising within a certain search depth, but actually is a blunder that deeper search can reveal. Through constructing a larger master tree by investing more workers, GPP might detect it earlier than a single worker does. However, Optimistic Pondering cannot see it until the search of the root worker reaches a sufficient

Table 1: Comparison of distributed search methods and systems similar to GPP. The first column is the name of the method or system. The second (third) column indicates whether minimax backup (tree style pipeline) is used. The fourth column describes when new leaves in the master tree are expanded. "New root" means when the root of the master tree changes. The fifth column describes what information source is used to select leaves to be expanded.

		Minimax	Pipeline	Growth	Source
GridChess [8]	(chess)	-	Yes	Anytime	PV
GPP [16]	(shogi)	Yes	(Yes)	New root	Shallow search
P-GPP	(chess)	Yes	Yes	New root	Previous PV+Hash

depth, even if many additional workers are involved. Additionally, Optimistic Pondering changes its master tree more dynamically. While frequent changes might improve playing strength, many heuristic parameters need to be tuned, regarding when to start and stop pondering a position.

The idea of GPP was first developed in connection with shogi, integrating more than 300 ordinary computers and reported in the literature [16]. The original version does not have tree style pipeline. Instead, a shallow search is used to determine the moves to be expanded at each step in the construction of a master tree. Because this shallow search decreases the available time for the primary search, our work introduced an alternative method for growing a master tree. Additionally, heuristic parameters r_0 and r were used in the assignment of workers; for the n-th promising move at a position having N workers, $r_0 r^{n-1} N$ workers are assigned. The values were $r_0 = 1/4, r = 3/4$ for the root, and $r_0 = r = 1/2$, otherwise. Later, many heuristic parameters including domain-dependent ones were introduced, when it was used in human–computer shogi matches [10, 11]. Therefore, our work is the first method incorporating both GPP and tree style pipeline, simultaneously, eliminating heuristic parameters.

4 P-GPP: Pipeline GPP with Parameter-Free Approach

This section presents our method (P-GPP) extending GPP. When the board is changed by a move played by a program or its opponent, the master tree for GPP must be updated so that the node corresponding to the new position becomes the root in a new tree. P-GPP constructs its new tree using the following steps, to find the best tree with respect to playing strength:

1. Unreachable nodes from the new root position are removed from the tree, and the corresponding workers are collected (e.g., node D, E, and F in Fig. 2b).
2. The greedy algorithm presented here determines the number of workers for each node (except for an "others" leaf), using realization probabilities.
3. For each node l and the number of workers n identified in the previous step, a concrete sub-tree rooted at l having n nodes is created, considering the transposition table of the worker at l.

The main contributions of this study are the new methods for steps 2 and 3 presented in the following subsections. For initiating the pipeline process, we define the initial master tree as a tree having only its root as the starting position. Alternatively, opening books can be used.

4.1 Utility of Master Tree based on Realization Probability

We introduce the method assuming that the realization probability is available for all nodes, and we later discuss how we apply it in practice. The utility $U(T)$ of a master tree T is the summation of the depth weighted by its realization probability:

$$U(T) = \sum_{v \in V(T)} p_v d_v, \tag{1}$$

where $V(T)$ is the set of vertices in the tree T, d_v is the depth of node v and p_v is its realization probability. GPP works effectively, if a position included in the master tree will be reached in an actual game in the future, and it is more beneficial when the position is further from the root to have more thinking time before it manifests. Therefore, if we can predict the future, then narrow, deep trees are preferable for that purpose. However, when we miss a prediction, many workers must be reallocated, and their results do not contribute to the playing. This means that the total number of workers collected in the future must be minimized. Thus, the expectation over the probability in Eq. (1) is adopted.

The realization probability of a node, defined as the product of the transition probability of each move [17], is the probability that the corresponding sequence of moves is actually played. By definition, the realization probability of the root is one. We also assume that the summation of the transition probability of all legal moves is one in each position. We ignored such edges in counting the depth in computing the utility that represent "others" moves, because they do not reduce the search space compared with normal edges. In our simplified notation of a tree (see Fig. 2a), each internal node corresponds to a leaf led by all other moves. Hence, the summation of the realization probabilities for all the nodes in a simplified tree is always one.

4.2 Greedy Growth Algorithm

Assuming that the realization probability of each node is available, we present a simple greedy algorithm based on iteratively adding a node having the largest realization probability. Fig. 3 shows a step of the greedy algorithm, where P_i is the transition probability of a rank i move, and the values are from our experiments listed in Table 2. This algorithm yields a tree of maximum utility in Eq. (1). The proof is based on the change in the utility when a node is added (and symmetrically removed). Let T_1 be a tree, x a leaf in T_1, and T_0 the tree with x removed from T_1. The differences between T_0 and T_1 are node x which exists only in T_1 with probability p_x and parent x', which exists both in T_0 and T_1 but $p_{x'}$ in T_0 is greater than that in T_1 by p_x. Thus, for the probability p_x, the

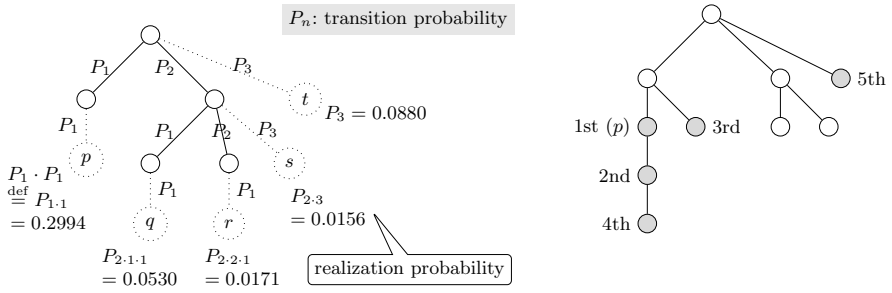


Fig. 3: Example of greedy steps: for a tree drawn with a solid line with candidates, each of which is a dotted circle with its probability, node p having the largest probability is added. After the fifth step, the tree in the right figure is obtained.

depth of x is added to $U(T_1)$ while the depth of x' is added to $U(T_0)$, where the difference in the depth of x and x' is 1. Therefore, we have $U(T_1) - U(T_0) = p_x$.

Here is a sketch of the proof by contradiction. Let T be the tree yielded by our greedy algorithm. We assume that there exists a tree T' having the maximum utility that is strictly better than that of T, i.e., $U(T') > U(T)$, $T \neq T'$ and $|V(T)| = |V(T')|$. Then, we select two nodes, v^* from $T \setminus T'$ and v from $T' \setminus T$. Let $v^* \in T \setminus T'$ be the node added at the earliest step. There exists a node v in $T' \setminus T$, such that $p_{v^*} > p_v$.[7] Thus, when we generate T'' by replacing node v in T' with node v^*, we have $U(T'') > U(T')$. This contradicts the assumption that T' has the maximum utility. □

4.3 Realization Probability in Practice

For the realization probability in P-GPP, we used the empirical transition probability that expert players play a move of n-th rank when moves are ranked by a worker program. While the class of a move (e.g., check) is used in the original literature [17], we believe the ranks determined by the scores in the previous master tree are more reliable sources here. The probabilities regarding the rank, the only parameters that P-GPP requires, are obtained from a worker program and game records, as shown in Table 2 in the experiments discussed later.

Having the score in each node, including "others" in the tree, the rank and consequently the realization probability are identified for each node by virtually sorting nodes by their scores. However, the ranks and probabilities are not available for newly added nodes at this step. Therefore, we virtually add an n-th node as needed without knowing the actual n-th best move leading to the node. Then, we count the number of virtual nodes added in this process for each node and recovers the original tree discarding these virtual nodes.

[7] By the definition of the greedy algorithm, we have $p_{v^*} \geq p_v$ for any $v \in T' \setminus T$. In the special case that $p_{v^*} = p_v$ for all $v \in T' \setminus T$, utility $U(T')$ equals $U(T)$ by the definition of v^*, and this contradicts the assumption that $U(T') > U(T)$.

Table 2: Frequency of move w.r.t. the rank, evaluated with Stockfish

Rank	1	2	3	4	5	6	7	8	9	10	11+
Frequency	0.5472	0.1769	0.0880	0.0522	0.0293	0.0247	0.0211	0.0128	0.0082	0.0110	0.0284

Now the problem is to construct an effective sub tree with n nodes rooted at the leaf, given a node l and the number n to be added. If $n = 1$, it is sufficient to expand the best move. Otherwise, this can be achieved by extracting the n most valuable positions from the transposition table of the corresponding worker (if l is a leaf) or the worker of its "others" leaf (if l is an internal node). We used the depth searched under a position for the criterion in this selection. These positions are sent from each leaf to the master every time the new master tree is constructed. Because the estimated data size is about one kilobyte for 32 entries, it can be expected to consume negligible time and network resources.

5 Experiments

To evaluate P-GPP, we implemented our method in chess. We first show the relative frequency (or empirical probability) of moves with respect to their rank, obtained from game records. Then, by using the frequency as the transition probability, self-play experiments are conducted.

5.1 Configurations

We adopted Stockfish DD[8] as a worker program, because it is an open source program and is expected to be one of the strongest chess programs. We added the function of reporting information described in Section 4.3, extending the UCI protocol. Each worker and the master are connected via standard TCP sockets. The master is implemented in C++ with the boost/asio library. For a worker, a utility program *netcat* is adopted as a proxy connecting stdin/stdout and a TCP socket. To simulate a distributed environment, we used at most 64 cores in two computers each of which is equipped with two Intel Xeon E5-4650 processors. Stockfish ran as a sequential program using a single thread. Each worker was allowed to use 32MiB (Stockfish uses 16 bytes per position) for its transposition table.

5.2 Empirical Probability with respect to the Rank of a Move

Table 2 shows the frequency of moves played for each rank. Twelve game records played between DeepBlue and Kasparov consisting of 1 091 plies were used. For each position, all moves are scored using fifteen-depth search and sorted to get the rank. We classified the ranks in eleven classes, from first to tenth, and the eleventh or greater. The result shows 54.7% for the first-ranked move and 81% in the third rank.

[8] https://s3.amazonaws.com/stockfish/stockfish-dd-src.zip

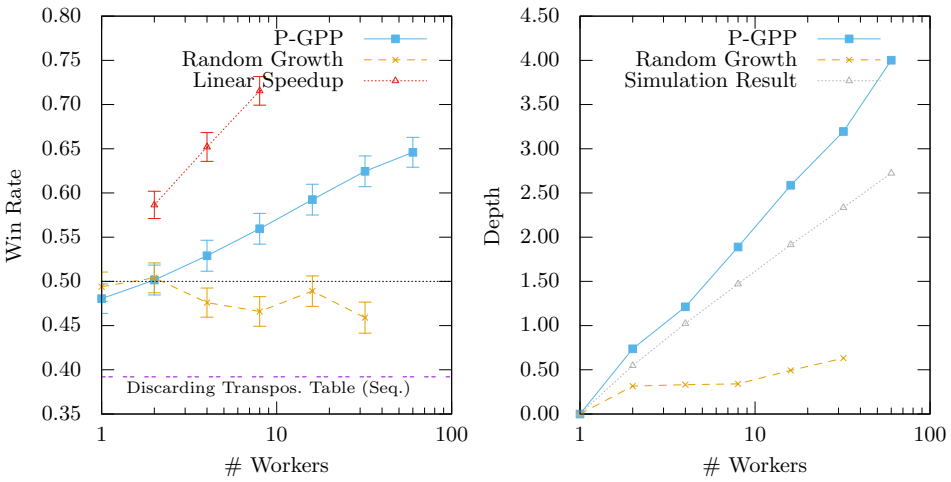
5.3 Improvements in Strength

We conducted self-play experiments and showed the winning probability of several variations of the presented system against a sequential program. The sequential program is nearly the same as the original Stockfish, except that it ponders the current position instead of a future predicted position. The reason for the adjustment is to average the effect of ponder-hits and misses. Additionally, the sequential program is nearly the same as the presented method with a single worker. A program (XBoard) managed matches in judging and recording the results and a program (Polyglot)[9] is used to connect XBoard and Stockfish. The book used was `performance.bin`.[10] The opening was randomly chosen from the book by Polyglot. The win rate here is defined as the probability of wins plus half of the probability of draws, following the literature [6].

To consider the communication overhead in distributed environments, we imposed a thinking-time penalty on the proposed program. While the sequential program was given 1 000ms to think per ply, the presented system P-GPP was given only 950ms. We believe that 50ms is more than sufficient for the communication in the presented method. In addition to P-GPP, we measured the win rate of "Linear Speedup" and "Random Growth." The former is the sequential program given n-times thinking time instead of using n workers.[11] This program gives the upper bound of the win rates gained by the ideal parallelization without overhead. The latter is a variation of P-GPP, ignoring transposition tables; it adds the position after the best move and randomly picked up $n-1$ positions, when adding n nodes to a leaf at the step in Section 4.3. For each configuration, 1 000 games were played alternating black and white.

Fig. 4a shows the win rate of the parallel programs against the sequential one. The horizontal axis indicates the concurrency in the log-scale, and the error bars indicate as 95% confidence interval. In P-GPP, the win rate increases with the number of workers. With a single worker, the win rate was 48.1%, not even reaching 50%. This was apparently caused by the thinking-time penalty. With 32 and 60 workers, our method achieved 62.5% and 64.6% win rates, respectively. The improvements did not reach those in the linear speedup, but they are similar to those reported in Optimistic Pondering built upon the cluster Toga [8]. Therefore, we conclude that our method scaled reasonably at least up to 60 workers. On the other hand, "Random Growth" did not improve the playing strength. In the case of a single worker, the win rate was even, as the two players are the same. However, it became weaker with four or more workers. These results show the importance of master trees. When the master tree is random, only the worker working on the root contributes to playing strength. Moreover, the transposition tables of the workers tend to be filled with irrelevant positions, possibly degrading strength. To examine this result further, we measured the win rate of such a sequential program, which clears its transposition table every

[9] Version 1.4w29 http://www.geenvis.net/polyglot1.4w29.zip
[10] http://wbec-ridderkerk.nl/html/downloada/lacrosse/performance.rar
[11] The opponent was configured not to use this additional time in pondering.



(a) Win rates

(b) Utility of trees

Fig. 4: Left (4a): win rates of parallel systems in self-play. Right (4b): observed utility. The average number of played plies included in the master trees, in self-play for the player's turn, with a "Simulation Result" that is $U(T)$ of the trees built virtually using only the greedy algorithm with the realization probability.

time it plays a move. The win rate was only 39.2%, explaining the contribution of the transposition table to strength.

Fig. 4b shows the utilities of the master trees constructed in self-play experiments, with simulated utilities considering only the realization probabilities listed in Table 2. The observed utilities are the average maximum depth of a position reached in actual games. In P-GPP, the utilities increase along with the concurrency and go beyond the simulated results, most likely because there are sometimes fewer legal moves in some positions than the branching factor of the simulated tree, and it is easier for Stockfish to predict moves played by the same program than to predict moves played by Kasparov's or by Deep Blue. The utilities did not reach 1.0 in "Random Growth."

6 Conclusion

We demonstrated that P-GPP, a new asynchronous parallel game-tree search method, works effectively in chess. P-GPP has two advantages: it is parameter-free in that users do not need to tune parameters through trial and error, and it is suitably efficient even for short-time matches. We confirmed that playing strength improves with up to sixty workers. The win rates are comparable to those of an existing method [8]. Therefore, we believe that P-GPP is a simple and promising alternative to existing methods. Interesting future work would involve scalability up to hundreds of workers.

References

1. Brockington, M.: Asynchronous Parallel Game-Tree Search. Ph.D. thesis, University of Alberta (1998)
2. Campbell, M., Hoane, Jr., A.J., Hsu, F.h.: Deep Blue. Artificial Intelligence 134(1–2), 57–83 (Jan 2002)
3. Donninger, C., Kure, A., Lorenz, U.: Parallel brutus: the first distributed, fpga accelerated chess program. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. pp. 44– (April 2004)
4. Donninger, C., Lorenz, U.: The chess monster hydra. In: Becker, J., Platzner, M., Vernalde, S. (eds.) Field Programmable Logic and Application, Lecture Notes in Computer Science, vol. 3203, pp. 927–932. Springer Berlin Heidelberg (2004)
5. Feldmann, R.: Game Tree Search on Massively Parallel Systems. Ph.D. thesis, University of Paderborn (1993)
6. Heinz, E.A.: New self-play results in computer chess. In: Marsland, T.A., Frank, I. (eds.) Computer and Games. pp. 262–276. No. 2063 in LNCS, Springer-Verlag, Hamamatsu, Japan (Oct 2001)
7. Himstedt, K.: An optimistic pondering approach for asynchronous distributed game-tree search. ICGA Journal 28(2), 77–90 (2005)
8. Himstedt, K.: Gridchess: Combining optimistic pondering with the young brothers wait concept. ICGA Journal 35(2), 67–79 (2012)
9. Himstedt, K., Lorenz, U., Möller, D.P.F.: A twofold distributed game-tree search approach using interconnected clusters. In: Luque, E., Margalef, T., Benitez, D. (eds.) Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5168, pp. 587–598. Springer (2008)
10. Hoki, K., Kaneko, T., Yokoyama, D., Obata, T., Yamashita, H., Tsuruoka, Y., Ito, T.: Distributed-shogi-system Akara 2010 and its demonstration. International Journal of Computer & Information Science 14(2), 55–63 (2013)
11. Kaneko, T., Tanaka, T.: Distributed game tree search and improvements – match between hiroyuki miura and gpsshogi –. IPSJ Magazine 54(9), 914–922 (aug 2013), (In Japanese)
12. Kishimoto, A.: Transposition table driven scheduling for two-player games. M.Sc. Thesis, University of Alberta (January 2002)
13. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning 6(4), 293–326 (1975)
14. Marsland, T.A., Popowich, F.: Parallel game-tree search. IEEE Transactions on Pattern Analysis and Machine Intelligence 7, 442–452 (1985)
15. Obata, T., Sugiyama, T., Hoki, K., Ito, T.: Consultation algorithm in computer shogi - a move decision by majority. In: Computers and Games – 7th International Conference (CG2010). pp. 156–165. No. 6515 in LNCS, Springer-Verlag (2011)
16. Tanaka, T., Kaneko, T.: Massively parallel execution of shogi programs. In: The Special Interest Group Technical Reports of IPSJ. 2, vol. GI-24, pp. 1–8 (2010), (In Japanese)
17. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 145–152 (2002)
18. Ura, A., Yokoyama, D., Chikayama, T.: Two-level task scheduling for parallel game tree search based on necessity. Journal of information processing 21(1), 17–25 (jan 2013)