Journal of Information Processing Vol.23 No.1 9–19 (Jan. 2015)

[DOI: 10.2197/ipsjjip.23.9]

Regular Paper

Dynamic Prediction of Minimal Trees in Large-Scale Parallel Game Tree Search

Akira Ura[1,†1,a)] Yoshimasa Tsuruoka[1,b)] Takashi Chikayama[1,†2,c)]

Received: February 21, 2014, Accepted: September 12, 2014

Abstract: Parallelization of the alpha-beta algorithm on distributed computing environments is a promising way of improving the playing strength of computer game programs. Search programs should predict and concentrate the effort on the subtrees that will not be pruned. Unlike in sequential search, when subtrees are explored in parallel, their results are obtained asynchronously. Using such information dynamically should allow better prediction of subtrees that are never pruned. We have implemented a parallel game tree search algorithm performing such dynamic updates on the prediction. Two kinds of game trees were used in performance evaluation: synthetic game trees and game trees generated by a state-of-the-art computer player of shogi (Japanese chess). On a computer cluster with 1,536 cores, dynamic updates actually show significant performance improvements, which are more apparent in game trees generated by the shogi program for which the initial prediction is less accurate. The speedup nevertheless remains sublinear. A performance model built through analyses of the results reasonably explains the results.

Keywords: distributed computing, game tree search, performance modeling

1. Introduction

Most game-playing programs perform game tree search to decide the move to play for a given position (i.e., game state). The alpha-beta algorithm [10] is arguably the most widely used game-tree search algorithm for chess-like games. Large-scale parallelization of the alpha-beta algorithm on a distributed computing environment is a promising way of improving the playing strength of the programs. For example, Deep Blue, which is a well-known Chess machine that defeated the then World Chess Champion, performed parallel alpha-beta search on a 30-node RS/6000 SP computer with 480 chess chips [3].

The alpha-beta algorithm significantly reduces the search space by pruning subtrees, but, given a search tree, one of certain subsets of nodes must be visited to complete the search. Such a subset of nodes is called a *minimal tree* [1]. Note that there can be multiple minimal trees in a search tree. Since minimal trees are yet to be known to the program during the search, a minimal tree needs to be predicted in actual search if we are to reduce the number of visited nodes.

Parallelization of the alpha-beta algorithm is challenging because parallelized programs tend to search unnecessary nodes in subtrees that would have been pruned in the sequential search. When multiple parts of the tree are searched in parallel, these tasks have to be started before knowing the results of others which could have been obtained in the sequential search. Several programs attempted to visit only those nodes in one of the minimal trees by suspending search of nodes unlikely to be included in the minimal tree [2], [4], [7]. The minimal tree is predicted using partial search results. Thus, it is important to utilize the results of partially completed search for minimal tree prediction, and dynamically update the prediction based on information incrementally obtained by other tasks.

Game tree search is conducted in an iterative deepening fashion, that is, the tree is inspected incrementally deeper, so as to predict a minimal tree based on the results of a shallower search. If subtrees were inspected independently and only the final results are used, results obtained during shallower search would not be used in prediction of a minimal tree.

Brockington has proposed a method to utilize results of shallower search by subtasks in prediction of a minimal tree [2]. The effectiveness of the idea on the performance has not been thoroughly evaluated, however. In this work, we have performed experiments using a computer cluster with 1,536 cores to evaluate the idea.

Two kinds of game trees were used: synthetic game trees and game trees generated by a state-of-the-art computer player of shogi (Japanese chess). The results show that the dynamic updates of a predicted minimal tree are especially effective for the game trees generated by the shogi player program.

This paper is structured as follows. Section 2 introduces the alpha-beta algorithm and describes related work on parallel alpha-beta algorithms. Next, the implementation of our parallel game tree search algorithm is explained in Section 3. Experimental results are then reported in Section 4. Factors of the sublinear speedup are analyzed in Section 5. Lastly, we summarize the

[1] Graduate School of Engineering, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[†1] Presently with FUJITSU LABORATORIES
[†2] Presently with UHM
[a)] ura@logos.t.u-tokyo.ac.jp
[b)] tsuruoka@logos.t.u-tokyo.ac.jp
[c)] chikayama@logos.t.u-tokyo.ac.jp

work and describe future work in Section 6.

2. Related Work

2.1 Alpha-Beta Algorithm

The alpha-beta algorithm [10] is a game tree search algorithm that computes the min-max value of a game tree, while pruning the edges that are guaranteed to have no effect in the min-max value. A pseudo-code of the alpha-beta algorithm is shown in **Fig. 1**. The alpha-beta pair is called a search window, which is used as the thresholds for pruning subtrees. Narrower search windows can prune more subtrees. The function Evaluate() returns the static evaluation of a given game position. The function SortChildren() returns a list of child positions of a given position. The list is sorted so that promising children come first.

2.1.1 Iterative Deepening

An internal iterative deepening search helps sorting the children. On line 4 in Fig. 1, a shallower search is performed to tell the most promising child position, i.e., the one that is expected to give the best min-max value among siblings. With iterative deepening, the first element of clist on line 5 is the child obtained by AlphaBeta() on line 4. Other children are also sorted so that promising children come first.

2.1.2 Transposition Table

A transposition table is used, but it is omitted in Fig. 1. Transposition tables store the results of an already finished search to avoid searching subtrees below the same position more than once. The return value of AlphaBeta() is the exact evaluation of the position if the min-max value is within the search window. Otherwise, the function returns the alpha or beta value. The value indicates an upper or lower bound of the min-max value. Transposition tables are looked up to know whether the valuation of a node resides within the search window or not, for which stored results may or may not be used.

2.2 Parallel Alpha-Beta Algorithms

Parallel alpha-beta algorithms are divided into two groups, synchronized and asynchronous. Synchronization means that processes wait other processes searching sibling nodes to finish. In general, synchronization algorithms search fewer nodes than asynchronous algorithms, but synchronized algorithms make more processors idle due to synchronization.

Young Brothers Wait Concept (YBWC) [4] is a successful synchronized parallel alpha-beta algorithm and has been studied well (e.g., Refs. [3], [5], [11], [18]). At each node, YBWC first searches the most promising child. It is only after the search of the most promising child finishes that YBWC starts a search in all the rest in parallel. YBWC is effective because the best child is often identified as the most promising. In this case, the information obtained in the search of the most promising child narrows the search window optimally. The narrowed search window leads to effective pruning in the succeeding search of other children. YBWC avoids searching nodes not in a predicted minimal tree by waiting for the completion of the search of the nodes in it.

A problem with YBWC is that synchronization may degrade the performance by making many processes idle [2]. This problem becomes critical when iterative deepening is performed in order to sort child nodes because the shallower search is also a synchronization point. To resolve this problem, some approaches have been proposed to reduce the required synchronization [9], [20]. However, these approaches may drastically increase the number of nodes visited because, like YBWC, the ordering of children is fixed even if the most promising child is found to be less promising during the search.

Dynamic Tree Splitting (DTS) [7] is an algorithm that attempts to predict a minimal tree. DTS maintains a *confidence* score for each node, indicating the degree of likelihood that all of its children have to be inspected. DTS updates the confidence score of a node each time search in its child node finishes. This update can be regarded as an update of the predicted minimal tree. However, the update is performed only when the search of a child node is completed. Results of a shallow search in child nodes are not used. Furthermore, the algorithm is designed for a shared memory architecture and not suitable for distributed environments.

Asynchronous algorithms have also been studied although there are fewer studies than on synchronized ones. Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search (UIDPABS) [13] is a basic asynchronous algorithm. UIDPABS distributes child subtrees of the root node to processes and each process keeps searching them within a time limit using the iterative deepening method. Kaneko proposed a similar algorithm more suitable for large-scale environments [8]. Some subtrees are further split and processes are assigned to subtrees. More processes are assigned to more promising children to search them deeper. **Figure 2** depicts examples of the process assignment in these two algorithms. These algorithms may visit many unnecessary nodes because they use no information about minimal trees.

Asynchronous Parallel Hierarchical Iterative Deepening (APHID) [2] is an asynchronous algorithm. APHID uses a

```
1  int AlphaBeta(position, depth, alpha, beta){
2    if(depth == 0 || position is a terminal position)
3      return Evaluate(position);
4    AlphaBeta(position, depth−2, alpha, beta);
5    clist = SortChildren(position);
6    foreach(child of clist){
7      alpha = max(alpha, −AlphaBeta(child, depth−1, −beta, −
           alpha));
8      if(beta <= alpha) return beta;
9    }
10   return alpha;
11 }
```

Fig. 1 Pseudo-code for the alpha-beta algorithm.

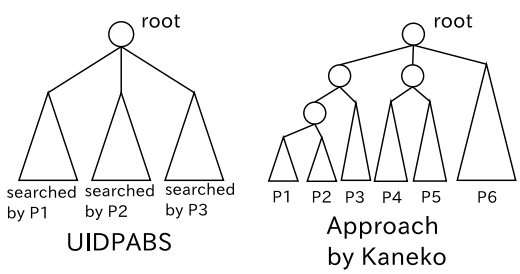


Fig. 2 Examples of process assignment in UIDPABS and the approach by Kaneko.

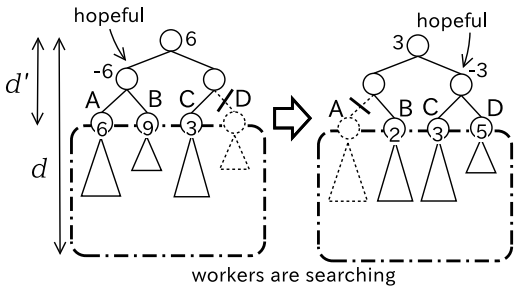


Fig. 3 Examples of an update of the master's tree in APHID.

master-worker model. Let d be the total search depth. The master repeatedly performs tree search to a certain depth d', each repetition of which is called a *pass*. Leaf nodes in the master's tree are regarded as tasks and sent to workers. The master continues the search using the static evaluation values as the values of the leaves without waiting for the results from workers. Workers search the subtrees rooted at the leaf nodes with the depth of $d - d'$, using iterative deepening. After each worker has finished one iteration of its shallow search, it reports the result to the master and then starts the next iteration, which goes down a little deeper. After receiving the results, the master can use them in place of the static evaluation values in the subsequent passes. This may allow better prediction of the minimal tree and change the form of the master's search tree, making the master send workers new tasks. Some tasks are stopped when they are found likely to be pruned.

One of the most important characteristics of APHID is that the prediction of the minimal tree can be promptly updated using new results obtained from the workers. This typically happens when the child that looked most promising has been found to be less so, making another child the most promising. **Figure 3** shows an example of the update. The figure on the left shows the situation in which the left child of the root is more promising than the right child. The subtree D is not searched at this point because pruning is anticipated. The figure on the right shows the situation in which the score of the subtree B has dropped to 2 from 9 after a deeper search finished. In this case, the right child of the root has become more promising. As a result, the form of the tree changes. The subtree D should be searched instead of the subtree A.

As shown in the above example, dynamic updates using shallow results may be effective, but its impact on the performance was not thoroughly evaluated in Ref. [2]. In addition, APHID itself has not been evaluated on systems with more than 64 processes.

Although this paper focuses on dynamic updates of the predicted minimal tree to avoid searching unnecessary nodes, it is also important to avoid searching the same position twice by sharing results of search in subtrees among processes. Several approaches to sharing the search results in distributed environments have been discussed. In the implementation of YBWC, each process keeps a part of the results. To obtain the result for a node, a process sends a message to the owner process of the part of the information. In the implementation of APHID, processes share small portion of results by periodically communicating with each other. As another approach to share the information, Transposition table Driven work Scheduling (TDS) [15] was used in implementation of parallel alpha-beta algorithms by Refs. [9], [18]. In TDS, search tasks for a position are always sent to the same process using the hash value of the position.

3. Implementation

We have implemented a parallel game tree search program based on the idea that the prediction of the minimal tree should be dynamically updated using tentative results obtained during the search. The implementation of APHID served as a rough guideline for the implementation of our program, such as the master-worker model and repetitive search in the master. Implementation details are changed, however. The main difference is in when information updates are utilized: (1) the master notifies workers as soon as the form of its search tree has changed; (2) workers promptly absorb the information and reflect it to their behavior.

Our implementation is similar to that of APHID described in Section 2. Therefore, we first describe further details. The master of APHID distinguishes *uncertain values* and *decided values*. When the master visits leaf nodes of its tree that have not been visited in any previous passes, their static evaluations are used as the *uncertain* values of the nodes. When the master visits the leaf nodes in a subsequent pass, it can use the results reported from workers if any. The values are still regarded as uncertain if the search depth of the worker is smaller than $d - d'$. In contrast, the search results obtained with the depth $d - d'$ are regarded as *decided*. The master continues its search until no uncertain values are found in an entire pass.

We now describe parts of our implementation not directly inherited from APHID. The master stores its search tree in memory. Moves are generated only once at a node when it is visited for the first time. If the move generation is costly, this shortens the time required for a single pass and enables the master to send new information to workers more frequently. Making passes quicker is important especially in large-scale environments because the master's tree should be large enough to generate an adequate number of tasks.

The master sends tasks at every pass even when they are the same as those of the last pass. This enables workers to promptly prioritize the tasks visited in the current pass over those not visited. Furthermore, the master sends *pass messages* when it finishes a pass. On receiving a pass message, the worker can discard the tasks that it has not received during the pass. For example, in the case of the figure on the right in Fig. 3, workers discard task A after the pass messages. Note that the master does not have to send messages to explicitly stop A.

Workers should promptly detect changes in the predicted minimal tree of the master. For this purpose, workers frequently check message arrivals even during search. If a received message indicates that the task a worker is running is less important than another task, the worker aborts the current task and starts the most important task. To realize this, we used a method proposed in Ref. [20]. When a worker receives a task, it suspends its search, inserts the received task into its priority queue, and then resumes its search. If the priority of the received task is higher than that of

```
1  int MasterSearch(position, depth){
2    while(true){
3      score = MasterAlphaBeta(position, depth, −INF, INF, −
           INF, INF);
4      if(no uncertain values have been found) return score;
5      SendPassMessages();
6      RecvMessagesFromWorkers();
7    }
8  }
9  int MasterAlphaBeta(position, depth, a1, b1, a2, b2){
10   if(position is a terminal position) return Evaluate(position);
11   if(depth == d−d')
12     return EvalAndSendTask(position, depth, a1, b1, a2, b2);
13   if(position is visited for the first time)
14     MasterAlphaBeta(position depth−2, a1, b1, a2, b2);
15   clist = SortChildren(position, a1, b1);
16   foreach(child of clist){
17     score = max(a1, −MasterAlphaBeta(child, depth−1, −b1,
           −a1, −b2, −a2));
18     a1 = max(a1, score);
19     if(no uncertain values were found in this child)
20       a2 = max(a2, score);
21     if(b1 <= a1) return b1;
22   }
23   return a1;
24 }
```

Fig. 4 Pseudo-code for the master program.

the suspended task, it starts the received task instead of resuming the suspended task. The efforts on the aborted task are not wasted thanks to transposition tables.

3.1 Details of the Master Program

Figure 4 shows a pseudo-code for the master program. INF is an integer large enough. The master executes passes in the function MasterSearch(). If no uncertain values have been found in a pass, the master finishes the search and returns the score on line 4. Otherwise, the master sends pass messages to all the workers to inform that the master has finished a pass, on line 5. The master receives messages from workers and updates the information about the leaf nodes of the master's tree on line 6 before it starts the next pass. When the master's tree does not change in this pass, the master does not have to start the next pass until receiving new information in messages from the workers.

The function MasterAlphaBeta() is a modified version of AlphaBeta() in Fig. 1. There are three major differences.

One difference is that MasterAlphaBeta() maintains two search windows. One (a1 and b1) is the *pass window* used to execute the pass and the other (a2 and b2) is the *task window* used for the search windows of worker tasks. The pass window is updated by both uncertain values and decided values on line 18 while the task window is only updated by decided values on line 20. For these updates, the master labels internal nodes with *decided* or *uncertain* depending on whether the search of the node down to the required depth has been completed or not. A pass window is an "estimated" window that may change in the next pass of the master, and workers' search with this window may become useless. Instead of using pass windows, we used task windows to decide search windows of tasks.

Another difference is that the master sends tasks if the remaining search depth is $d - d'$ on line 12. The function EvalAndSendTask() first checks the deepest result obtained so far for the position in a similar way to APHID. If the result can be used with the pass window, the function returns the result. Otherwise, the function checks shallower results in turn. If all the results cannot be used with the pass window, the function returns the static evaluation. The master also sends a task to a worker when the returned value is uncertain. The search window of the task is set to the task window (a2 and b2). Notice that the master sends tasks even if the tasks have already been sent in previous passes.

The last difference is how child positions are sorted. Iterative deepening is performed only when positions are visited for the first time on line 14. In the shallower search, the master does not send tasks to workers and only static evaluation values are returned on line 12. If positions are already visited, the master can use obtained results to sort the children. When the master can use decided values of some children with the pass window, it updates both the pass window and the task window using the decided values, and then searches other children.

A task consists of the following: a position, a required search depth ($d - d'$), a search window, and a *signature*. Note that an identical position can generate multiple tasks which are different from each other in search windows. A signature is a list of numbers identifying the path from the root to the node [9]. Each number represents the rank of the child in the order of predicted promises of the children. Signatures were used as priorities of tasks in Ref. [18]. They are designed so that tasks can be executed in the depth-first-search order. We used signatures as a part of priorities.

The master selects a worker to send a task from the viewpoint of load balancing. That is, the master attempts to reduce the number of idle workers that have no tasks. To realize this, the master counts the number of positions sent to each worker in every pass. Two or more tasks with the same position are counted only once, because that does not mean heavy load. When the position of a task has not been sent in any previous passes, the master selects the worker which has the smallest number of positions in the last pass and increments the count for the worker. If there are multiple candidate workers, a worker is selected in a round-robin fashion. When the position was already sent in previous passes, the master normally selects the owner worker to which the position was sent because an identical position should be searched by only one worker to utilize the transposition table. The owner worker of the position is changed, however, to balance the load, if the following three conditions are met: (1) the position has not been sent in this pass; (2) there is at least one idle worker which has received no positions in the last pass; (3) the owner worker of the position has already received at least one position in this pass. In this case, the idle worker becomes the new owner worker and the position will be sent to this new owner also in the succeeding passes.

3.2 Details of the Worker Program

Figure 5 shows a pseudo-code for the worker. A worker maintains a priority queue of tasks (taskQueue in Fig. 5). The function RecvMessages() checks and receives messages from other processes. Message checking and receiving is also performed during the execution of WorkerAlphaBeta(). When a worker receives a

```
1  void WorkerSearchLoop(){
2    while(the master continues the search){
3      do{
4        RecvMessages();
5      }while(taskQueue.empty());
6      task = taskQueue.top();
7      d = GetSearchDepth(task);
8      if(task need not be executed){
9        tasksQueue.dequeue();
10       continue;
11     }
12     score = WorkerAlphaBeta(task.position, d, task.alpha,
           task.beta)
13     if(this search was not aborted){
14       SendResult(task.position, d, score);
15       SetTaskResult(task.position, d, score);
16       if(d == task.depth) TaskQueue.dequeue();
17     }
18   }
19 }
```

Fig. 5 Pseudo-code for the worker program.

task, it enqueues the task into its task queue. When a worker receives a pass message, it removes the tasks from its task queue that were not received after it had received the last pass message. The currently running task WorkerAlphaBeta() is executing may not be the highest priority task due to such alterations in the task queue. In such a case, the current execution of WorkerAlphaBeta() is aborted. A worker retrieves the task with the highest priority on line 6. The function GetSearchDepth() checks whether the position has already been searched deep enough with the same or a wider search window. If so, the task is ignored on lines from 8 to 11. Otherwise, this function adds 2 to the depth with which the position was already searched. For this, the worker stores results for positions obtained through finished search (on line 15). The function WorkerAlphaBeta() is identical to AlphaBeta() in Fig. 1 except that RecvMessages() is inserted before line 4 of Fig. 1. The search result is returned to the master on line 14 when the search completes without abort. The task is removed from the queue if the finished search depth is equal to $d - d'$. Otherwise, the task remains in the queue and a deeper search will be performed in succeeding repetitions.

Sharing the transposition table between workers is an important issue in parallel game tree search. We have implemented a scheme based on APHID's idea that only the results of deep searches are shared. We call this scheme a *partial sharing scheme*. Workers form a communication ring and share the deep results by circulating them. When a worker stores a result into its transposition table, it also stores a pointer to the transposition table entry into another table (called a *pointer table*) to easily specify updated entries. The first worker sends the entries pointed to from this pointer table to the next worker and then clears its pointer table. Every sent entry includes the ID of the worker that added the entry. When a worker receives entries from the previous worker, it inserts them into its own transposition table. The worker then sends both the received entries and entries pointed to from its pointer table to the next worker. Before sending them, the worker removes entries whose worker ID is equal to the next worker's ID, because all workers have already shared such entries.

The priorities of tasks are decided as follows. First, tasks received after the last pass messages are prioritized over the others. Second, tasks with shallower completed search are prioritized. Third, the signatures of tasks are checked: tasks preceding in the depth-first tree traversal order are prioritized. Lastly, tasks with narrower windows are prioritized.

4. Evaluation

We used two kinds of game trees for evaluation: synthetic game trees and game trees generated by the move generation method and evaluation function of Gekisashi [19], one of the strongest shogi programs. The former is for making the analysis of the search algorithm simpler and the latter is for evaluating it under more realistic situations.

4.1 Synthetic Game Trees

We used incremental random trees used in Ref. [16], which assigns random values to edges of trees. The evaluation value of a leaf node is the sum of the values of the edges on the path from the root [*1]. While a uniform distribution was used in Ref. [16], we used a more realistic distribution for the random values. The distribution parameters were decided based on statistics of differences of Gekisashi's static evaluation values between parent nodes and their children.

Nodes in the game tree are visited more than once in repeated search. The same node should always have the same value at every visit. This is realized by a method similar to the one described in Ref. [12]. The synthetic game trees we generated are not just directed acyclic graphs but real trees, i.e., two nodes through different paths always indicate different states. Note that the transposition table is still necessary, because the same state is visited many times.

The branching factor may considerably affect the performance of parallel search programs. YBWC's performance is heavily dependent on the branching factor while APHID's performance is almost independent [2]. We generated game trees with a branching factor of 5 or 20 – all nodes of each tree have the same number of children. Note that the average number of legal moves for a shogi position is about 80, but the effective branching factor, i.e., the number of meaningful moves in an actual shogi position, is usually much smaller [6].

The generated game trees are accurately ordered. That is, the child found to be the best in a shallower search is highly probable to be the best child in the following deeper search. The performance of parallel search is higher with stronger ordering than for not accurately ordered trees because programs can more easily predict a minimal tree and thus the number of visited nodes can be reduced.

4.2 Game Trees Generated by a Shogi Program

The alpha-beta search program we implemented for more realistic game trees bases on features of Gekisashi: its static evaluation function, move generation, and child node ordering method.

[*1] These synthetic trees are also known as N-Game trees [17].

Gekisashi has a variety of other features for further improvement of its performance, which were not used in our experiments for ease in analyzing the behavior of the parallel search. We call the game tree structure this program searches *shogi trees* in this paper.

We limited the branching factors to 5 and 20. At each node, all of its children are generated and sorted, and then the most promising 5 or 20 children are chosen. Note that some nodes have fewer children than these designated numbers (e.g., positions where the king is in check). Since all children need to be generated first, the search speed is slow (approximately 5,000 or 10,000 nodes are visited per second).

There are three main differences between synthetic game trees and shogi trees. First, two nodes through different paths can indicate the same game position in a shogi tree. Sharing evaluation of positions among workers may be beneficial in avoiding duplicated search. Second, shogi trees are not ordered so accurately as synthetic trees. **Figure 6** shows the difference in the accuracy of the ordering between synthetic trees and shogi trees. When the branching factor is 5, only 82.3% of the child nodes judged to be the best with the search depth of 14 are also found to be the best with depth 16, which is 89.8% for synthetic trees. With the branching factor of 20, it is as low as 66.7% between the depths of 6 and 8, while it is 81.9% for synthetic trees. Third, search time varies more largely between subtrees than that for synthetic trees. This may be mainly because the numbers of children vary among nodes.

4.3 Experimental Settings

A computer cluster with 1,536 cores consisting of Xeon E5-2665, E5530, and E5620, all with the clock speed of 2.40 GHz was used for experiments. Each process ran on one core. When we performed experiments using up to 512 cores, only computing nodes with E5-2665 were used. The computing nodes were interconnected through a 10-Gbps Ethernet. Every worker process of parallel versions has a table with 31,250,000 entries. The master uses a fully associative table that keeps the full information on already finished search results. These settings were used for both synthetic and shogi trees.

The sequential program for performance comparison uses a transposition table with 1,000,000,000 entries for synthetic trees. For shogi trees, the sequential program is the baseline method described in Section 5.2. It uses two transposition tables and each table has 1,000,000,000 entries.

We generated 32 synthetic trees by changing random number seeds. For shogi trees, we used 32 shogi middle-game positions that were randomly taken from game records of professionals. Each game tree was searched only once.

We compared three configurations: with updates, without updates, and with share. The first method performs dynamic updates on prediction of minimal trees while the second does not. When the dynamic updates are off, the function GetSeachDepth() always returns the required depth of the task. Note that workers still perform iterative deepening, but results of search with shallower depths are not reported. In these two methods, workers do not share transposition tables. In the third method, the dynamic updates are performed and transposition tables are shared among workers. Workers share only results of nodes whose depth from root nodes of tasks' subtree is less than or equal to 4 and 2 when $b = 5$ and when $b = 20$, respectively.

Performance measurement results are shown in **Table 1**, **Table 2**, and **Table 3**. Search time figures are the geometric means of search time periods for 32 different trees. The speedup is the sequential search time divided by the parallel search time. The average numbers of leaf nodes visited and the average idle time per worker are also shown. The idle time is divided into start-up idle time and other idle time. The start-up idle time is the average idle time before workers receives their first task. The rest is idle time after finishing a task and before receiving a new task or the termination of the whole search.

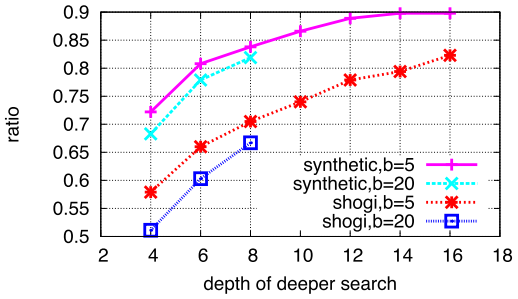


Fig. 6 Ratios of the number of nodes judged to be the best with shallower search to that with the following deeper search.

4.4 Results for Synthetic Trees

Table 1 shows the results with synthetic game trees. We can find that the dynamic updates tend to improve the performance, but the improvement is not so significant. This can be explained by accurate ordering made for synthetic trees. A minimal tree can be predicted with high accuracy using only the game state without search. In this case, results of shallow search are not so useful and the impact of the dynamic updates is small. On the other hand, since programs can mainly search those nodes in the minimal tree, the speedup is high compared to that reported in

Table 1 Results using synthetic trees.

		$b = 5$, $d = 24$, and $d' = 12$					$b = 20$, $d = 12$, and $d' = 6$				
# of proc.	method	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle time [sec.]	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle time [sec.]
1	sequential	19,438.92	1	790			7,321.58	1	149		
512	w/ updates	56.71	343	1,162	0.64	0.43	20.85	351	208	0.17	0.57
	w/o updates	50.43	385	1,008	0.65	0.56	22.88	320	228	0.18	0.59
1,536	w/ updates	25.91	750	1,381	0.88	0.68	11.21	653	289	0.47	0.54
	w/o updates	28.55	681	1,401	0.89	2.13	13.31	550	323	0.47	1.09

Table 2 Results using shogi trees when $b = 5$ and $d = 24$.

# of proc.	d'	method	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle time [sec.]
1		sequential	30,767.09	1	165		
64	10	w/ updates	1,297.36	23.72	398	0.40	23.18
		w/ share	1,257.79	24.46	384	0.42	18.05
	12	w/ updates	1,155.40	26.63	362	1.90	4.98
		w/ share	1,138.39	27.03	353	1.99	4.74
128	10	w/ updates	708.36	43.43	454	0.45	25.91
	12	w/ updates	592.00	51.97	405	1.81	7.00
256	10	w/ updates	435.48	70.65	557	0.48	33.38
	12	w/ updates	334.97	91.85	468	2.02	11.01
512	10	w/ updates	268.82	114.45	652	0.57	39.30
	12	w/ updates	209.13	147.12	584	2.06	12.43
		w/o updates	443.19	69.42	1,080	2.07	68.04
1,024	10	w/ updates	205.59	149.65	763	0.79	64.92
	12	w/ updates	149.01	206.47	740	2.27	16.33
1,536	10	w/ updates	184.31	166.93	820	0.92	81.49
	12	w/ updates	122.23	251.72	847	2.51	18.48
		w/o updates	312.41	98.48	1509	2.52	121.35
		w/ share	119.03	258.49	829	2.51	17.91

Table 3 Results using shogi trees when $b = 20$.

			$d = 12$					$d = 14$				
# of proc.	d'	method	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle time [sec.]	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle time [sec.]
1		sequential	4,020.24	1	41			48,773.45	1	457		
64	4	w/ updates	186.58	21.55	113	0.02	18.78	2,180.14	22.37	1,227	0.02	220.01
		w/ share	174.51	23.04	106	0.02	15.18	2,086.01	23.38	1,171	0.02	203.32
	6	w/ updates	146.72	27.40	83	0.18	1.28	1,781.06	27.38	959	0.18	14.89
		w/ share	140.97	28.52	79	0.18	1.04	1,718.05	28.39	913	0.19	15.43
128	4	w/ updates	122.17	32.91	144	0.03	22.65	1,433.14	34.03	1,530	0.03	298.46
	6	w/ updates	75.97	52.92	96	0.20	1.53	902.72	54.03	1,032	0.20	21.41
256	4	w/ updates	92.84	43.30	174	0.05	34.13	1,102.05	44.26	1,871	0.05	431.26
	6	w/ updates	44.15	91.06	117	0.22	1.87	522.71	93.31	1,240	0.23	25.91
512	4	w/ updates	77.64	51.78	187	0.12	46.06	937.99	52.00	2,070	0.12	566.95
	6	w/ updates	29.50	136.26	157	0.28	2.74	317.55	153.59	1,512	0.29	32.79
		w/o updates	44.88	89.57	224	0.29	7.38	643.98	75.74	2,858	0.29	130.03
1,024	4	w/ updates	82.12	48.96	188	0.42	64.70	975.66	49.99	2,013	0.55	779.26
	6	w/ updates	22.91	175.45	211	0.43	4.19	239.48	203.67	1,976	0.46	53.12
1,536	4	w/ updates	82.84	48.53	183	2.11	69.37	972.22	50.17	1,960	5.56	835.18
	6	w/ updates	20.26	198.46	254	0.63	4.77	203.65	239.50	2,229	0.66	60.64
		w/o updates	34.41	116.85	333	0.63	13.99	532.31	91.63	3,980	0.62	276.48
		w/ share	20.71	194.12	254	0.66	5.11	195.48	249.50	2,137	0.69	59.74

previous work (e.g., the speedup of YBWC for chess was 344 using 1,024 processes [4]).

4.5 Results for Shogi Trees

Table 2 shows the results for shogi trees when $b = 5$ and $d = 24$. When $b = 20$, Table 3 shows the results with the depth of 12 and 14. We changed d' to study the relationship between the number of processes and the task granularity.

First, we focus on the impact of the dynamic updates on the performance. **Figure 7** shows the improvements in the performance. The x-axis represents the search time with the dynamic updates divided by that without them. The y-axis represents the ratio of the number of leaf nodes visited. The numbers in parentheses are b, d, and the number of processes. We can find that the dynamic updates reduce both the search time and the number of leaves. The improvements by dynamic updates are larger when the problem size d is larger. The search time is roughly halved when $b = 5$ and $d = 24$, and when $b = 20$ and $d = 14$,

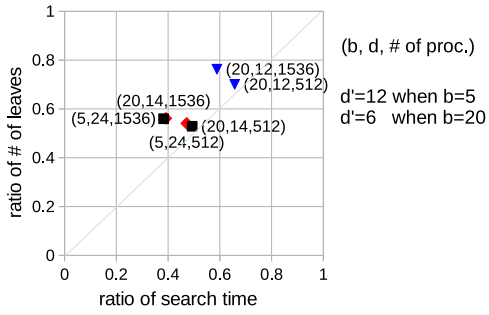


Fig. 7 Ratio of the search time and the number of leaf nodes between the methods with and without the dynamic updates.

which is more significant than for synthetic trees. This is because shogi trees cannot be ordered so accurately as synthetic trees by the initial prediction. In this case, dynamic updates improve the performance considerably.

Next, we compare the methods with and without sharing transposition tables between workers. **Table 4** shows the ratio of the

Table 4 Improvements by sharing transposition tables.

# of proc.	b	d	d'	ratio of search time
64	5	24	10	0.969
			12	0.985
	20	12	4	0.935
			6	0.961
		14	4	0.957
			6	0.965
1,536	5	24	12	0.974
	20	12	6	1.022
	20	14	6	0.960

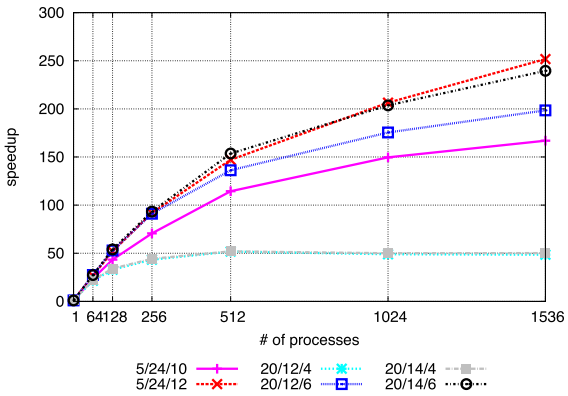


Fig. 8 Speedup of the parallel search program.

search time with sharing to that without sharing. We can observe that sharing transposition tables shortens the search time by at most 7%.

Lastly, **Fig. 8** shows the speedups of our program with dynamic updates and without sharing transposition tables. If d is large and d' is appropriately set, greater speedups can be obtained as the number of processes increases. The speedup of 250 is, however, much smaller than for synthetic trees. From Table 2 and Table 3, we can find that the main reason for the sublinear speedup is that the number of visited leaf nodes increases significantly. The idle time is also a cause, but less problematic than the increase of the number of leaves.

5. Analysis

5.1 Suspected Factors of Sublinear Speedup

The increase of the number of leaf nodes may be attributed to the following three factors: (1) the scheme for sharing transposition tables might not have worked well; (2) workers execute tasks using wider search windows than in the sequential search because most tasks start before the results of other tasks are obtained; (3) inaccurate initial prediction of a minimal tree may increase the number of tasks. These factors cannot be clearly differentiated, but we attempted to estimate the extents of the three factors. For this purpose, we have performed supplementary experiments using the sequential program.

5.2 Supplementary Experiments

We made a small modification to the sequential search program in order to estimate how much of the increase is due to imperfect sharing of transposition tables. Two separate transposition tables are used: one for the results of nodes whose depths from the root are less than or equal to d', simulating the master's table; the other for the results of nodes whose depths are greater than d', simulating a transposition table fully shared among workers. To simulate the behavior of the parallel program without table sharing, we restricted the usage of the second transposition table so that search results obtained from tasks for different positions will not be used. This simulation assumes that tasks for different positions are executed by different workers. In addition, to simulate the partial sharing scheme described in Section 3.2, we allowed the sequential program to use results obtained from other tasks if the search depths for the results are large. We call this method *partial share*.

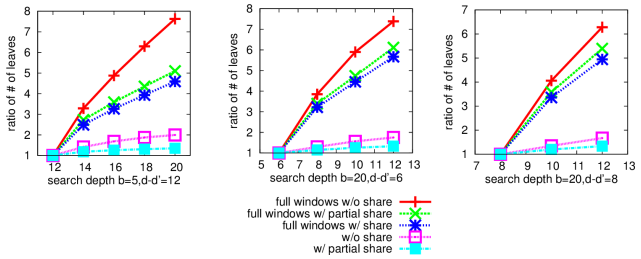


Fig. 9 Increase of the number of leaves caused by not sharing transposition tables and by using full search windows.

We made another small modification to determine the extent of the increase caused by using wider search windows. The sequential program uses full search windows [*2] for search in subtrees with the depth of $d - d'$. That is, alpha and beta values are set to −INF and INF before search in a subtree with the depth of $d - d'$. The original window is restored after the search in the subtree. This method simulates workers which execute all tasks using full search windows. This is the estimation for the worst case because actually there are cases where the search windows are narrowed.

We counted the number of leaf nodes visited by each simulation method. The baseline method simulates the master's transposition table and another table fully shared among workers. The search windows are not changed in this method. We show results of the simulation for shogi trees in **Fig. 9**. We fixed $d - d'$ and changed d step by step. The x-axis denotes d and the y-axis represents the ratio of the number of leaves in each method to that in the baseline.

5.3 Findings

We find that the program without sharing transposition tables searches approximately twice as many nodes as the one with sharing when $b = 5$. When $b = 20$, the difference becomes smaller. The difference becomes smaller when full search windows are used. Sharing transposition tables partially can actually decrease the number of leaves. On the other hand, the improvement by sharing transposition tables was at most 7% (see Table 4). The improvement using a similar sharing scheme was about 18% with 16 processes in Ref. [2]. These results show that sharing transposition tables is difficult on larger-scale computing environments.

Figure 9 also shows that the increase of the number of leaves caused by using full windows is substantial. As this is for the worst case, we counted the number of tasks finished using full

[*2] In this paper, we use the word "full search windows" or "full windows" to indicate search windows (−INF, INF).

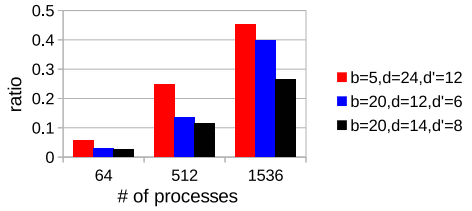
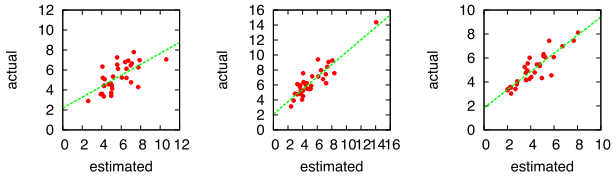


Fig. 10 Ratio of the number of tasks finished using full search windows to the total number of tasks finished.

Table 5 Increase of the number of leaves caused by each factor when the number of processes is 1,536.

b	d	d'	table	windows	tasks	estimated	actual
5	24	12	1.942	2.877	0.960	5.362	5.126
20	12	6	1.573	2.863	1.072	4.829	6.204
20	14	6	1.788	2.328	0.966	4.021	4.873



(a) $b = 5$, $d = 24$, $d' = 12$ (corr = 0.656)

(b) $b = 20$, $d = 12$, $d' = 6$ (corr = 0.896)

(c) $b = 20$, $d = 14$, $d' = 6$ (corr = 0.885)

Fig. 11 Scatter plots between the estimated and the actual increase of the number of leaves.

search windows in actual parallel search. **Figure 10** shows the ratio of this number to the total number of tasks finished when shogi trees are searched. The ratio increases with the number of processes.

We estimated the extent of each factor for the increase of the number of leaves for shogi trees. The factors described in Section 5.1 are (1) not sharing transposition tables, (2) using wider windows, and (3) increase of the number of tasks. **Table 5** shows estimated increase ratio of the number of leaves given by each factor. We also show the estimated (the product of the three ratios) and actual increase ratio. **Figure 11** shows scatter plots between the estimated and the actual increase. Each point in the figure indicates the increase for a game position. We also show correlation coefficients (corr). The increase ratio caused by not sharing transposition tables is calculated as $(1-r)t_n + rt_f$, where r is the ratio obtained from Fig. 10, and t_n and t_f are the increase ratios with normal windows and with full windows estimated from Fig. 9. The ratio by using full windows is calculated as $(1-r)+rw$, where w is the ratio estimated from Fig. 9. The increase ratio of the number of tasks is decided by comparing the numbers in the sequential and parallel program. We find that the actual ratio can be reasonably explained by these factors. One of the reasons of the difference between estimated and actual ratio is that tasks that are aborted and unfinished ones are ignored. These results also show that using full windows is the most substantial factor of the sublinear speedup.

5.4 Experiments with NegaScout

We have shown that the major bottleneck of the performance comes from the fact that many tasks are executed using full search windows. This is because we use task windows (a2,b2) for workers' search windows. Here, we discuss other methods to decide search windows in workers, comparing them to APHID's method.

The experiments reported so far were carried out without using the null-window search mechanism used in NegaScout [14], but we here discuss the behavior of the master with NegaScout and search windows used by its workers. In NegaScout, the most promising child of every node is searched using a normal search window, and then other children are searched using a null-window. A null-window is a search window $(\alpha,\alpha+1)$. If a child returns a value higher than α, the child is re-searched using a normal search window.

If the master is certain of the min-max value of the most promising child at a node on the estimated principal variation, APHID's workers use a null-window to search other children of the node. Otherwise, the workers use estimated windows around a guessed value, which is an estimated value of the root node of the master. APHID's estimated windows can be a good strategy because they are narrower than our task windows in most cases, but they may require a re-search since the estimation can be wrong. The re-search may complicate the performance analysis. This is why we use task windows instead of APHID's estimated windows or pass windows. In order to decrease the number of nodes visited, however, we have then tried using some estimated narrow windows.

The master can perform NegaScout also in our algorithm. We discuss search windows used in workers. Recall that both task windows (a2,b2) and pass windows (a1,b1) are decided by the master (see Fig. 4). If task windows are still used by workers, the difference between its behavior with and without NegaScout emerges only when an estimated minimal tree changes, and the performance difference may be small. This is because the master mostly searches only estimated minimal trees even without NegaScout. In this case, both our algorithm and APHID decide workers' search windows irrespective of whether the master performs NegaScout or not. On the other hand, if pass windows are used by workers instead of task windows in our algorithm, most of search windows used by workers are null-windows and this is different from the case where the master does not perform NegaScout.

We have modified our program to perform NegaScout in the master. In the modified program, workers decide search windows based on pass windows sent from the master. If pass windows (a1,b1) are null-windows, workers use (a1-ε,b1+ε) instead, where ε is an integer. We call these windows *ε-windows*. We note that ε-windows are more similar to APHID's estimated windows than task windows (a2,b2) are.

When ε-windows are used, we should consider when they are updated. Assume that a worker is executing a task using an ε-window (10-ε,11+ε) under a pass window (10,11). If the pass window is updated to (15,16), should the worker use a new ε-window (15-ε,16+ε)? If it uses the new window, it cannot use some results in its transposition table. This means that frequent updates of ε-windows may degrade the performance. In our modified implementation, we do not update workers' search windows when the new pass windows are within the old ε-windows. In

Table 6 Results using synthetic trees with NegaScout.

b	d	d'	# of proc.	method	ε	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle time [sec.]
5	24	12	512	w/o ε-windows		51.47	378	1,030	0.68	0.56
				w/ ε-windows	200	38.50	505	775	0.69	0.53
20	12	6		w/o ε-windows		22.97	319	228	0.18	0.56
				w/ ε-windows	200	17.34	422	188	0.17	0.62

Table 7 Results using shogi trees with NegaScout.

b	d	d'	# of proc.	method	ε	search time [sec.]	speedup	# of leaves [$\times 10^6$]	start-up idle time [sec.]	other idle [sec.]
5	24	12	512	w/o ε-windows		210.39	146.24	582	2.13	13.31
				w/ ε-windows	200	122.54	251.08	287	1.97	17.59
			1,536	w/o ε-windows		126.04	244.11	862	2.33	21.05
				w/ ε-windows	200	88.70	346.87	421	2.34	32.62
20	12	6	512	w/o ε-windows		28.72	139.98	152	0.28	2.84
				w/ ε-windows	100	21.20	189.63	113	0.29	3.02
			1,536	w/o ε-windows		20.33	197.75	255	0.65	4.75
				w/ ε-windows	100	16.51	243.50	159	0.61	6.98
20	14	6	512	w/o ε-windows		312.53	156.06	1,495	0.28	31.23
				w/ ε-windows	100	227.09	214.78	942	0.32	42.18
			1,536	w/o ε-windows		202.87	240.42	2,227	0.66	60.15
				w/ ε-windows	100	174.93	278.82	1,210	0.58	88.08

the case presented above, the worker does not update its search window when the 16 is less than or equal to 11+ε.

We have performed experiments using the modified program in which the master performs NegaScout. Workers did not share their transposition tables. The results are shown in **Table 6** and **Table 7** using synthetic trees and shogi trees, respectively. We show results both with and without using ε-windows. When ε-windows are used, we set ε to 50, 100, 200, and 500 for synthetic trees and 100, 200, and 500 for shogi trees [*3] using 512 processes. We set the best ε when experiments with 1,536 processes were performed for shogi trees. We show the best ε in the tables. By comparing the results to Table 1, Table 2, and Table 3, we can see that performing NegaScout only in the master does not lead to performance improvements. On the other hand, the number of leaves visited decreases and the performance is improved thanks to ε-windows. However, the idle time is increased by the use of ε-windows. This may be because tasks with null-windows are smaller than tasks with wider windows and load imbalance occurs.

As described above, we have conducted experiments using both NegaScout and ε-windows, but ε-windows may be useful also when the master does not perform NegaScout. In this case, however, we would have to consider a more sophisticated update policy for ε-windows, because pass windows are not null windows anymore and they can get narrowed or widened during search.

Dynamically extending the depth of subtrees of the master's tree for better load balancing is a promising way of improving the performance. APHID dynamically divides subtrees which are expected to take much time to search. Depth extension schemes, however, may degrade the performance because many obtained results in a worker's transposition table are lost unless we have an efficient scheme to share transposition tables between workers. We did not dynamically extend the search depth because the increase of the number of leaves was more problematic than load imbalance, but the extension must be future work when ε-windows are used.

[*3] The value of a pawn is approximately 100 in Gekisashi.

6. Conclusion

Parallel alpha-beta algorithms should dynamically update prediction of a minimal tree using results of shallow search to avoid searching unnecessary nodes. However, the effect of the dynamic updates has not been evaluated in detail on large-scale computing environments with more than 100 cores. We have implemented a parallel alpha-beta search algorithm and evaluated the effect of the updates. The results with game trees generated by a shogi program show that the updates are important to shorten the search time. We have also analyzed the reason why our parallel program still suffered from visiting many unnecessary nodes. Sharing transposition tables can reduce the number of leaves, but its effect is limited on large-scale environments. The increase is caused also by using wider search windows than in the sequential search because the results of other tasks cannot be obtained beforehand. We have also shown that the number of leaves visited is decreased by performing NegaScout in the master and using ε-windows in workers.

Evaluation through games between programs is also our future work. In games, game-playing programs must determine the move to play within a time limit. A game tree of the master can drastically change its form during search in our algorithm. If the best move at the root node is selected at a point in time, this may be a poor decision because the principal variation may be still not searched deeply. Iterative deepening as in APHID can be a good method when the proposed method is used in games.

Another direction of future work is introducing hierarchical masters to alleviate the overload of the master with many workers. The idea of using multiple masters, which Brockington has

already proposed, is especially suitable to multi-cluster environments if a middle-level master and its workers can be located in one cluster. In order to leverage the hierarchical masters, however, we have to discuss how to prioritize tasks in middle-level masters. Tasks in middle-level masters are different from tasks of their workers because the middle-level masters execute many passes as the top-level master. Brockington has not discussed the prioritization policy in the middle-level masters, although it is not apparent.

References

[1] Björnsson, Y. and Marsland, T.A.: Multi-cut $\alpha\beta$-pruning in game-tree search, *Theoretical Computer Science*, Vol.252, No.1-2, pp.177–196 (2001).

[2] Brockington, M.G.: Asynchronous Parallel Game-Tree Search, PhD Thesis, University of Alberta (1998).

[3] Campbell, M., Hoane Jr, A.J. and Hsu, F.-h.: Deep blue, *Artificial Intelligence*, Vol.134, No.1-2, pp.57–83 (2002).

[4] Feldmann, R.: Game Tree Search on Massively Parallel Systems, PhD Thesis, University of Paderborn (1993).

[5] Himstedt, K., Lorenz, U. and Moller, D.P.F.: A Twofold Distributed Game-Tree Search Approach Using Interconnected Clusters, *Euro-Par 2008 Parallel Processing*, Lecture Notes in Computer Science, Vol.5168, pp.587–598, Springer (2008).

[6] Hoki, K. and Muramatsu, M.: Efficiency of three forward-pruning techniques in shogi: Futility pruning, null-move pruning, and Late Move Reduction (LMR), *Entertainment Computing*, Vol.3, No.3, pp.51–57 (2012).

[7] Hyatt, R.M.: The Dynamic Tree-Splitting Parallel Search Algorithm, *ICGA Journal*, Vol.20, No.1, pp.3–19 (1997).

[8] Kaneko, T. and Tanaka, T.: Distributed Game-tree Search Based on Prediction of Best Moves, *IPSJ Journal*, Vol.53, No.11, pp.2517–2524 (2012).

[9] Kishimoto, A.: Transposition Table Driven Scheduling for Two-Player Games, Master's thesis, University of Alberta (2002).

[10] Knuth, D.E. and Moore, R.W.: An analysis of alpha-beta pruning, *Artificial Intelligence*, Vol.6, No.4, pp.293–326 (1975).

[11] Kuszmaul, B.C.: The StarTech massively parallel chess program, *ICGA Journal*, Vol.18, No.1, pp.3–19 (1995).

[12] Manohararajah, V.: Parallel Alpha-Beta Search on Shared Memory Multiprocessors, Master's thesis, University of Toronto (2001).

[13] Newborn, M.: Unsynchronized iteratively deepening parallel alpha-beta search, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.10, No.5, pp.687–694 (1988).

[14] Reinefeld, A.: An Improvement to the Scout Tree-Search Algorithm, *ICCA Journal*, Vol.6, No.4, pp.4–14 (1983).

[15] Romein, J., Plaat, A., Bal, H. and Schaeffer, J.: Transposition Table Driven Work Scheduling in Distributed Search, *16th National Conference on Artificial Intelligence* (*AAAI '99*), pp.725–731 (1999).

[16] Shoham, Y. and Toledo, S.: Parallel Randomized Best-First Minimax Search, *Artificial Intelligence*, Vol.137, No.1-2, pp.165–196 (2002).

[17] Smith, S.J.J. and Nau, D.S.: An Analysis of Forward Pruning, *12th National Conference on Artificial Intelligence* (*AAAI '94*), pp.1386–1391 (1994).

[18] Steenhuisen, J.R.: Transposition-Driven Scheduling in Parallel Two-Player State-Space Search, Master's thesis, Delft University of Technology (2005).

[19] Tsuruoka, Y., Yokoyama, D. and Chikayama, T.: Game-Tree Search Algorithm Based On Realization Probability, *ICGA Journal*, Vol.25, No.3, pp.146–153 (2002).

[20] Ura, A., Yokoyama, D. and Chikayama, T.: Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity, *Journal of Information Processing*, Vol.21, No.1, pp.17–25 (2013).

Akira Ura was born in 1985. He received his master's degree and Ph.D. degree from the University of Tokyo in 2011 and 2014, respectively. He has been engaged in FUJITSU LABORATORIES since 2014. His current research interest is machine learning and parallel computing.

Yoshimasa Tsuruoka received his Ph.D. in Engineering at the University of Tokyo in 2002, and then worked as a postdoctoral researcher at Japan Science and Technology Agency and at the University of Manchester. He then joined the faculty of Japan Advanced Institute of Science and Technology in 2009. Since 2011, he has been working as an associate professor at the University of Tokyo. His research topics include machine learning-based natural language processing and artificial intelligence for games.

Takashi Chikayama finished the Graduate School of Engineering, the University of Tokyo, and received Dr. Eng degree in 1977. He had then been engaged in the Fifth Generation Computer Systems national project until 1995. He then joined the faculty of the University of Tokyo, and had been a professor of its Graduate School of Engineering until 2014, when he started to work for UHM.