Electronic Preprint for Journal of Information Processing Vol.21 No.1

Regular Paper

Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity

Akira Ura[1,a)] Daisaku Yokoyama[2,b)] Takashi Chikayama[1,c)]

Received: February 20, 2012, Accepted: September 10, 2012

Abstract: It is difficult to fully utilize the parallelism of large-scale computing environments in alpha-beta search. The naive parallel execution of subtrees would result in much less task pruning than may have been possible in sequential execution. This may even degrade total performance. To overcome this difficulty, we propose a two-level task scheduling policy in which all tasks are classified into two priority levels based on the necessity for their results. Low priority level tasks are only executed after all high priority level tasks currently executable have started. When new high priority level tasks are generated, the execution of low priority level tasks is suspended so that high level tasks can be executed. We suggest tasks be classified into the two levels based on the Young Brothers Wait Concept, which is widely used in parallel alpha-beta search. The experimental results revealed that the scheduling policy suppresses the degradation in performance caused by executing tasks whose results are eventually found to be unnecessary. We found the new policy improved performance when task granularity was sufficiently large.

Keywords: game tree search, distributed computing, scheduling algorithm, speculative execution

1. Introduction

The alpha-beta algorithm is efficient for the sequential game tree search that is used by many computer game players and it is important to parallelize it to improve the efficiency of these players since they can be strengthened if game trees can be searched deeper. However, it is difficult to parallelize alpha-beta search because the algorithm conducts pruning to reduce the number of calculations, i.e., a task requires the results from other tasks to efficiently prune subtrees. Most processes will be idle due to the lack of concurrently executable tasks in parallel algorithms that wait for all required data to carry out the same strict pruning as that in sequential algorithms. This means that the massive number of computational resources cannot be effectively exploited. However, processes may execute many tasks that may turn out to be unnecessary for the final result in naive parallelization where the results from other tasks do not need to be waited for. This often causes degradation in performance because the computational cost of wasteful tasks usually increases exponentially.

The Young Brothers Wait Concept (YBWC) [3] is widely used in parallel alpha-beta search (e.g., Refs. [2], [5], [6], [8], [11]). However, as has been pointed out [1], YBWC has a problem in that processes frequently become idle in having to wait for results from other tasks in large-scale computing environments. Several methods that do not have to wait for all the results have been proposed to increase the number of concurrently executable tasks (e.g., Refs. [1], [3], [6], [14]). However, the necessity of tasks for the final result has not been fully considered. We have called the additional tasks *speculative tasks* and the original tasks of YBWC *mandatory tasks*. Once the execution of speculative tasks has started, the execution diverts computing resources until it terminates and this prevents new mandatory tasks from being executed.

We propose a *two-level task scheduling policy* that classifies all tasks into either high or low priority levels to overcome this problem with YBWC and its naive extensions. The low priority level tasks are executed when there are no high priority level tasks currently available. Two-level prioritization has already been used [12]. However, the execution of low level tasks is suspended in our policy so that the execution of high level tasks is not disrupted. Suspension is easy in game tree search by utilizing a transposition table, which records the state of computation for each game position. Our target environment in this paper is a distributed environment and all processes have their own transposition tables that are independent of other processes.

The contribution of this paper is three fold: (1) we explain how speculative execution effectively accomplishes parallel speed-up of YBWC, but excessively speculative execution degrades performance, (2) we propose tasks be classified into two levels based on YBWC, and (3) we explain how the two-level scheduling policy suppresses degradation and improves performance when task granularity is sufficiently large.

The remainder of this paper is organized as follows. Section 2 explains the alpha-beta algorithm and introduces parallel alpha-beta algorithms. Section 3 describes the two-level task scheduling policy for parallel alpha-beta search. Section 4 explains our implementation of parallel alpha-beta search in more detail. We

[1] Graduate School of Engineering, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[2] Institute of Industrial Science, The University of Tokyo, Meguro, Tokyo 153–8505, Japan
[a)] ura@logos.t.u-tokyo.ac.jp
[b)] yokoyama@tkl.iis.u-tokyo.ac.jp
[c)] chikayama@logos.t.u-tokyo.ac.jp

© 2013 Information Processing Society of Japan

experimentally evaluated the proposed scheduling policy using a homogeneous distributed computing environment, which is discussed in Section 5. Finally, Section 6 summarizes the paper and describes future work.

2. Alpha-beta Algorithm

The alpha-beta algorithm is widely used for game tree search. Subtrees unnecessary for the final result are pruned based on results from preceding searches. There is a pseudo-code for the alpha-beta algorithm listed in **Fig. 1**. The functions AlphaBeta() and Evaluate() return an evaluation value, which is an integer representing the advantage of a game position from the viewpoint of a player who makes a move at that position. The alpha-beta pair represents the search window. Beta is the threshold for pruning. Alpha and beta are swapped on each recursive call because children positions are evaluated from the opponent's viewpoint. The three main features of the alpha-beta algorithm are below.

- When a better result is found, the search window is narrowed.
- The narrower the search window is, the more frequently pruning occurs.
- If the best child of each node is always searched first, the fewest nodes are visited.

Before the children of each position are searched, they are sorted so that more promising children are searched earlier. Since elder children are considered more promising, the children are sorted from left to right according to their "age." This sorting can be done using many techniques such as internal iterative deepening. Figure 1 includes this internal iterative deepening. As shallower search is performed on line 4 and the children are sorted on line 5, the best child of the shallower search is the eldest child.

Certain nodes in the alpha-beta algorithm other than those below the eldest child must be searched even when all children are perfectly sorted. A tree only consisting of those nodes is called a minimal tree. There is an example of a minimal tree in **Fig. 2**. The nodes are classified into three types [7]: PV*1, CUT, and ALL. PV nodes are nodes for which exact evaluation is required. All of their children have to be searched. The best child of each PV node is classified as PV and the remaining children are considered to be CUT nodes. CUT nodes are nodes for which search can be completed by finding a child with an evaluation value higher than or equal to the upper bound of the search window. This child is classified as ALL. ALL nodes are nodes for which all children have to be searched to make sure that all of them have evaluation values lower than or equal to the lower bound of the search window. These children are classified as CUT.

The computational complexity of the alpha-beta algorithm in the best cases where the best children of all the PV and CUT nodes are searched first is $O(b^{\frac{d}{2}})$ where b is the branch factor, which is the number of children of a node, and d is the search depth [7]. Because the complexity is $O(b^d)$ without pruning, alpha-beta pruning is important for game tree search.

*1 PV stands for principal variation, which gives the sequence of the best moves for both players.

```
1  int AlphaBeta(position, depth, alpha, beta){
2    if(depth == 0 || position is a terminal node) return Evaluate(
         position);
3    //Do shallower search to decide the eldest
         child.
4    AlphaBeta(position, depth−2, alpha, beta);
5    ...; //Sort the children using the shallower
         search's result.
6    foreach(child of position){
7      alpha = max(alpha, −AlphaBeta(child, depth−1, −beta, −
           alpha));
8      if(beta <= alpha) return beta; //cutoff
9    }
10   return alpha;
11 }
```

Fig. 1 Pseudo-code for alpha-beta algorithm using internal iterative deepening.

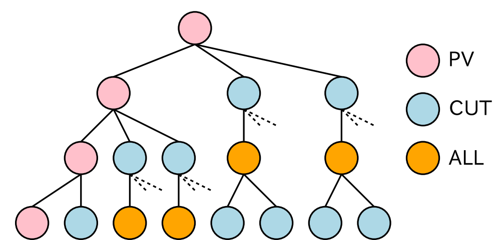

Fig. 2 Minimal tree.

2.1 Parallel Alpha-beta Algorithms

The Young Brothers Wait Concept (YBWC) [3] is a widely used method of parallel alpha-beta search (e.g., Refs. [2], [5], [6], [8], [11]). The eldest child of each node is searched first in YBWC. The remaining children are not searched until the search of the eldest child has terminated. After termination, the search window is narrowed by the result for the eldest child and the remaining children are then searched in parallel with the narrowed window. If the best child of each node is always searched first, YBWC executes exactly the same tasks as the sequential search does. YBWC executes as few unnecessary tasks as possible on actual game trees because children can be sorted very accurately so that the best child frequently becomes the eldest. That is why many parallel implementations of the alpha-beta algorithm employ YBWC.

YBWC has a drawback in that the number of tasks that can be executed in parallel is limited even if many processors are available because of data-flow synchronization [1]. Therefore, several improvements to YBWC have been proposed. All children of each ALL node in YBWC* are searched in parallel without waiting for the result for the eldest child [3]. Kishimoto and Schaeffer [6] also searched all children of the root node in parallel. Nonetheless, they only introduced some exceptions to data-flow synchronization, which did not solve the fundamental problem of the lack of executable tasks. Weill [14] suggested all the children of each node be searched. Himstedt et al. [5] argued that some computational resources should be assigned to expected moves of the opponent in advance. Moreover, some algorithms not using YBWC have been proposed (e.g., Refs. [1], [9], [12]). However, these methods have not been evaluated using more than 100 processors and can lead to degradation in performance in

© 2013 Information Processing Society of Japan

larger-scale environments due to many unnecessary tasks being executed.

3. Two-level Task Scheduling Policy

3.1 Two-level Prioritization for Game Tree Search

We present a *two-level task scheduling policy* for parallel alpha-beta search. All nodes in game trees in the policy are divided into high priority and low priority levels based on the necessity for search. Two-level prioritization was also used by Steinberg and Solomon [12], but their method was not based on YBWC. We classify nodes into the two priority levels based on YBWC because it visits a similar number of nodes as a sequential search. We propose two classification rules. The rules will be discussed later (in Section 5). Here, one of them is introduced as follows.

- The root node is a high level node.
- All children of each low level node are low level nodes.
- The eldest child of each high level node is a high level node.
- All younger children (except the eldest one) of each high level node are high level nodes if the parent is PV or ALL. If the parent is CUT, the children are low level nodes until the search of the eldest child has terminated. They are promoted to high level nodes if they are not pruned after the eldest child has terminated.

Because all children of each PV or ALL node are expected to be necessary and all younger children of each CUT node are expected to be unnecessary, we classify all younger children of CUT nodes into the low level. Nodes are estimated as PV, CUT, or ALL to apply the rule. For this, a minimal tree is estimated assuming that the eldest child of each node is the best child. In addition to the minimal-tree estimate, all younger children of each CUT node are estimated to be CUT [3].

We must also introduce some prioritization over nodes on the same priority level to determine the order for search. We employ the priority used by Steenhuisen [11] for this purpose. Nodes on the same level are searched in depth-first order.

3.2 Task Scheduling Algorithm

We regard subtrees in game trees as tasks and classify them into the priority levels of the root nodes of the subtrees. High level tasks are assigned to be processed first. After all high level tasks currently executable have been assigned, low level tasks are assigned to idle processes. Note that new high level tasks can be generated and low level tasks can be promoted to high level tasks as the search progresses. The newly-generated high level tasks and the promoted tasks are also assigned in advance of any low level tasks. When there are no idle processes, our method selects a process executing a low level task at that moment. The process suspends the low level task and begins to execute a new high level task. The execution of the suspended task is resumed after the high level task has finished.

The procedure for the two-level task scheduling is outlined using a simple example in **Fig. 3**. Assume that there are six tasks: A, B, C, D, E, and F. A and D are high level tasks and the rest are low level tasks. The result for A determines whether B and C should be executed or not. Similarly, whether E and F should be executed or not is determined after D is executed. Assume that these tasks are executed by three processes: P1, P2, and P3. First, A and D are assigned to different processes because they are high level tasks. The processes are P1 and P2 in this example. Furthermore, a low level task, B, is assigned to P3 because P3 is still idle. The left of Fig. 3 shows this initial assignment of tasks. Next, assume that D has finished execution. E and F may still be found to be unnecessary at a later stage for real parallel game tree searches, according to the results for A, B, and C. However, our method considers E and F as high level tasks and assigns them to processes. One of them is assigned to P2, which is now idle. The other is assigned to P3, which is executing a low level task. P3 suspends the execution of B and starts executing the new high level task. This scenario is shown at the right of Fig. 3. After F has finished and B is not pruned, P3 resumes the execution of B. Even if B is also promoted to a high level task during the execution of F, P3 continues F and B is still on P3. Although this may adversely influence performance, we believe that its influence is negligible.

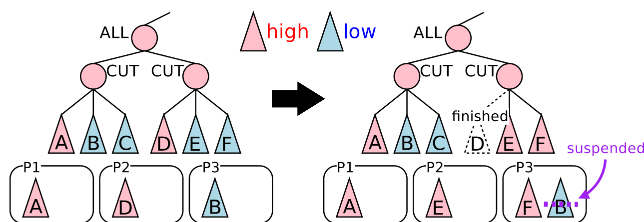

Fig. 3 Scenario in two-level task scheduling policy.

4. Implementation

4.1 Construction of Search Tree and Task Generation

We implemented a parallel game tree search program on Gekisashi[*2], which is one of the strongest shogi programs available. It was implemented with the master-worker model shown in **Fig. 4**. We denote the total search depth by d and the task granularity by g. The master searches a game tree with a depth of $d - g$ and stores the tree in memory. A leaf of the tree is regarded as a subtree with a depth of g. The subtree corresponds to a task and the master sends it to a worker. The worker searches the subtree and returns the result to the master. The master updates the tree with the result and sends another task to the worker.

The master executes a simple alpha-beta search with a transposition table and internal iterative deepening for move ordering, and uses realization probability for selective deepening [13]. Many sophisticated techniques used in strong programs were omitted from the master to simplify implementation. We use the term "search depth" instead of "realization probability" in this paper because negative logarithms of realization probabilities can be regarded as the search depths.

Internal iterative deepening does a shallower search to determine the eldest child at every node. Child nodes are searched after the shallower search in the sequential alpha-beta search. Our parallel program executes the child search concurrently with the shallower search and gives the highest priority to the shallower search. The children are searched with the depths calculated from

[*2] Gekisashi, http://www.logos.t.u-tokyo.ac.jp/~gekisashi/

© 2013 Information Processing Society of Japan

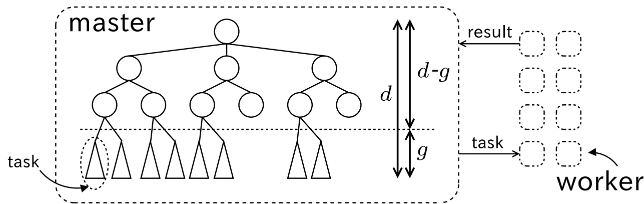

Fig. 4 Overview of our parallel game tree search program.

```
1  void TreeConstructionLoop(){
2    while(1){
3      node = NodeQueue.dequeue();
4      if(node.position is a terminal node){
5        ResultQueue.enqueue(Result(node.parent, Evaluate(
             node.position)));
6      }
7      else if(node.depth <= granularity){
8        ...; //Send the task to a worker.
9      }
10     else{
11       //Do shallower search to decide the eldest
           child.
12       NodeQueue.enqueue(Node(node.position, node.depth
           −2, node.alpha, node.beta));
13       foreach(child of node.position){
14         //Cost(child) returns a real number more
             than 1.
15         NodeQueue.enqueue(Node(child, node.depth−Cost(
             child), −node.beta, −node.alpha));
16       }
17     }
18   }
19 }
```

Fig. 5 Pseudo-code to construct master's search tree.

```
1  void ProcessResultsLoop(){
2    while(1){
3      result = ResultQueue.dequeue();
4      ProcessResult(result);
5    }
6  }
7
8  //Update the information of result.node using
       result.value.
9  void ProcessResult(result){
10   node = result.node;
11   value = result.value;
12   if(result is received from the shallower search){
13     ...; //Abort the task corresponding to the
           eldest child.
14     //The eldest child is re-searched deeper.
15     NodeQueue.enqueue(Node(eldestChild, node.depth−1, −
           node.beta, −node.alpha));
16   }
17   else{ //The result is received from a child.
18     if(node.alpha < −value){
19       node.alpha = −value;
20       ...; //Update the windows of all the
             descendants.
21       if(node.beta <= node.alpha){ //cutoff
22         ...; //Abort all the children.
23         ProcessResult(Result(node.parent, node.beta));
24       }
25     }
26   }
27   if(all chidlren have finished){
28     ProcessResult(Result(node.parent, node.alpha));
29   }
30 }
```

Fig. 6 Pseudo-code to process received results.

the realization probabilities. After the shallower search, only the eldest child is re-searched deeper than when using the realization probability. However, the younger children are not re-searched deeper even if the results update alpha in our current implementation. The best child should be searched deeper than when using the realization probability to confirm that it is truly the best. This re-search will be part of our future work. We have shown the pseudo-code to construct the search tree in **Fig. 5** and the pseudo-code to process received results in **Fig. 6**. NodeQueue is a priority queue in the code, which contains frontier nodes of the master's search tree. Function Node() generates a node corresponding to AlphaBeta() in Fig. 1. The search depth using the realization probability is calculated using function Cost() on line 15.

When the master receives the results of tasks, it enqueues the results into ResultQueue. Each result includes the node and the value. The master updates the information for the node using the value shown in Fig. 6. The update includes pruning subtrees that have turned out to be unnecessary, narrowing the search windows of nodes, and promoting low level nodes. When the master needs to update such information for tasks that have already been sent to workers, it sends messages to the workers to update the information. The master for these operations remembers the workers that it sent the tasks to.

4.2 Master Process

The master consists of seven threads for the following tasks.

- Expanding nodes in the order of priority and generating tasks
- Sending tasks to workers
- Receiving results of tasks from workers
- Updating the search tree using the results of tasks
- Sending messages to workers for aborting pruned tasks
- Sending messages to workers for updating search windows
- Sending messages to workers for task promotion

The master expands[*3] nodes step by step in best-first order. However, expanding all nodes down to the depth of $d-g$ is inefficient because the tree can become too large. Handling a huge tree is time-consuming and storing it in memory may be impossible. For this reason, low level nodes are not expanded first. When the master has expanded all the high level nodes currently available and has no tasks that can be sent, low level nodes are expanded. If low level nodes are promoted to high level nodes, they are also expanded before any low level nodes are. Tasks are sent to workers selected in a round-robin fashion. Note that high level tasks are not only sent to idle workers but also to workers executing low level tasks.

4.3 Worker Process

Each worker receives a task from the master and executes it by calling the search function of Gekisashi. After a worker has sent the result of the task to the master, the worker receives the next task. The worker has its own transposition table indepen-

[*3] Expanding a node means appending the children of the node to the tree.

© 2013 Information Processing Society of Japan

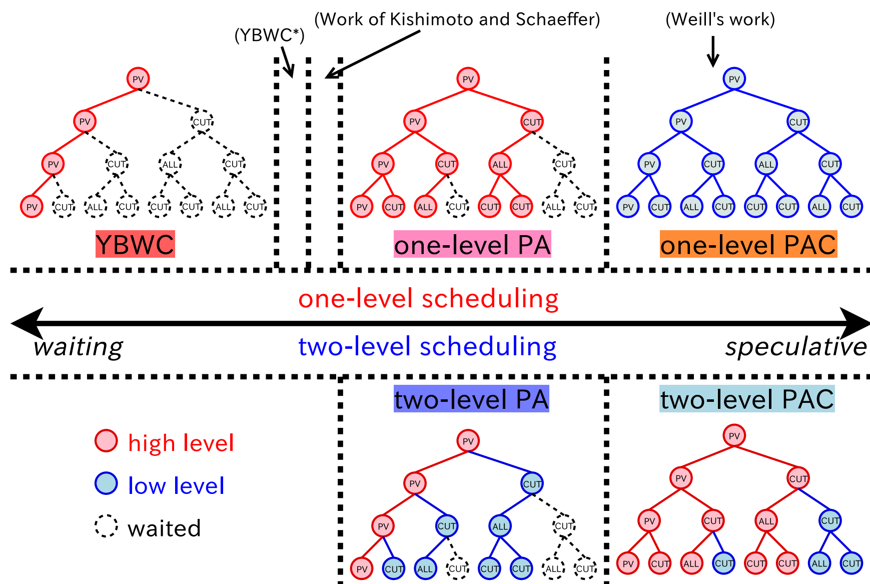

Fig. 7 Classification of scheduling policies.

dent of other workers and does not clear its transposition table until the master has completed the search. When the worker receives a high level task during the execution of a low level task, it suspends the low level task and executes the high level task. The suspended task remains with the worker and execution is resumed after the high level task has finished. Workers simply abort the execution of a low level task in the actual implementation and restart it from the top to resume it. As the transposition table is retained during suspension, the degradation in performance is minor while implementation is easy. Each worker only communicates with the master and there is no communication between workers. Apart from executing tasks, workers abort their tasks and update information for their tasks when they receive messages from the master.

5. Experimental Evaluation

We evaluated the two-level task scheduling policy by measuring its execution time and strength as a shogi program. We will first explain the execution time and then the strength as a shogi program.

All experiments were carried out on a cluster where each node had two quad core Xeon E5530 processors at 2.40 GHz with 24 GB of memory. The nodes were interconnected through a 10-Gbps Ethernet and ran 64-bit Linux. Workers for a node ran on each of eight cores. The master ran on another node on which no workers were executed. Our implementation used basic TCP/IP sockets for communication between the master and workers.

We compared the five scheduling policies shown in **Fig. 7**. These scheduling policies are separated into two large groups, i.e., one-level and two-level scheduling. In contrast to two-level scheduling policies, one-level scheduling policies classify all nodes into the same priority level. In addition to the number of levels, it is also important to consider which of the speculative tasks are executed. From this standpoint, three one-level scheduling policies are presented in the upper row of Fig. 7. YBWC is a one-level scheduling policy which waits for the eldest child of each node to be terminated. One-level PAC (one-level PV, ALL, and CUT) searches all children of each node in parallel. All nodes in one-level PAC are regarded as low level nodes in the sense that nodes are only expanded when there is a shortage of tasks. One-level PA (one-level PV and ALL) waits for the result of the eldest child of each CUT node because all younger children of CUT nodes are likely to be unnecessary due to pruning. One-level PA can be positioned between YBWC and one-level PAC from the viewpoint of speculative execution. Additionally, Fig. 7 shows where YBWC* [3], the work of Kishimoto and Schaeffer [6], and Weill's work [14] are positioned. Next, let us consider changing one-level PA and one-level PAC into two-level policies. To achieve this, we need to determine how to classify tasks into two priority levels. Two-level PAC, which is one of our proposals described in Section 3, classifies all younger children of each CUT node into the low level until the eldest child terminates. All younger children of each PV or ALL node in two-level PA are classified into the low level until the eldest child terminates. We not only propose two-level PAC but also two-level PA. The lower row in Fig. 7 outlines the two-level scheduling policies.

5.1 Evaluation by Measuring Execution Time

We prepared the six game positions shown in **Fig. 8** to measure the execution time. We made Gekisashi play games against itself with a search depth of 14 and extracted the six positions from six different game records. A and D are positions after 30 moves. B and E are positions after 60 moves. C and F are positions after 90 moves. D, E, and F can be considered to be large tasks, as they (especially E) take much longer to search than the others with a sequential program. The difference in the search times arises largely from whether the children of the root node are correctly sorted or not. The sequential program does the same search as the parallel program using one worker. It carries out a depth-first alpha-beta search with a depth of $d - g$ and calls Gekisashi's search function with a depth of g at leaf nodes. We did not use the original Gekisashi as the sequential program because it performs deeper re-search for younger children while our parallel program does not. We wanted to compare the parallel and se-

© 2013 Information Processing Society of Japan

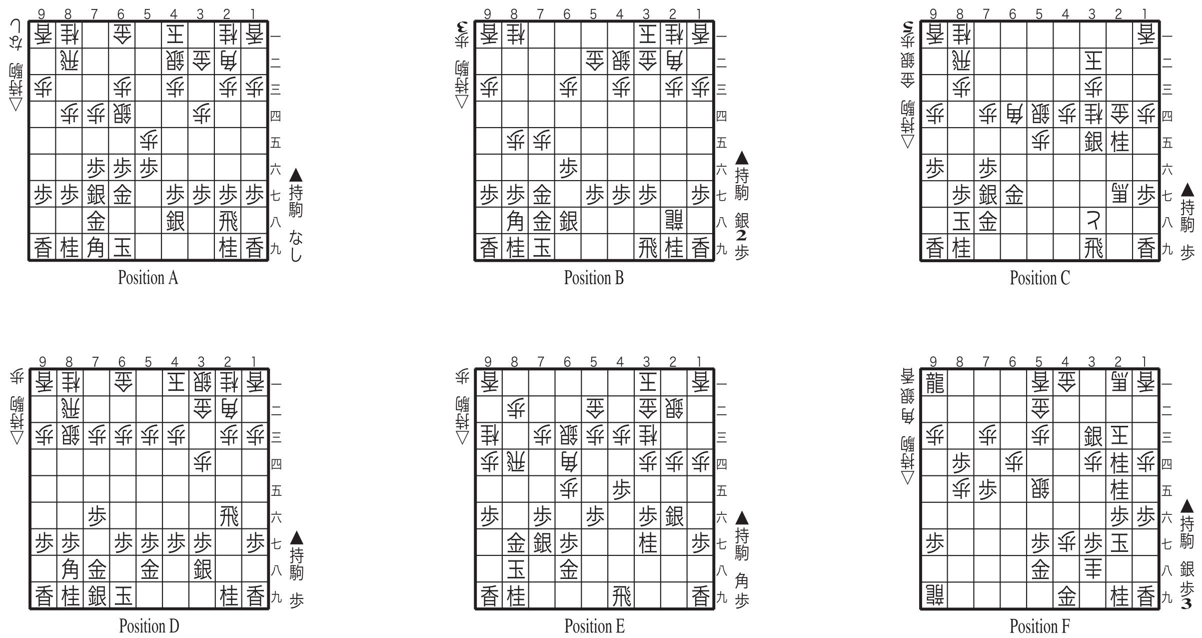

Fig. 8 Six shogi positions for evaluation.

Table 1 Execution time for sequential program.

	$g = 9$	$g = 10$	$g = 11$	$g = 12$	$g = 13$	$g = 14$	$g = 15$
$d = 18$	124s	108s	87.7s	83.9s	82.8s		
$d = 22$			2,800s	2,460s	2,410s	2,160s	2,480s

quential programs under the same conditions. We conducted experiments varying the search depth, task granularity, and number of workers. Search depth d was set to 18 and 22. Task granularity varied from 9 to 13 with the search depth of 18 and from 11 to 15 with the search depth of 22 so that the execution time for the parallel program was reasonably short. The number of workers, denoted by N, varied from 16 to 512. The search times for the sequential program are summarized in **Table 1**. The values are geometric means of the six positions. The execution time varies with granularity because the search trees varied with granularity due to the differences in the search methods of the master and the workers.

The execution time using 512 workers with the search depth of 18 is shown in **Fig. 9** and that with the search depth of 22 is shown in **Fig. 10**. Five trials of a parallel search in each position were conducted. We computed arithmetic average values and standard deviations of the execution time between the trials, and then geometric mean values of those average values and the standard deviations between the six shogi positions. That is, we calculated the averaged execution time μ and the standard deviation σ for each method as

$$\mu_i = \frac{1}{5}\sum_{j=1}^{5} t_{ij} \tag{1}$$

$$\sigma_i^2 = \left(\frac{1}{5}\sum_{j=1}^{5} t_{ij}^2\right) - \mu_i^2 \tag{2}$$

$$\mu = \left(\prod_{i=1}^{6} \mu_i\right)^{\frac{1}{6}} \tag{3}$$

$$\sigma = \left(\prod_{i=1}^{6} \sigma_i\right)^{\frac{1}{6}}, \tag{4}$$

where t_{ij} is the execution time of the j-th trial for the i-th position. The deviations are indicated by the error bars. Some points have not been plotted because the program was unable to run due to memory limitations.

First, we noted the difference between the three one-level scheduling policies to investigate the impact of speculative execution without two-level task scheduling. One-level PA performed the best out of the three policies. We discovered that more speculative execution than YBWC is appropriate and that excessive speculative execution degrades performance.

Second, we focused attention on the execution time for the two two-level scheduling policies. While two-level PA performed similarly to one-level PA, two-level PAC outperformed one-level PAC. These results demonstrated that two-level scheduling avoids the degradation in performance caused by executing many meaningless tasks. Degradation does not occur in one-level PA. Moreover, two-level PAC is the best at large granularity. However, it is worse than one-level PA and two-level PA at small granularity.

Third, we will discuss CPU utilization by workers. CPU utilization is defined as the time spent on executing tasks divided by the total execution time including idle time. We computed arithmetic average values for CPU utilization over the six shogi positions. **Figure 11** and **Fig. 12** show the averaged CPU utilization of 512 workers at large granularity for the former and at small granularity for the latter. YBWC has the lowest CPU utilization of the policies. We can say that speculative execution shortens the execution time because the idle time of workers shortens. However, the difference in the execution time at large granularity between two-level PA (or one-level PA) and two-level PAC is small despite the large difference in CPU utilization. This is because most of the low level tasks in two-level PAC are actually unnecessary. Although the number of concurrently executable tasks increases at small granularity, the difference in the CPU utilization between policies is smaller than at large granularity. This phe-

© 2013 Information Processing Society of Japan

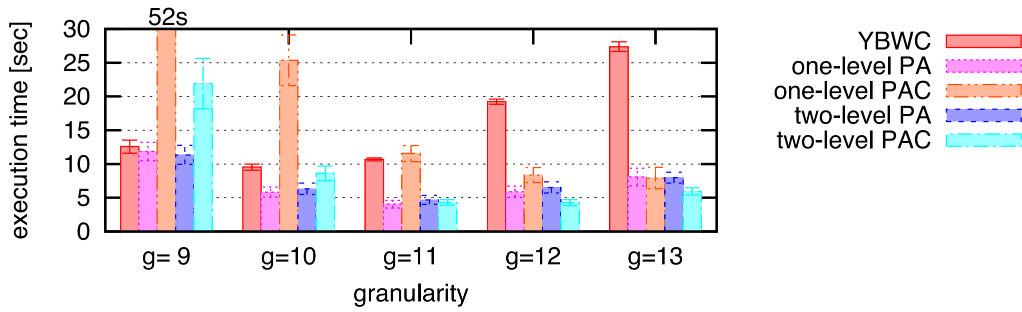

Fig. 9 Execution time in the scheduling policies (d = 18, N = 512).

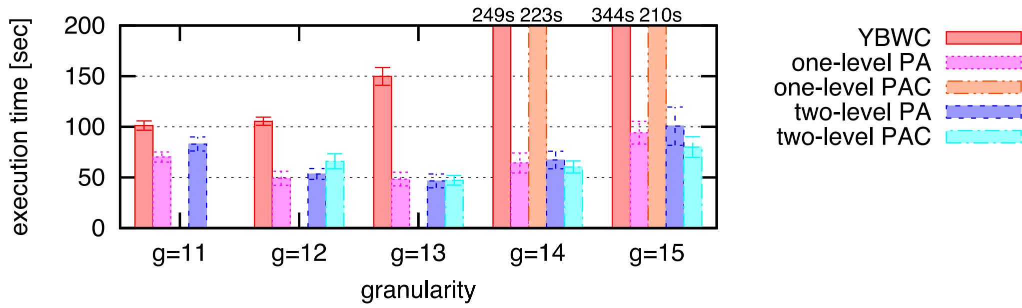

Fig. 10 Execution time in the scheduling policies (d = 22, N = 512).

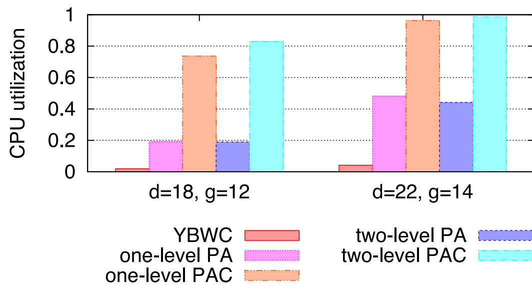

Fig. 11 CPU utilization of workers at large granularity (N = 512).

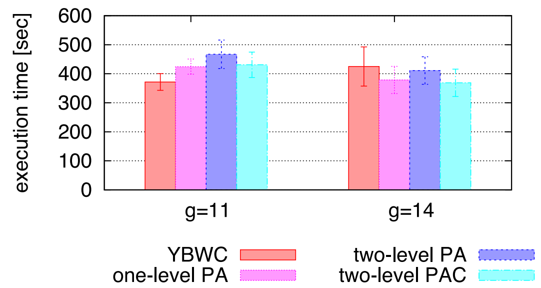

Fig. 13 Execution time using 16 workers (d = 22).

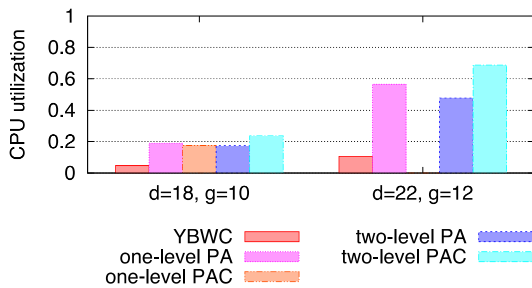

Fig. 12 CPU utilization of workers at small granularity (N = 512).

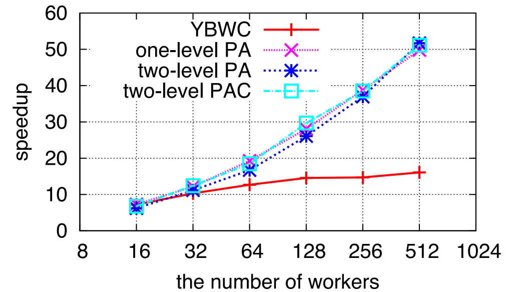

Fig. 14 Speedups using from 16 to 512 workers (d = 22, g = 13).

nomenon is caused by a bottleneck in the master process. While the master takes a constant time to handle a task regardless of task granularity, workers only need a few moments to execute a task with small granularity. As a result, the master cannot deal with many workers. This explains why two-level PAC is worse than two-level PA at small granularity. Even though the difference in CPU utilization between two-level PAC and two-level PA is small, two-level PAC needs more time to maintain larger game trees than two-level PA.

Fourth, we measured the execution time using 16 workers. We focused on situations in this research where massive computational resources were available. It was, however, also important to verify our results in situations where only tens of processes were available. The execution time with the search depth of 22 is shown in **Fig. 13**. When the search depth was 18, we observed the same tendency as that in Fig. 13. The relative difference between the policies is smaller than that using 512 workers. This is because the CPU utilization of workers is high even in YBWC and therefore the difference in CPU utilization between the policies is small.

Last, **Fig. 14** plots speedups compared to the sequential program used for Table 1. A speedup is defined as the execution time for the sequential program divided by that for the parallel program. This result was acquired with a search depth of 22 at a granularity of 13. The performance of YBWC saturates when N is greater than 32 and does not seem to improve if the number

© 2013 Information Processing Society of Japan

of workers increases. This is mostly because the CPU utilization of YBWC diminishes more rapidly with the increase in N than that of other methods when N is larger than 32. One-level PA, two-level PA, and two-level PAC are expected to lead to further improvements in performance if more workers are available. However, we did not observe large difference in performance between the policies.

5.2 Evaluation by Measuring Strength as a Shogi Program

We measured the execution time for only six positions in the preceding experiments. This was insufficient to evaluate the performance of a game tree search. Therefore, we also examined the strength of our parallel program as a shogi program. We measured the rate of winning our parallel program had to the original multithreaded Gekisashi that consisted of eight threads. We prepared 50 distinct game positions by randomly choosing the first 20 moves from Gekisashi's opening book. We performed two games from every position. Our parallel program was black in one of the two games and was white in the other. The rate of winning was calculated from 100 games in which each player had 10 seconds of thinking time per move. Fixing granularity was not a good strategy to measure strength in these experiments because of the time limit for making moves. For this reason, we gradually enlarged granularity as the search progressed. We chose granularity based on the root's search depth because our program performs iterative deepening and searches the root concurrently with shallower searches using different depths. We examined the three patterns of increasing granularity shown in **Table 2**, small, medium, and large. The search depth was set to 24, where our program did not finish searches in 10 seconds.

The rates of winning using 128 workers and using 512 workers are listed in **Table 3** for the former and in **Table 4** for the latter. The experiment using 512 workers was only carried out with parameter setups that exhibited a high rate of winning with 128 workers. We can see from Table 3 that YBWC is weaker than the others with the medium and large granularity patterns. However, two-level PAC is weaker than YBWC with the small granularity pattern. The reason for these observations is the same as that described in the evaluation of execution time.

Table 2 Three patterns of increasing granularity.

the root's search depth	≤14	16	18	20	22	24
granularity of the small pattern	9	9	10	11	12	13
granularity of the medium pattern	9	10	11	12	13	14
granularity of the large pattern	10	11	12	13	14	15

Table 3 Rate of winning our parallel program consisting of 128 workers had to multithreaded Gekisashi consisting of eight threads.

	small	medium	large
YBWC	41.0%	53.1%	40.4%
one-level PA	60.8%	68.8%	61.1%
two-level PA	53.1%	69.8%	61.6%
two-level PAC	40.2%	60.8%	69.5%

Table 4 Rate of winning our parallel program consisting of 512 workers had to multithreaded Gekisashi consisting of eight threads.

	medium	large
two-level PA	76.3%	
two-level PAC		53.7%

Two-level PAC using 512 workers is weaker than that using 128 workers. The reason for this can be explained as follows. Shallower search of the root node is executed concurrently while its children are searched. Deeper search is wasted unless shallower search finishes within the time limit. When many workers are available, deeper search is started concurrently with shallower search and the high level tasks of deeper search may prevent the low level tasks of shallower search from being executed. This should be solved through introducing more sophisticated priority control.

6. Conclusion

We proposed a two-level task scheduling policy for parallel alpha-beta search to avoid both depletion of tasks and degradation of performance caused by the execution of many unnecessary tasks. The policy only assigns low level tasks to processes when there are no unassigned high level tasks remaining. Tasks expected to be pruned are classified as low level. We explained that the policy suppresses the degradation in performance resulting from the execution of many unnecessary tasks. Moreover, the policy improves performance at large granularity. We believe speculative execution becomes important to achieve high levels of performance in large-scale distributed computing environments.

However, our experiments revealed that the improvements in performance were not as high as expected given that CPU utilization became quite high at large granularity through the execution of speculative tasks. This is clearly because most of the speculative tasks were unnecessary. Speculative tasks that are more likely to be necessary for the final result must be selected to achieve further improvements in performance. Therefore, more than two levels should be introduced for task scheduling. For example, it should be considered to estimate the probabilities that tasks are necessary for the final result and to utilize the probabilities as task priorities. If more than two levels are introduced, however, we must more carefully consider the necessity for suspending tasks. For example, we must assess whether medium level tasks should be suspended to execute high level tasks or not when there are three levels (high, medium, and low).

Furthermore, we should parallelize the master process to prevent it from becoming a bottleneck. Our experiments indicated that only one master could deal with up to tens of workers when granularity was rather small. We plan to use work-stealing [4] or Transposition-table Driven work Scheduling (TDS) [10], in which all processes run as masters and as workers. Idle processes steal tasks from other processes in work-stealing. Each task in TDS is assigned to a certain process based on the hash value for the task [6], [11]. In either case, communication between masters is needed to abort tasks and to update information for tasks when we employ more than one master. Because frequent communication can degrade performance, we have to find an appropriate communication policy.

Acknowledgments This research was partially supported by a project for "Research and development on cloud service infrastructure for recovering wide-area disaster (Reliable cloud service platform technology)" of the Ministry of Internal Affairs and Communications.

© 2013 Information Processing Society of Japan

References

[1] Brockington, M.G.: Asynchronous Parallel Game-Tree Search, PhD Thesis, University of Alberta (1998).
[2] Campbell, M., Joseph Hoane Jr., A. and Hsu, F.: Deep blue, *Artificial Intelligence*, Vol.134, No.1-2, pp.57–83 (2002).
[3] Feldmann, R.: Game Tree Search on Massively Parallel Systems, PhD Thesis, University of Paderborn (1993).
[4] Halstead, R.: Implementation of Multilisp: Lisp on a Multiprocessor, *Proc. 1984 ACM Symp. on LISP and Functional Programming*, pp.9–17 (1984).
[5] Himstedt, K., Lorenz, U. and Möller, D.P.F.: A Twofold Distributed Game-Tree Search Approach Using Interconnected Clusters, *Proc. Euro-Par '08*, pp.587–598 (2008).
[6] Kishimoto, A. and Schaeffer, J.: Distributed game-tree search using transposition table driven work scheduling, *Proc. ICPP '02*, pp.323–330 (2002).
[7] Knuth, D. and Moore, R.: An Analysis of Alpha-Beta Pruning, *Artificial Intelligence*, Vol.6, No.4, pp.293–326 (1975).
[8] Kuszmaul, B.C.: The StarTech Massively Parallel Chess Program, *International Computer Chess Association Journal*, Vol.18, No.1, pp.3–19 (1995).
[9] Newborn, M.: Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search, *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol.10, No.5, pp.687–694 (1988).
[10] Romein, J.W., Plaat, A., Bal, H.E. and Schaeffer, J.: Transposition Table Driven Work Scheduling in Distributed Search, *Proc. AAAI '99*, pp.725–731 (1999).
[11] Steenhuisen, J.R.: Transposition-Driven Scheduling in Parallel Two-Player State-Space Search, Master's thesis, Delft University of Technology (2005).
[12] Steinberg, I. and Solomon, M.H.: Searching Game Trees in Parallel, *Proc. ICPP '90*, Vol.3, pp.9–17 (1990).
[13] Tsuruoka, Y., Yokoyama, D. and Chikayama, T.: Game-Tree Search Algorithm Based on Realization Probability, *International Computer Games Association Journal*, Vol.25, No.3, pp.146–153 (2002).
[14] Weill, J.-C.: The ABDADA distributed minimax search algorithm, *Proc. CSC '96*, pp.131–138 (1996).

Akira Ura was born in 1985. He received his master's degree from the Graduate School of Engineering, the University of Tokyo in 2011. Currently, he is a Ph.D. candidate at the Graduate School of Engineering, the University of Tokyo.

Daisaku Yokoyama was born in 1974. He received his Ph.D. degree from the Graduate School of Frontier Sciences, the University of Tokyo in 2006. He is now a research associate at the Institute of Industrial Science, the University of Tokyo. His major research interests include distributed programming framework, search problem, and game solver.

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 Unviersity of Tokyo, and is currently a professor of the Graduate School of Engineering.

© 2013 Information Processing Society of Japan