Parallel AND/OR tree search based on proof and disproof numbers

Akihiro Kishimoto and Yoshiyuki Kotani
Tokyo University of Agriculture and Technology
{kishi, kotani}@fairy.ei.tuat.ac.jp

Abstract

Proof and disproof numbers are very important ideas to reduce the search space of AND/OR trees. This paper suggests a method for asynchronously parallelizing AND/OR tree search using proof and disproof numbers. The experimental results in Othello show the rate of speed-up was 3.60 on a distributed machine with 16 processors.

1 Introduction

1.1 Background

Game tree search is one of the representative topics in AI, and AND/OR trees can be seen as a process of solving two agent games. In a recent paper, Allis introduced proof and disproof numbers as a method for proving or disproving game trees[1]. Allis' proof-number search is a best-first algorithm, which has the disadvantage of using a large amount of memory. Nagai's PDS overcomes this defect by the use of depth-first search[8].

Parallelization is one method for improving the running time of a search algorithm. Generally speaking, to achieve a sufficient speed-up, we must mainly cope with the following three kinds of overhead: Search overhead results from the extra work that a parallel algorithm does, compared with its sequential counterpart. Communication overhead is the communication latency that occurs when processors exchange information. Synchronization overhead is the idle time at synchronization points where some processors have to wait for the others to finish their search. These overheads depend on one another. For example, if we reduce the communication overhead and the synchronization overhead, the search overhead will be increased, and on the other hand, reducing search overhead may increase the other two overheads.

In the field of parallel alpha-beta search algorithms, there are two major approaches: *Synchronous* and *asynchronous*. Keeping the search overhead of synchronous parallel algorithms such as PV-Split[5] and YBWC[3] low depends on strong move ordering in the same tree. On the other hand, the asynchronous approach was developed to get rid of the synchronization points of synchronous parallel algorithms. Newborn's UIDPABS[10] first tried this approach in his chess program, but the speed-up was worse than PV-Split. However, recently Brockington's asynchronous algorithm, named APHID, has shown that there is a possibility to outperform YBWC in real game trees[2].

Though many researchers have developed parallel alpha-beta search algorithms as men-

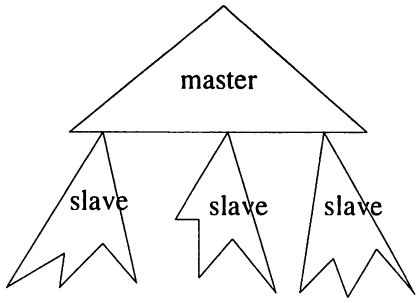

Figure 1: Organization of ParaPDS

tioned above, there is no previous work on parallelizing AND/OR tree search algorithms based on proof and disproof numbers. Regarding theoretical aspects of parallel AND/OR tree search algorithms, Karp and Zhang showed that their AND/OR tree search algorithm P-SOLVE allocates the right number of processors to the subtrees, and proved that their algorithm achieves a linear speed-up on every instance of a uniform tree[4]. Nakayama et.al. implemented a tsume-shogi solver, which was designed by combining a depth-first iterative deepening algorithm as a slave with a best-first search algorithm as a master[9]. Though this algorithm is a parallel one, its performance is inferior to Seo's sequential tsume-shogi solver[11], which uses proof numbers.

1.2 Contribution of This Paper

In this paper, we propose ParaPDS, a parallel version of Nagai's PDS. ParaPDS uses a master and a series of slave processors (Figure 1). The master processor traverses the search tree from the root node to some extent, and asks the slave processors to search the rest of the tree. The algorithm of the slaves is PDS. The slaves expand the subtrees assigned by the master in parallel, and return the results of search to the master, which backs up the proof and disproof numbers. ParaPDS achieves low communication overhead and has no synchronization overhead because the slaves search subtrees independently. Search overhead is experimentally about 300 percent with 16 processors in solving Othello endgames, using local transposition tables.

The organization of this paper is as follows: Section 2 introduces a sequential AND/OR tree search algorithm PDS. Section 3 describes ParaPDS. In Section 4 we show and discuss the experimental results in Othello on a distributed memory machine with 16 processors. We make some concluding remarks in Section 5.

2 Sequential Algorithm

We use Nagai's PDS as a sequential search algorithm, which used less memory and expanded less search nodes than Allis' proof-number search in Nagai's experiments on Othello and random game trees. In this section we give an overview of PDS, which is explained in [7] or in [8] in detail.

2.1 Proof and Disproof Numbers

The idea of proof numbers originates from McAllester's conspiracy numbers, which was introduced by measuring the reliability of min-max values[6]. Allis called conspiracy numbers for AND/OR trees *proof numbers* and used them to find which player wins in two player zero-sum game with perfect information[1]. In the case of AND/OR trees, the definition of conspiracy numbers(proof numbers) becomes the minimum number of the leaf nodes to prove the game tree. Allis also defined *disproof numbers* as the dual notion of proof numbers. While proof numbers estimate how easy it is to prove game trees, disproof numbers focus on the facility to disprove.

The proof number pn(n) and the disproof

number $\mathrm{dn}(n)$ at a node n are calculated as follows:

(1) For a leaf node n which is neither proved nor disproved, $\mathrm{pn}(n) = \mathrm{dn}(n) = 1$.

(2) For a proved node n, $\mathrm{pn}(n) = 0$ and $\mathrm{dn}(n) = \infty$.

(3) For a disproved node n, $\mathrm{pn}(n) = \infty$ and $\mathrm{dn}(n) = 0$.

(4) For an interior OR node n,

$$\mathrm{pn}(n) = \min(\mathrm{pn}(n_1), \cdots, \mathrm{pn}(n_k))$$

$$\mathrm{dn}(n) = \mathrm{dn}(n_1) + \cdots + \mathrm{dn}(n_k)$$

where $n_1, \cdots, n_k$ are the children of n.

(5) For an interior AND node n,

$$\mathrm{pn}(n) = \mathrm{pn}(n_1) + \cdots + \mathrm{pn}(n_k)$$

$$\mathrm{dn}(n) = \min(\mathrm{dn}(n_1), \cdots, \mathrm{dn}(n_k))$$

where $n_1, \cdots, n_k$ are the children of n.

2.2 Depth-First Search

Proof-number search is a best-first search, which expands a leaf node by continuously selecting a child whose (dis)proof number is minimum at OR(AND) nodes, starting from the root. Unlike proof-number search, Nagai's PDS is an iterative deepening search algorithm which uses two thresholds of the proof and the disproof number in each node. Its behavior is almost the same as proof-number search. The techniques used in PDS are the same in Seo's algorithm, which achieved the best known results in solving hard tsume-shogi problems[11].

For the simplicity of our explanation, we first explain Seo's algorithm, which has only one threshold of the proof number, before explaining PDS. Seo's algorithm begins searching with a threshold of the root node pn = 1 to prove.[1] If there is no proof, then it increments pn and tries to prove within the threshold of pn until there is a proof of the root node. Since this process re-expands many nodes again and again, Seo uses a large transposition table to reduce re-expansion. Though normal iterative deepening methods such as alpha-beta search only iterate at the root node, Seo does iterative deepening at all OR nodes, which is called *multiple iterative deepening*. Multiple iterative deepening reduces the number of search nodes dramatically though there is overhead of re-expansion. Seo mentioned in his M.Sc. thesis the ratio of re-expanded nodes is about 20 percent through the experimental results of tsume-shogi.

Like Seo's algorithm, PDS expands a node n while $\mathrm{pn}(n)$ and $\mathrm{dn}(n)$ do not both exceed each threshold, uses a large transposition table, and does multiple iterative deepening at all nodes. Once proof and disproof numbers exceed each threshold, it is necessary to increment the threshold of the proof or the disproof number. Let $n.\phi$ be a proof number at an OR node n or a disproof number at an AND node n, and $n.\delta$ be a disproof number at an OR node n or a proof number at an AND node n, the incrementing strategy of PDS is as follows:

(1) If $n.\phi \leq n.\delta$, increment $n.\phi$.

(2) Otherwise increment $n.\delta$.

This means that PDS aims to prove if search trees seem to be easy to prove, and to disprove if not. By using of both proof and disproof numbers, PDS is good not only at proofs but also at disproofs.

3 ParaPDS: Parallel Algorithm

In this section we describe a new parallel tree search algorithm, ParaPDS, using proof and

[1] In fact we can begin with a threshold of 2.

disproof numbers. ParaPDS is based on an *asynchronous* approach. There are no communication and synchronization overhead among the slaves. The structure of ParaPDS is very similar to APHID[2], in which the master is responsible for the first d' ply of an alpha-beta search while the slaves search the remaining $d - d'$ ply with the iterative deepening alpha-beta algorithm, and can be regarded as an application of APHID to AND/OR trees.

3.1 Operation of the Master

The master (a)traverses the game tree from the root node to some extent, (b)assigns subtrees to the slaves, (c)receives search results from the slaves, and (d)backs up proof and disproof numbers. Though the depth searched by the use of proof and disproof numbers is variable, the depth searched by the master is fixed to d'. The benefit of the fixed depth-first search is that (a)(b)(d) can be done fast and simultaneously. To achieve this mechanism, the master has a tag field at its transposition entries. There are two kinds of tags: One is an UNCERTAIN mark that represents that this node is now being searched by one of the slaves. The other is an EXPAND mark that stands for this node has a descendent that has an UNCERTAIN mark.

The master expands in a depth-first manner the nodes marked EXPAND and selects candidate nodes to assign to the slaves, as will be explained later. When the master reaches the depth d', the proof and disproof number, whose current value is found in the transposition table of the master, are backed up, and EXPAND marks are erased if there are no UNCERTAIN marks in the descendents of a node.

While expanding game trees in a depth-first manner, the master selects the candidates to assign to the slaves. This selection phase makes use of the property of proof and disproof numbers. The master selects nodes to be expanded the proof and disproof number a little bigger than the root node. Let $n.\Delta\phi, n.\Delta\delta$ be small integers added to $n.\phi$ or $n.\delta$ respectively, n_i be the child of n that is about to be expanded. n_i can be a candidate node if the condition $n_i.\delta - n.\phi < n.\Delta\phi$ is satisfied. $n_i.\Delta\phi$ and $n_i.\Delta\delta$ is set to $n.\Delta\delta$ and $n.\phi + n.\Delta\phi - n_i.\delta$ respectively. If the master reaches the depth d' and a leaf node n is a candidate node that is not marked UNCERTAIN, the master marks n as UNCERTAIN and allocates n to one of the slaves with a threshold of $n.\phi + n.\Delta\phi$.

The selection phase begins with $root.\Delta\phi = root.\Delta\delta = 1$, and the master tries to partition into the subtrees. If there are insufficient nodes for the slaves and $root.\phi$ and $root.\delta$ is the same as each previous value, the master increments $root.\Delta\phi$ or $root.\Delta\delta$ and restarts the selection again, and continues this increment until enough nodes are generated. The strategy for incrementing is the same as PDS:

(1) If $root.\phi + root.\Delta\phi \leq root.\delta + root.\Delta\delta$, increment $root.\Delta\phi$.

(2) Otherwise increment $root.\Delta\delta$.

Our implementation of receiving the search results from the slaves is that the master checks for new results from the slaves every time before expanding, and updates the contents of the transposition table , and removes the UNCERTAIN mark if there is a reply from the slave.

3.2 Operation of the Slaves

The algorithm of the slaves is PDS with one modification: While PDS continue searching to prove or disprove game trees, our algorithm of the slaves expands only up to the threshold decided by the master.

(1) Search the node n assigned by the master with the threshold of $n.\phi + n.\Delta\phi$.

(2) Return the result to the master at the end of each iteration.

(3) Ask the master to clear the UNCERTAIN mark after the search is over.

This process repeats until the master has completed the search.

If the slave has finished searching the node assigned by the master, there may be some idle time until the master assigns the next node to the slave. When there is no work waiting for the slave, it increments the threshold, and starts the next iteration. When a new node arrives from the master, the slave stops searching immediately and starts to work on the new node.

3.3 Current Implementation of Transposition Tables

Since we implement our algorithm on a distributed memory machine, it is necessary to consider the implementation of transposition tables. At present our implementation of the transposition table is *local*. Each processor has a transposition table locally without exchanging its information with processors. The local transposition tables can eliminate communication overhead, but they offer a great deal of search overhead, compared with distributed transposition tables[3]. The local transposition tables can easily use a garbage collection scheme such as SiblingGC proposed by Nagai[8]. When the transposition table becomes full, it is necessary to throw some information away. If $n.\phi$ becomes 0, SiblingGC discards the information on the siblings of n and their descendents from the transposition tables. SiblingGC achieves dramatical reduction of the amount of memory compared with ad hoc replacement schemes such as [12]. All we have to do in ParaPDS with local transposition tables is to do SiblingGC locally. It would

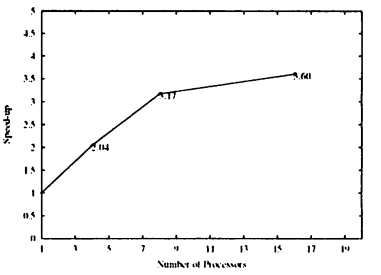

Figure 2: Ratio of Speed-up

be necessary to consider some good methods to implement distributed transposition tables with SiblingGC, because it must exchange the infomation on the transposition tables among the processors whenever SiblingGC runs.

For the purpose of avoiding the increase of search overhead by the use of local transposition tables, the master must carefully select which slave to assign the subtrees. Our implementation is a natural one. Let C be the number of nodes currently allocated to the slaves, k be the number of slaves, w_i be the number of nodes that the slave i has, n be the node about to be assigned.

(1) Assign n to the slave i if n is assigned to the slave i and $w_i < \lceil C/k \rceil$.

(2) Find a slave that has no node, and assign n to it.

(3) Otherwise assign to a slave in a round robin manner.

4 Experimental Results

We implemented ParaPDS on a HITACHI SR-2201, a distributed memory machine whose CPU is 150MHz and each processor has 256MB memory. We used 8 positions from Othello endgames each of which has 20 empty squares

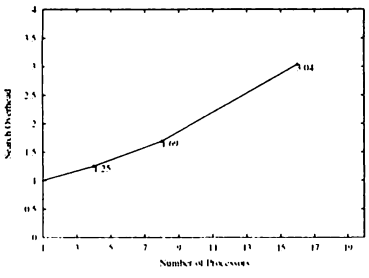

Figure 3: Ratio of Search Overhead

as a sample test set. The search depth of the master is 5, each processor has 500,000 positions in its transposition table, and can expand nodes at a rate of 6,500 nodes per second.

Figure 2 and 3 show the speed-up and the increase of search nodes. The speed-up increases as we increase from 4 to 16 processors, and the speed-up with 16 processors is 3.60. However, the ratio of speed-up from 8 to 16 processors seems not to be good, which can be attributed to the increase of search overhead. Though the ratio of search overhead is 1.69 with 8 processors, it becomes 3.04 with 16 processors. One of the important reasons why the search overhead increases is that we use local transposition tables. There are two demerits of local transposition tables: One is information deficiency, and the other is load imbalance that is caused by allocating a node to the slave that has its information in the transposition table as possible. Table 1 shows the bottlenecks by using local transposition tables. Hash miss means that a slave hash no information in its local transposition table on a node assigned by the master. The ratio of overwork can be calculated as follows if we let k be the number of the slave processors, s_i be the number of really effective search nodes of the processor i which is calculated by subtracting the number of search nodes of i that was done to reduce

n	Ratio of Hash Miss	Ratio of Overwork
1	-	-
4	0.16	1.20
8	0.35	2.06
16	0.36	2.43

Table 1: Bottlenecks by Using Local Transposition Tables

the idle time[2] from the total number of search nodes of i:

$$\text{Ratio of overwork} = \frac{\max(s_1, \cdots, s_k) \times k}{s_1 + \cdots + s_k}$$

While the ratio of missing information of transposition tables between 8 and 16 processors is almost the same, the ratio of overwork with 16 processors is inferior to that with 8 processors. While the overworked processor searches a new important node assigned by the master, the rest slaves continues searching the previous nodes from the master to reduce the idle time. The results of search by the rest slaves are mostly of no use, which increases the search overhead.

5 Conclusion and Future Work

In this paper we proposed an asynchronous parallel algorithm based on proof and disproof numbers, achieved a speed-up of 3.60 in solving some Othello endgame examples on a distributed memory machine with 16 processors.

There are some directions for further work. First of all, since we have an environment with

[2] As mentioned in Section 3.2, the slaves search the previous nodes assigned by the master until they receive new nodes from the master.

64 processors, we must show the experimental results in the case of 64 processors. Second it is indispensable for PDS to the information on proof and disproof numbers, we must develop distributed transposition tables which have a lot of possibility to achieve more speed-ups by the decrease of search overhead. Finally, our current implementation may cause load imbalance with the increase of the number of the processors. We must develop something new to achieve good load balance.

6 Acknowledgment

We would like to thank Ayumu Nagai not only for his kind explanation of his sequential program that is available at his web page, but also for his sharp questions of our parallel algorithm, and Martin Müller for reading an early draft of this paper and giving us some valuable comments.

References

[1] Louis V. Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.

[2] Mark G. Brockington. *Asynchronous parallel game-tree search.* PhD thesis, University of Alberta, 1998.

[3] Rainer Feldmann. *Spielbaumsuche auf massiv parallen Systemen(Game trees on massively parallel systems).* PhD thesis, University of Paderborn, 1993. English translation is also available.

[4] Richard M. Karp and Yanjun Zhang. On parallel evaluation of game trees. In *1989 ACM Symoposium on Parallel Algorithm and Architecures*, pages 409–420, 1989.

[5] Tony A. Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):442–452, 1985.

[6] David A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[7] Ayumu Nagai. A new AND/OR tree search algorithm using proof number and disproof number. Booklet for the Workshop of the First International Conference on Computers and Games(CG'98), 1998. The postscript file is available at http://www.etl.go.jp/~7236/Events/workshop98/.

[8] Ayumu Nagai. A new depth-first search algorithm for AND/OR trees. Master's thesis, University of Tokyo, 1999. His sequential Othello program is available at http://www.is.s.u-tokyo.ac.jp/~nagai/.

[9] Yasuichi Nakayama, Tadafumi Akazawa, and Kohei Noshita. A parallel algorithm for solving hard tsume-shogi problems on the workstation cluster. *ICCA Journal*, 19(2):94–99, 1996.

[10] Monroe Newborn. Unsynchronized iteratively deepening parallel alpha-beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):687–694, 1988.

[11] Masahiro Seo. The C* algorithm for AND/OR tree search and its application to a tsume-shogi program. Master's thesis, University of Tokyo, 1995.

[12] Masahiro Seo. Solving tsume-shogi using conspiracy numbers(in Japanese). In Hitoshi Matsubara, editor, *Advances in Computer Shogi*, volume 2, pages 1–21. Kyoritsu Shuppan, 1998.