APHID: Asynchronous Parallel Game-Tree Search

Mark G. Brockington and Jonathan Schaeffer
Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2H1
Canada

February 12, 1999

Running Head: APHID: Asynchronous Parallel Game-Tree Search

Send Proofs To:
Jonathan Schaeffer
615 General Services Building
Department of Computing Science
University Of Alberta
Edmonton, AB T6G 2H1
Canada
(403) 492-3851

Abstract

Most parallel game-tree search approaches use synchronous methods, where the work is concentrated within a specific part of the tree, or at a given search depth. This article shows that asynchronous game-tree search algorithms can be as efficient as or better than synchronous methods in determining the minimax value.

APHID, a new asynchronous parallel game-tree search algorithm, is presented. APHID is implemented as a freely-available portable library, making the algorithm easy to integrate into a sequential game-tree searching program. APHID has been added to four programs written by different authors. APHID yields better speedups than synchronous search methods for an Othello and a checkers program, and comparable speedups on two chess programs.

Keywords: parallel search, alpha-beta, computer games, heuristic search.

List of Symbols:

$\alpha\beta$:	alpha, beta
d:	dee
d':	dee prime

1 Introduction

Making computers play games in a skillful manner, comparable to that of a strong human player, is a challenging problem that has attracted the attention of many computer scientists over the last fifty years. Two-player zero-sum games with perfect information, such as chess, Othello[1] and checkers, are programmed using the same basic techniques. The $\alpha\beta$ algorithm [9] is used to exhaustively search variations that are d moves deep in a depth-first manner to determine the best move and its value. A large hash table, called the *transposition table* [5], is used to store previously determined best moves and values for positions. The values from this table are re-used during the search to prevent the same position from being explored twice.

Instead of immediately searching a variation d moves deep (or d *ply*), most programs search to 1 ply, then to 2 ply, *et cetera*. This technique is known as *iterative deepening* [16], and is used to acquire move-ordering information in the transposition table. The best move for a $(d-1)$-ply search is likely to be the best move for a d-ply search, and the $\alpha\beta$ algorithm will build a smaller search tree (by eliminating, or cutting-off, irrelevant subtrees) if the best move is searched first.

A game-playing program that can out-search its opponent has a high probability of winning. It has been shown that there is a strong correlation between the search depth and the relative strength of chess, Othello and checkers programs [8]. Thus, programs are developed to search as deeply as possible while staying within the time constraints imposed by the rules of the game.

When using parallelism to search game trees deeper, almost all of the research has concentrated on synchronous parallel search algorithms. These algorithms force work on one part of the tree to be completed before work on the rest of the tree can be carried out. There are global synchronization points during the search that all processors must reach before any process is allowed to proceed. In some synchronous algorithms, the work is synchronized at every choice along the hypothesized best-move sequence, commonly known as the principal variation. In all synchronized algorithms, the work is synchronized at the root of the tree between steps of iterative deepening; a

[1] Othello is a registered trademark of Tsukuda Original, licensed by Anjar Co.

complete d-ply search must be finished before the $(d + 1)$-ply search can begin.

The advantage of the synchronous approaches is that they can use the value of the principal variation in the same way as the sequential search algorithm does. Synchronous parallel algorithms are successful at keeping the size of the search tree built close to the sequential search tree size, assuming that processors are able to share transposition table information in an efficient manner. However, there are fundamental problems with synchronous parallel $\alpha\beta$ search algorithms:

1. There are many times when there is insufficient parallelism to keep all the processors busy. If there are more processors than work granules at a synchronization point, then some processors must go idle. This idle time increases in magnitude (and significance) as the number of processors increases. This problem is exacerbated in games that have a small average number of move choices.

2. They require an efficient implementation of a shared transposition table between the processes to achieve high performance. Typically, the algorithms will exhibit poor performance without such a table, since each processor's search results must be made available to all other processes. Because of this, most synchronous algorithms are tested on shared memory systems. On distributed memory systems, sharing a table is not as efficient, and the speedups portrayed in the literature for shared memory systems are not achievable.

3. Many synchronous algorithms attempt to initiate parallelism at nodes which are better done sequentially. For example, having searched the first branch at a node and not achieved a cut-off, the remaining branches are usually searched in parallel. However, if the second branch causes a cut-off, then all of the work done on the third and subsequent branches was unnecessary. This suggests that parallelism should only be initiated at nodes where there is a high probability that all branches must be considered.

4. Many of the synchronous algorithms do not integrate well into typical sequential algorithms. This causes many changes to the main search algorithm to incorporate parallelism. This will likely result in a parallel program for which it is difficult to verify its correctness.

In other work, a theoretical model was developed for comparing a typical synchronous game-tree search algorithm to an asynchronous one [2]. The theoretical results indicated that an asynchronous algorithm could outperform a synchronous algorithm on game trees similar to those seen in practice. This paper shows that it is possible for asynchronous search algorithms to outperform their synchronous counterparts in practice.

The paper's major contributions include:

1. The APHID (Asynchronous Parallel Hierarchical Iterative Deepening) algorithm is introduced that addresses the previously mentioned problems. First, the algorithm is asynchronous in nature; it removes all global synchronization points from the $\alpha\beta$ search and from iterative deepening. Second, the algorithm does not require a shared transposition table for move ordering information. Third, parallelism is only applied at nodes that have a high probability of needing parallelism.

2. APHID has been designed to conform to the structure of the sequential $\alpha\beta$-based game-tree search algorithm. Consequently, parallelism can be added to an existing application with minimal effort. APHID has been programmed as an application-independent and portable library.[2] This was used to generate all of the parallel applications reported in this article. Each of the implementations took less than a day of programming time to achieve a parallel program that executed in the same way as the sequential program, and a few days of additional tuning to achieve the reported speedups. In contrast, adding a synchronous parallel algorithm to an existing sequential algorithm may take months of work.

3. APHID's performance for four game-playing programs is presented. APHID yields better speedups than synchronous search methods for an Othello and a checkers program, and comparable speedups on two chess programs.

Section 2 gives a brief survey of relevant parallel search methods. Section 3 describes the APHID algorithm in detail, along with an illustrative example of how to add APHID into existing

[2]It is freely available at `http://www.cs.ualberta.ca/~games/aphid/`.

sequential game-tree search code. Section 4 shows the experimental results of adding APHID to four applications. Section 5 summarizes this paper.

2 Previous Work

2.1 The $\alpha\beta$ Algorithm

In a two player zero-sum game with perfect information and a finite number of moves, an optimal strategy can be determined by the minimax algorithm. In general, minimax is a depth-first search of a game tree. Each node represents a position, and the links between nodes represent the move required to reach the next position. The player to move alternates at each level of the tree. The evaluation of each leaf node in the search is based on an approximation of whether the first player is going to win the game (heuristic evaluation function). Whenever the first player moves, he chooses a move to maximize his evaluation (so-called Max nodes in the search tree). Conversely, when the second player has to move at a node, he will choose the move that minimizes the evaluation (Min nodes).

Although the tree is finite, it can get quite large. To search a tree of depth d and with b branches at every node, b^d nodes would be evaluated. Fortunately, there are straightforward "bounding" techniques that can prove some evaluations are irrelevant. The most popular of these is the $\alpha\beta$ algorithm, given in Figure 1.

The $\alpha\beta$ algorithm only considers nodes that are relevant to the search window [α, β]. α represents the best value that the side to move can achieve thus far in the search. Additional search at this node is intended to find moves that improve this lower bound (i.e. have a value $> \alpha$). β is the best that the player to move can achieve or, conversely, the smallest value that the opponent can provably restrict the side to move to. When $\alpha \geq \beta$, no additional search is needed at this node (a *cut-off* occurs). In effect, this *prunes* parts of the tree that provably cannot contribute to the minimax value. It has been shown that the $\alpha\beta$ algorithm will return the correct minimax value if the root position is searched with $\alpha = -\infty$, $\beta = +\infty$ [9]. For a depth d tree with b branches at every node, $\alpha\beta$ has the potential to search the minimum number of leaf nodes possible to de-

```
int AlphaBeta(position p, int alpha, int beta) {

   int numOfSuccessors;     /* total moves */
   int i;                  /* move counter */
   int sc;     /* score returned by search */

   if (EndOfSearch(p)) { return(Evaluate(p)); }

   numOfSuccessors = GenerateSuccessors(p);
   for(i=1; i <= numOfSuccessors; i++) {
      sc = -AlphaBeta(p.succ[i],-beta,-alpha);
      alpha = max(alpha, sc);
      if (alpha >= beta) { return(alpha); }
   }
   return(alpha);

} /* AlphaBeta */
```

Figure 1: A Negamax Formulation of the $\alpha\beta$ Algorithm

termine the minimax value: $b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$ (assuming that there are no transpositions in the tree). This best case is achieved when the "best" move is considered first at all nodes in the search tree (a perfectly-ordered tree).

There is a portion of the $\alpha\beta$ tree that must always be searched to determine the minimax value. This *critical tree* is defined as the perfectly-ordered tree that is generated when $\alpha\beta$ is started with the search window ($-\infty$,$+\infty$). Figure 2 shows the structure of the critical tree. Nodes marked ALL have all of their successors explored by $\alpha\beta$. Nodes marked CUT have at least one branch that can cut off further search at this node. In the critical tree, first move searched at all CUT nodes causes a cut-off. CUT and ALL nodes are also known as type-2 and type-3 nodes, respectively [9].

The principal variation (type-1 nodes, labeled PV in Figure 2) of an $\alpha\beta$ critical tree is the first (left-most) branch searched. All of the PV nodes are searched with the window ($-\infty$,$+\infty$). Thus, all children at PV nodes are searched, meaning that they are effectively ALL nodes.

2.2 Parallel $\alpha\beta$-based Search Algorithms

The *PV-Split* algorithm [13] is based on the regular structure of the critical game tree. The first stage of the algorithm involves a recursive call to itself as PV-Split travels down the principal variation. Once the left subtree of a PV node has been examined, all of the other subtrees below that PV node are searched in parallel. Each processor is given one subtree at a time to search, without