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

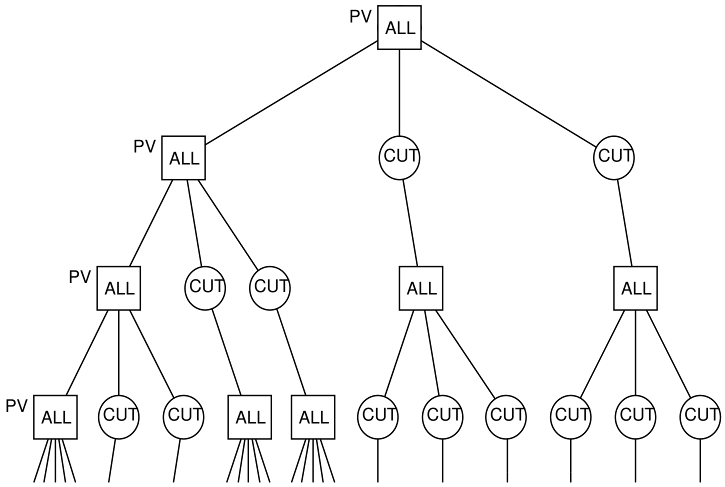

Figure 2: The Critical Tree

assistance from other processors. After all of the subtrees have been completely explored, that PV node returns its score to the PV node above it. At any time, only one node's subtrees are being examined in parallel. The major problem with PV-Split is the large amount of synchronization; many processors are often forced to wait for long periods of time while the last unevaluated branch of a PV node is finished.

The best speedups for parallel $\alpha\beta$ have been reported for the *Young Brothers Wait Concept* (YBWC) algorithm [3]. In this paper, YBWC is used as a representative of a large group of popular synchronous algorithms (including ABDADA [17], Dynamic Tree Splitting [7] and Jamboree [11]) which all share the same underlying parallel algorithm and differ only in implementation details. All of these algorithms synchronize at the end of each iteration, and may also synchronize at additional points within each iteration.

In a game tree that has near-perfect move ordering, there is a high probability that a node is an ALL node if the left-most branch is evaluated and does not cause a cut-off. The basic YBWC states that the left-most branch (the eldest brother) must be evaluated before any other branches (the young brothers) can be distributed to other processors. This is not necessarily limited to the principal variation (i.e. PV-Split) or a pseudo-principal variation as in other algorithms; it can happen at any node within the game tree. The optimized algorithm is called YBWC*. In this variant, young brothers are not forced to wait at ALL nodes, allowing sequential evaluation of all "reasonable" moves at CUT nodes.

The UIDPABS algorithm (*Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search*) [14] was the first attempt to asynchronously start the next level of an iteratively deepened search instead of synchronizing at the root of the game tree. The moves from the root position are partitioned among the processors, and the processors search their own subset of the moves with iterative deepening. Each processor is given the same initial search window, but some of the processors may have changed their windows based on the search results of their moves. The UIDPABS algorithm combines all the processors' results once a predetermined time limit has been reached. Some of the moves may have been evaluated to larger depths than those on other processors, which may yield a better quality move choice. However, it is important to note that each move may only be searched by one processor; thus, the algorithm can only use as many processors as there are moves at the root of the game tree.

3 The APHID Algorithm

This section describes the Asynchronous Parallel Hierarchical Iterative Deepening (APHID[3]) game-tree search algorithm. APHID represents a departure from the synchronous algorithms described in Section 2 and has been designed to address the problems described in Section 1. 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, although one can be used if duplicate detection is important in the underlying application. Third, parallelism is only applied at nodes that have a high probability of needing parallelism, and this decision is based on the best information available at the time. Finally, APHID is designed to easily fit into an existing sequential $\alpha\beta$-based search algorithm.

APHID subdivides the tree into many distinct pieces which each process can search indepen-

[3] An aphid is a soft-bodied insect that sucks the sap from plants. One could say the APHID algorithm sucks the minimax value from a game tree.

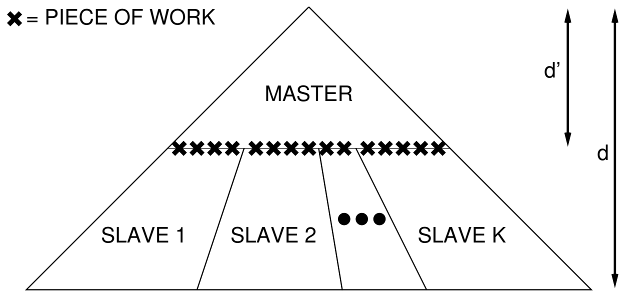

Figure 3: APHID Partitioning Game Tree Amongst Processes

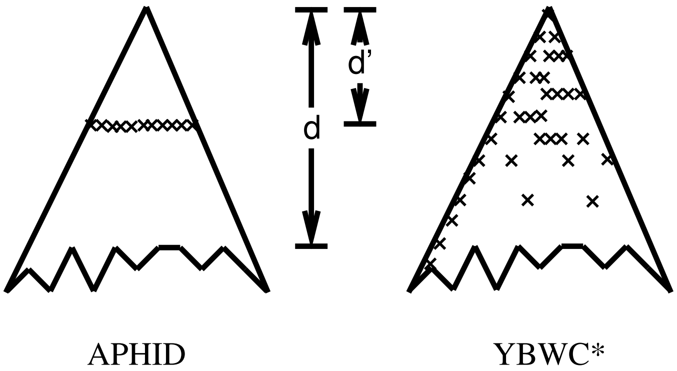

Figure 4: Location of Parallelism in Typical APHID and YBWC* Search

dently of each other. Figure 3 gives an example of how the tree would be divided by the APHID algorithm. The figure shows us a single-level master/slave hierarchy (multiple levels are possible). The master process controls the top d' ply of the game tree, including the root. The leaves of the master's d'-ply tree are the pieces of work that each slave process examines. Each of the k slave processes gets a portion of the master's leaves to search, implicitly dividing the remainder of the tree as shown in Figure 3. Note that the diagram is partially misleading in that, for load balancing purposes, a slave's pieces of work are actually distributed across the bottom of the master's tree.

It is interesting to compare APHID against a synchronous parallel algorithm. Figure 4 illustrates graphically where work is allocated over the course of a typical APHID and YBWC* search. Each location marked with an x shows where the parallelism typically takes place. Although more parallelism could be generated in YBWC*, one must be careful because each x along the left side of the YBWC* tree represents a global synchronization point, and the rest are local synchronization points.

In APHID, the master process makes repeated searches or *passes* over its part of the game tree. The master's tree is small and is quickly searched. The master is responsible for ensuring that the slaves determine all of the minimax values required to get a d-ply $\alpha\beta$ search of the full game tree finished, after which it proceeds, via iterative deepening, to the next search depth repeatedly until the time limit expires.

Each slave searches its pieces of work $(d - d')$ plies deep, plus any search extensions that the master requires. If APHID were a synchronous algorithm, the slaves would finish their work for the master's depth d iteration, and then wait for the next iteration $(d + 1)$ to begin. APHID's slaves do not sit idle waiting. Instead, they use iterative deepening to search their work lists an additional ply deeper (and beyond, if there is time) without confirmation from the master. Thus, the slaves attempt to determine minimax values that may be necessary, in anticipation of the master process asking for them. The slaves continue searching until the master signals that the search at the root of the tree has been terminated.

By partitioning the tree in this manner, APHID's performance does not rely on the implementation of a global shared memory or a fast interconnection network between the processes. This makes the APHID algorithm suitable for loosely-coupled architectures (such as a network of workstations), as well as tightly-coupled architectures.

This section contains a description of the master process (Section 3.1), the slave processes (Section 3.3), and the APHID table through which the master and slave communicate (Section 3.2). Balancing the work load between the slaves is discussed in Section 3.4. The interface between APHID and the application is described in Section 3.5.

3.1 The APHID Master

The master is responsible for ensuring that the result of a d-ply search is obtained. The amount of work retained by the master, d' ply, is automatically determined based on the characteristics of the search tree generated for the root position and domain-dependent information provided by the user. Further details on the automatic determination of d' can be found elsewhere [2].

APHID completes a d-ply $\alpha\beta$ search by repeating the following steps:

1. Execute a quick search (or *pass*) of the d'-ply tree using the application's sequential $\alpha\beta$ algorithm (where $d' < d$), aided by a transposition table (if available).
2. If the search returns an accurate d-ply value, then exit the loop.
3. Based on the leaf nodes visited during this pass, report any changes to the slaves' work lists.
4. Wait for new information from a slave process.
5. Go to step 1.

The d'-ply tree is not kept in memory. Thus, on each pass the tree is traversed to determine any changes to the work required. The tree is regenerated on each pass by the application's $\alpha\beta$ algorithm and transposition table. Since d' is usually small, the cost of a pass is also small.

During a pass, when the master reaches a leaf of the d'-ply tree, an evaluation of that node must be done. If the subtree underneath that node $(d-d')$ is smaller than g ply (the *minimum granularity* for a piece of parallel work), then the master searches the remaining $(d-d')$ ply itself. Otherwise, the node is assigned to a slave processor.

Assuming that the master does not search the tree underneath a given node ($g \leq (d-d')$), APHID must determine a value with respect to the $\alpha\beta$ search window at that node without further search. For example, an $\alpha\beta$ search window of (0,5) says that the $\alpha\beta$ algorithm needs to know whether the node's minimax value is ≤ 0, ≥ 5, or the exact value if it lies in between 0 and 5. Without knowing which case is correct, the $\alpha\beta$ search may give inaccurate results.

If the slave has already given the master a $(d-d')$-ply search result, that value is used.[4] If the $(d-d')$-ply result is unavailable because the slave has not reached that depth of search, or the $(d-d')$-ply result has returned a bound that yields insufficient information on the minimax value with respect to the $\alpha\beta$ search window, then the algorithm guesses at the minimax value. Going

[4]In the implementations presented here, the deeper ply values are not used even if they are available. This is discussed in Section 4.

back to our earlier example, a value of ≤ 10 does not yield useful information at a node with an $\alpha\beta$ search window of (0,5), hence a guessed value for this node would be used until a slave could return better information. Any node where APHID has to guess at the value is marked as an *uncertain* node.

For uncertain nodes, information gathered from both previous searches and the current search is used to determine a guessed value. The guessed value is chosen so that it will not alter the shape or ordering of moves within the game tree until new information is received from a slave processor. Details of the guessed score algorithm are provided in Appendix A.

As minimax values get backed up the d'-ply tree during the search, the master maintains a count of how many uncertain nodes have been evaluated. When the master has no uncertain evaluations in its d'-ply tree, the value of the complete d-ply tree is accurate, and APHID exits the loop.

Sometimes, the $(d - d')$-ply result may be insufficient for the master to determine the minimax value. A further search at the same depth, with a different search window, is required when this happens. The master sends a message to the slave responsible for searching the node that it needs to re-execute the search with a revised search window. The slave will eventually return updated information that is consistent with both the original information and the search window requested.

APHID solves one of the problems that synchronous algorithms have with respect to initializing parallelism incorrectly at a potential CUT node. Most synchronous algorithms use application-dependent information to determine when to initiate parallelism. By using the guessed scores when accurate information is not available, the APHID algorithm automatically determines if a subsequent child is likely to generate a cut-off at a failed CUT node. Nodes which are pruned by the master, using either real or guessed scores, are not searched by the slaves. If it seems unlikely that the node will be pruned (e.g. due to low real/guessed values), then all children are visited and the work would be initiated in parallel. This is all handled implicitly in an application-independent way by the $\alpha\beta$ routine.

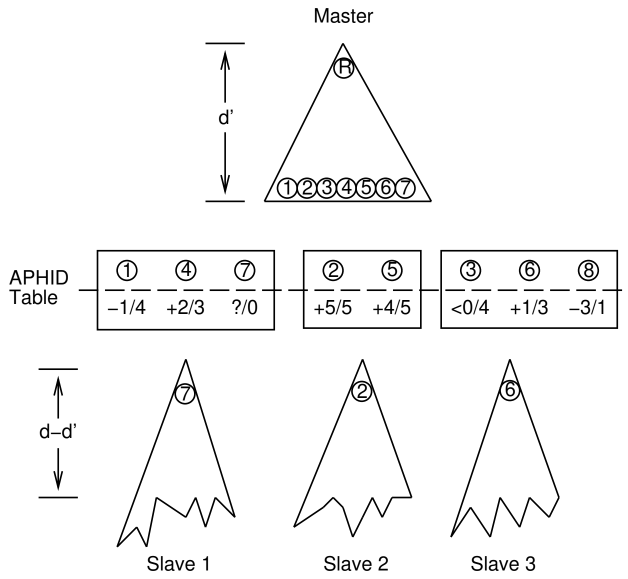

Figure 5: A Snapshot of APHID Search in Operation

3.2 The APHID Table

If a leaf node is visited by the master for the first time, it is allocated to a slave process. This information is recorded in the *APHID table* that is shared by all processes. Figure 5 shows an example of how the APHID table would be organized at a given point in time.

The table is replicated on the master and slave processes. However, each slave only knows of the entries relevant to it within the table. For example, in Figure 5, slave 1 only knows about the entries for 1, 4 and 7. The master, which is responsible for distributing the work to all of its slaves, has copies of every table entry.

The master and slave only read their local copies of the information; there are no explicit messages sent between them asking for information. The entries in the APHID table are partitioned into two parts: one which only the master can write to, and one which only the slave that has been assigned that piece of work can write to. Any attempt to write into the table generates a message that informs the master or slave process to update its replicated copy of the table entry.

The master's half of the table is illustrated above the dashed line in Figure 5. For each leaf visited by the master, there is an entry in the APHID table. Information maintained on the leaves includes the moves required to generate the leaf positions from the root R, the approximate location of the leaf in the tree (which is used by the slave to prioritize work), whether this leaf was visited on the last pass that the master executed, and the number of the slave that the leaf was assigned to.

In our example, roughly the same number of leaves have been allocated to each slave. Note that there is an additional leaf, 8, that is not represented in the master's d'-ply search tree. This leaf node was visited on a previous pass by the master, but was not visited on the latest pass. However, the information that the slave has generated about this node may be needed in a later pass, and is not deleted from the table. Leaves are initially allocated to the slaves in a round-robin manner, and may move due to load balancing (as described in Section 3.4).

The slave's part of the table (the area below the dashed line) contains information on the minimax value at various depths of search. The best information (with respect to search depth) and the ply to which the leaf was examined is given underneath each leaf node. For leaf 1, the score returned is -1 with a search depth of 4. Leaf 3 illustrates that the score information returned by the slave may not always be an exact number. The slaves maintain upper and lower bounds on the minimax value of each node for each ply of search depth.[5] This information is determined by the minimax value returned and the search window used by the slave. For simplicity, only a single bound is shown.

The APHID table data structure must support two time critical operations: finding a node in the table, and adding a node to the table. Hence, the APHID table has been implemented as an extendible hash table. Applications that would use APHID generally have a transposition table; the transposition table hashing function can also be used to find and add leaves to the APHID table.

3.3 The APHID Slave

A slave process essentially executes the same code that a sequential $\alpha\beta$ searcher would. The slave's main loop simply repeats the following steps until the master tells it that the search is complete:

[5] Note that two bounds may be needed. A search may return an upper bound on a score. That node may have to be re-searched to get more information on its score. The second search may return a lower bound that is good enough to cause a cutoff.

1. Look in its local copy of the APHID table and find the best node to search.

2. Execute the search.

3. Report the result back to the master (fetching any update to its APHID table in return).

4. Go to step 1.

A slave chooses the "best" node based on two criteria: the depth to which the node has already been searched by the slave, and the node's "priority" within the search tree. The first criterion ensures that all pieces of work are searched to equal depths. Nodes searched to shallower depths are preferred over those searched to deeper depths, because they represent more work to be done. In Figure 5, slave 1 has master leaf nodes 1, 4 and 7, and they have been searched to depths 4, 3 and 0, respectively. Thus, slave 1 is attempting to search leaf 7 to 1 ply, and will continue to search leaf 7 up to 3 ply using iterative deepening, if no new high-priority work arrives from the master.

The second criterion is the location of the node within the master's latest pass over the tree. Children of nodes are usually considered in a best-to-worst ordering, implying that the left-most branches at a node are less likely to be pruned than the right-most ones. For slave 2 in Figure 5, leaves 2 and 5 have both been searched to 5 ply, but leaf 2 is being searched to 6 ply since it is further left in the tree than leaf 5.

Unfortunately, maintaining a complete ordering of each leaf in the master's d'-ply tree can be expensive. Thus, APHID uses a priority scheme to give an approximation of the second criterion. The formula for the priority is as follows: For each principal variation node that lies on the path from the root of the tree to the node in question, four points are added to the node's priority. Two points are given to the node's priority if it is considered to be part of the critical tree. The priority scheme ensures that the search proceeds in a roughly left-to-right manner, and that nodes within the critical tree are preferred over nodes that might not be evaluated. Every node visited in a pass of the game tree by the master gets at least 4 points, since the root is part of the principal variation. Nodes that are not touched on the master's latest pass of the game tree are given a priority of zero.

A node with zero priority will never be selected for further search by a slave. For slave 3, notice that leaf 8 has been searched previously, but not on the last pass. This leaf is ignored by the slave's work selection algorithm because it is not currently part of the master's tree.

If there are nodes that must be searched for the current iteration of the master, the node with the highest priority is always scheduled until it has been searched to the requisite depth. This allows the master's development of the search tree to proceed in an orderly manner. When all nodes at a slave have been searched to the master's required depth, the nodes at the lowest search depth have their search extended, with priority values as a secondary consideration.

Before a search can be executed, an $\alpha\beta$ search window must be generated by the slave. The window selection algorithm is application-dependent. The general-purpose window selection algorithm used by the applications in this paper involved centering the search window around the probable minimax value of the game tree. The search window was made slightly larger to reflect the amount of uncertainty in the probable minimax value. Further details on the algorithm can be found in Appendix B.

3.4 Load Balancing

Although the master attempts to give an equal amount of work to each slave in APHID, neither the master nor the slave can predict the amount of effort required to complete a $(d - d')$-ply search for a given piece of work. Thus, load imbalances can occur.

As part of a pass of the d'-ply tree, the master computes how many uncertain nodes it is waiting for from each slave. The master can move leaves of the d'-ply tree from an *overworked slave* (with a large number of uncertain nodes) to an *underworked slave* (with no uncertain nodes). This yields a tradeoff between faster convergence for a given ply search of the tree and additional search overhead, since the previous searches for the piece of work to be moved must be re-searched on another processor.

The load-balancing algorithm always attempts to strip pieces of work away from the most overworked slave. The algorithm prefers to take pieces of work that are small, since they lead to less duplicated work. The first uncertain node encountered for each slave during a pass is not considered for load-balancing purposes, since it is likely being searched by that slave at this time. Another stipulation is that the same piece of work cannot be moved twice in a row; this prevents a very small piece of work from being passed from process to process.

Another cause of a load imbalance is a piece of work that is much larger than the other pieces of work. For example, the search tree for a node along the principal variation is generally much larger than the last subtrees examined during a sequential search. When there is such a large piece of work, multiple processes should participate in computing the minimax value. Thus, a mechanism is needed for breaking a large piece of work into a number of smaller pieces that can be distributed (via the load-balancing algorithm) to other processes. This can be accomplished by moving the master's parallelization horizon deeper within the tree for a large piece of work. This allows the master to subdivide a single piece of work into many smaller pieces. It could be said that the large piece of work is *exempted* from the parallelization horizon at d'-ply. Note that new pieces of work created from exemptions can also be exempted. Thus, Figure 4 is misleading; APHID parallelism does not necessarily occur at the same depth in the tree.

The master is responsible for determining the pieces of work to be exempted. It bases these decisions on the slave's feedback on the effort required to search each piece of work (subtree size). Periodically, the master determines the largest pieces of work that have been explored recently, along with the average size of each piece of work. If the size of the largest piece of work is v times the size of the average piece of work, the largest piece of work is subdivided in future searches. v is an application-dependent parameter in APHID.

The nature of the $\alpha\beta$ algorithm does not guarantee that many more work granules will be created if the search horizon is extended by a single ply. For example, if the node to be exempted is a CUT node, then the search will likely generate a single ALL node. When APHID exempts a large piece of work, the horizon is extended by 2 ply to guarantee that the work will be split into multiple pieces.

3.5 External Interface of the APHID Algorithm

The APHID algorithm has been written as an application-independent library of C routines (using PVM for process communication [4]). It was designed to provide minimal intervention into a working version of sequential $\alpha\beta$. For a program to use the library, a few application-dependent routines (such as move format, how to make/unmake moves, setting a window for a slave's search, *etc*.) must be written so that APHID can access the required information without having to know the data structures used in the application.

To parallelize a sequential $\alpha\beta$ program, the user modifies their search routine as shown in Figure 6. The changes required by APHID are marked by shading, and easily fit into the standard $\alpha\beta$ framework. This one piece of code functions as the search algorithm for both the master and the slave processes. The `aphid_master` routine identifies whether the process is a master or a slave. The master uses `aphid_horizon` to tell if it is at a leaf node in its tree. Since games are played under real-time constraints, `aphid_checkalarm` is periodically called to check whether there is a reason to abort the search. `Aphid_intnode_start` and `aphid_intnode_end` tell the master that a search is beginning/ending for this interior node. Which searching an interior node, `aphid_intmnode_move` keeps track of which move is being searched, and `aphid_intnode_update` records the value of the search.

In addition to the above, approximately 10 lines of code have to be added outside of the $\alpha\beta$ algorithm to allow APHID to make the sequential program into a parallel program. More specific information on each of the `aphid_` functions and call-back routines can be found elsewhere [2].

Other parallel $\alpha\beta$-based algorithms require significant changes to the sequential search algorithms used in practice. If the search algorithm has been designed without regard for multitasking or a specific parallel model, integrating a parallel algorithm into the code can be a significant task. By using the sequential algorithm and call-back functions to the user's code whenever possible, APHID represents a significant decrease in the effort required to achieve a working parallel game-tree search program over its synchronous counterparts.

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

   move movelist[MAX_LEGAL_MOVES];       /* ordered list of all moves */
   int numOfSuccessors;        /* total number of moves in movelist[] */
   int gamma;                            /* current best minimax value */
   int i;                                              /* move counter */
   int sc;                                 /* score returned by search */
   int under;                         /* alpha for move to be searched */
   int over;                           /* beta for move to be searched */
   move move_opt;                                 /* current best move */
   char *p_entry;         /* pointer to physical location of TT entry */
   char *p_hash;                              /* pointer to hash value */
   char *p_key;                          /* pointer to hash table lock */
   int h_length;               /* ply position previously searched to */
   int h_score;                       /* score at h_length ply from TT */
   int h_flag;             /* type of score (VALID, LBOUND or UBOUND) */
   move h_move;                            /* recommended move from TT */

   /* Generate hash value and key for this position */
   generate_hash(p, p_hash, p_key);
   /* Fetch information from transposition table  */
   retrieve(p_hash, p_key, p_entry, h_length, h_score, h_flag, h_move);

   /* If we have searched position deep enough, use score info */
   if (aphid_master() == FALSE && h_length >= plytogo) {
      if (flag == VALID)  { return(h_score); }
      if (flag == LBOUND) { alpha = max(alpha,h_score); }
      if (flag == UBOUND) { beta = min(beta,h_score); }
      if (alpha >= beta)  { return(h_score); }
   }

   /* Evaluate position if at bottom of tree */
   if (plytogo == 0) { return(Evaluate(p)); }
   if (aphid_horizon(depth, plytogo, p_hash, p_key) == TRUE) {
      return(aphid_eval_leaf(alpha, beta, depth, plytogo, p_hash, p_key));
   }

   /* Generate move list, evaluate position if no moves */
   numOfSuccessors = GenerateSuccessors(p);
   if (numOfSuccessors == 0) { return(Evaluate(p)); }
   if (aphid_checkalarm(FALSE) != FALSE) {
      terminate_search = TRUE;
      return(0);  /* Should exit AlphaBeta quickly when alarm on */
   }

   gamma = -INFINITY;
   under = alpha;   over = beta;
   aphid_intnode_start(depth, p_hash, p_key);

   for(i=1; (i <= numOfSuccessors && score <= beta); i++) {
      aphid_intnode_move(depth, &(movelist[i]));
      make_move(p, movelist[i]);

      sc = -AlphaBeta(p,-over,-under,plytogo-1);
      /* Is a research necessary? */
      if (sc > under && i > 1 && sc < beta && plytogo > 2) {
         sc = -AlphaBeta(p,-beta,-sc,plytogo-1);
      }

      unmake_move(p, movelist[i]);
      aphid_intnode_update(depth, value);
      if (sc > gamma) {
         gamma = sc;
         move_opt = movelist[i];
      }

      /* set window for next child */
      under = max(gamma, alpha);  over = under + 1;
   }

   aphid_intnode_end(depth, score, beta);

   /* Write information into transposition table */
   h_flag = VALID;
   if (score <= alpha) { h_flag = LBOUND; }
   if (score >= beta) { h_flag = UBOUND; }
   if (h_length <= plytogo) {
      p_entry = store(p_hash, p_key, plytogo, score, h_flag, move_opt);
   }

   return(gamma);

} /* AlphaBeta */
```

Figure 6: Code Example: APHID within the $\alpha\beta$ Algorithm

It should also be noted that, in terms of ease of use, the APHID library is almost unique. Given an $\alpha\beta$ program with the code structured to use the Cilk directives, then parallel *synchronous* search algorithms, such as YBWC and PV-Split, can be easily implemented [1]. Cilk uses multiple threads to coordinate the search, where each thread represents a node where work has been split off from its parent.

4 Experimental Results

Most authors demonstrate the effectiveness of their parallel algorithm using a single application. Chess is often chosen because synchronous algorithms usually yield good speedups when the branching factor is large. In contrast, APHID's performance will be presented using four different applications, each written by different authors and with different coding styles. This ambitious comparison of performance in multiple application domains is feasible because of APHID's ease of integration into an existing sequential program.

This section presents experimental results for APHID. Section 4.1 describes the experimental methodology. Section 4.2 describes the overhead model used to analyze performance. The experimental results are presented in Section 4.3.

4.1 Methodology

APHID has been implemented in four applications: CHINOOK, CRAFTY, KEYANO and THETURK. CHINOOK is the Man vs. Machine World Checkers Champion program (principal author: Jonathan Schaeffer). KEYANO is a state-of-the-art Othello program, which ranked among the top five computer Othello programs worldwide from 1992 to 1996 (author: Mark Brockington). CRAFTY is a strong freeware chess program (author: Robert Hyatt). THETURK is a strong chess program designed by two graduate students at the University of Alberta (Andreas Junghanns and Yngvi Bjornsson). Both chess programs have been participants in a World Computer Chess Championsip. All of the applications have a finely-tuned sequential search algorithm that considers the best move first approximately 90% (or more) of the time. In practice, they each search trees that are close to the critical tree in size.

All experiments were performed on a Silicon Graphics Origin 2000 with 64 processors.[6] Each of the applications were compiled with SGI's `cc` compiler. For the parallel tests, two different versions of the code were compiled. The first parallel program could be used at run-time as a master or a slave by APHID. The second program was an executable that could only be used by APHID as a slave. This separation improved performance, since a number of callback routines used only by the master impacted the performance of the slaves when left in the $\alpha\beta$ algorithm.

Parallel and sequential algorithms often do not agree with each other about the search result (value and/or best move) when the full version of the program is used [12]. For example, different search windows can cause different search extensions to be turned on, resulting in a different value at the root. One would prefer that the parallel and sequential algorithms return identical values and principal variations, allowing for a meaningful comparison. It is difficult to make this happen without severely crippling the program's sequential search algorithm. As a compromise, most of the search result differences can be eliminated by disabling all $\alpha\beta$ window-based search extensions and reductions. This does not imply that a fixed search depth was enforced on the programs. As an example, the checkers and chess programs still used quiescence search to extend the search well beyond the d-ply search horizon.

The size of each of the searches is important to the observed speedup. In synchronous parallel game-tree search algorithms, the speedup can be improved arbitrarily by increasing the size of the search [15]. However, searching game trees that occur under tournament time controls is critical to assessing a parallel game-tree search algorithm's performance. Thus, the size of the individual tests was limited so that the average time spent searching on a 64-processor machine did not exceed the usual time controls in the game being studied. This is 180 seconds in chess, 60 seconds in Othello, and 120 seconds in the game of checkers. Consequently, the search depth achieved depended on the game being studied and the position being tested. For checkers, the search depth varied from 25 to 29 ply, while for chess it was 11 or 12 ply. For Othello, all positions were searched 15 ply

[6] Additional experiments have been done using Sun equipment to test the portability of the APHID library [2].

deep. The parallelization horizon varied from 4 to 8 ply (4 in chess and Othello, 6 to 8 ply in checkers). A suitable benchmark set was chosen for each game: the Bratko-Kopec test set for chess [10], positions from the first Chinook-Tinsley match for checkers [12], and positions from a World Championship game for Othello [2].

Using a transposition table that is large enough to accommodate results discovered during the search is important. that kept the overall size of the program and data below 400 MB was chosen. It was unanticipated that this limitation would dramatically affect the speedup on a large number of processors but, as discussed later, 400 MB proved to be inadequate for the large searches attempted in the chess and checkers programs.

Distributed transposition table schemes quickly become communication-bound on modern architectures. Thus, it is usually infeasible to implement sharing of transposition table entries on systems where shared memory is not available in hardware. The APHID library contains a method for distributing transposition table information that does not overload the interconnection network. Experiments with APHID using distributed transposition tables are available elsewhere [2]. The experiments reported in this paper were run on a shared-memory computer, allowing us to compare our results with those reported in the recent literature.

Note that the transposition table size did not increase as the number of processors increased. The sequential tests used exactly the same number of table entries as each of the parallel tests. If the parallel run used 64 processors, then each process received 1/64th of the table entries that the sequential process received. Thus, the experiments measured the scalability of the number of processors and not the scalability of memory. Other researchers have allowed the number of table entries to increase in tandem with the number of processors. This can result in an unfair comparison (and significantly inflated speedups) because the sequential algorithm's table may not be large enough to hold the critical tree in memory, whereas the parallel version can. When examining the game-tree search literature, the reader should avoid directly comparing speedups without understanding the conditions under which they were achieved.

The standard set of tests for each program involved examining a fixed-depth game tree (modulo any quiescence search) and using a shared memory transposition table, over a varying number of processors. Each program was tested on n=16, 32 and 64 processors, using a single-level hierarchy: one master allocated work to $n-1$ slaves for each test position.

There are two methods of reporting speedups and overheads for a large number of samples. The first is to add all of the searches for the multiple positions together, and perform the speedup and overhead analysis on the combined data. The second method is to perform the speedup and overhead analysis on each individual position, and average the speedups and overheads. The second method is preferred (and used in this paper) because it will not overestimate the observed speedup for a large number of test positions.

4.2 Overheads in APHID

The performance of APHID will be analyzed in terms of the impediments (overheads) to high parallel performance. Since the overhead model used in this section is slightly different than that used by other authors, the terminology used is defined here.

The *total overhead* represents the additional computing time required by the parallel algorithm on n processors to achieve the same result as the sequential version:

$$\text{total overhead} = \frac{(\text{parallel time} \ \times \ \text{n}) - \text{sequential time}}{\text{sequential time}}.$$

The total overhead can also be computed by examining the component overheads independently. The three main overheads are (1) dedicating a processor to be used exclusively as a master (the master overhead), (2) the effective decrease in nodes per second examined (parallelization overhead), and (3) the larger search tree built by the parallel algorithm (total search overhead). There is no synchronization overhead in the APHID algorithm since the algorithm operates in an asynchronous manner. The breakdown of overheads can be expressed in the following formula:

$$\begin{aligned} \text{total overhead} \quad &= \quad (1 + \text{master overhead}) \times (1 + \text{parallelization overhead}) \\ &\quad \times (1 + \text{total search overhead}). \end{aligned}$$

The *master overhead* is the approximate penalty incurred by having a single processor being allocated completely to the handling of the master. This is simply $1/(n-1)$, the benefit of adding another slave to the other $n-1$ slaves.

The *parallelization overhead* is the penalty incurred by the APHID library on the speed of the slaves. The difference between the rate at which the parallel slaves explore nodes and the sequential program's node rate is the parallelization overhead. This overhead is derived partially from the cost of using PVM, and partially from the work-scheduling algorithm on each slave. In effect, the parallelization overhead includes synchronization overhead, complexity overhead and communication overhead, as used in previous parallel $\alpha\beta$ models [12, 15].

The *total search overhead* represents the number of additional nodes searched by the algorithm while attempting to determine the d-ply minimax value. In the case of APHID, there are two overheads that combine to make the total search overhead: the search overhead (as seen in other parallel models) and the speculative search.

The *search overhead* represents the additional nodes searched to achieve the d-ply minimax value. This can be computed by dividing the number of nodes examined in the parallel search by the number of nodes examined in the corresponding sequential search. Most of this overhead is incurred when the parallel version uses a search window that is not as precise as that used by the sequential version. If a shared-memory transposition table is not used, some of the increase in search overhead can be attributed to information deficiency, since data is not shared efficiently between the processes.

The remainder of the increase in search overhead is attributable to the load-balancing algorithm. The APHID algorithm forces work to be recalculated when it is moved to another processor. When there are more processors in the system, the load-balancing algorithm is more active in balancing the workload, thus causing more search overhead.

Since each position is searched to d ply, the asynchronous nature of the slaves will result in some work being done at depth $(d+1)$ or greater. The *speculative search* represents the amount of additional search that APHID has undertaken which is beyond what the sequential algorithm would attempt for a d-ply search. It can be computed by taking the number of speculative nodes searched and dividing that by the number of nodes searched in the sequential case. In our experiments, speculative search results are computed, but these results are not used to influence the value of shallower searches. This allowed the APHID algorithm to produce the same search values as the sequential version in most cases.

Ignoring the speculative search results understates the potential of the APHID algorithm. In a real tournament game, speculative search could be used to look an extra move ahead on some key variations. It is highly likely that the variations that were speculatively extended will be in the left-most branches of the tree. Thus, sometimes key variations will be searched an additional ply (or more) deeper. This allows APHID to find important variations much sooner than a synchronous parallel program with the same observed speedup.

To quantify the effects of speculative search is difficult. To do so means that there must be a metric for measuring the quality of the move selected by an algorithm. There are a few test sets that measure move quality in chess, where the performance metric is how quickly a program determines the best move in a tactical position. There is insufficient knowledge about tactics in Othello to design a similar test. Thus, the performance metric used is to compare speedups for each program.

4.3 Test Results

Figure 7 gives the speedups for each of the programs tested, while Figure 8 gives the overheads for each program on separate graphs.

Figure 7 shows that the speedups for the Othello program, KEYANO, are significantly larger than the speedups for the other three programs. There are two reasons for this. First, the trees that were measured in KEYANO are fixed-depth trees, whereas the other programs had a variable search depth due to quiescence searches. This makes the load balancing very easy for KEYANO. APHID thrives when there is little variance in the size of each piece of work. When the equivalent of quiescence search is added to KEYANO, the speedup drops from 37 to 22 on 64 processors [2].

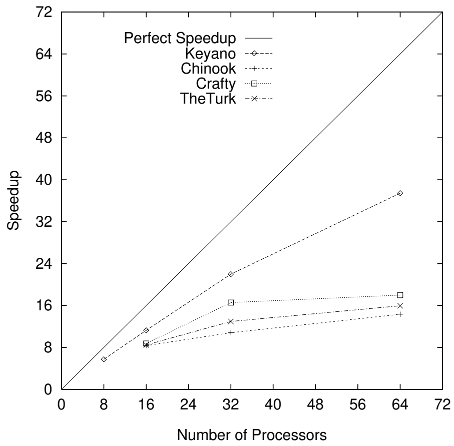

Figure 7: Speedups for All Programs

The second reason is that the transposition table is not as critical in Othello as it is in the other three programs. When all 64 processors attempt to write into a 400 MB transposition table, a processor overwrites another processor's table entries frequently. This inhibits the ability of the processors to detect positions that have already been searched. Duplicate node detection via the transposition table is important to controlling the size of the search tree in chess and checkers programs. The limited size of the transposition table is of little importance in searching Othello game trees in parallel, because achieving the same position by different sequences of moves is much more difficult. Thus, better results are obtained when searching Othello trees than chess or checkers trees.[7]

In an algorithm that does not synchronize, it should not be surprising to discover that the largest portion of the total overhead is accounted for by the search overhead and the speculative search. As the number of processes increases, the processors get fewer pieces of work and the need for load balancing increases. This also allows some of the less busy slaves to get further ahead than the overworked slaves, causing a rise in the speculative search.

To put the APHID results in perspective, a version of YBWC has been implemented in KEYANO. The algorithm was taken directly from Feldmann's thesis [3]. A considerable effort was spent attempting to optimize the performance of the algorithm. YBWC and APHID were tested under

[7]A larger transposition table (4-8 GB of RAM) for both the sequential and parallel tests would improve the speedups of the checkers and chess programs. This was not possible to test because of resource constraints.

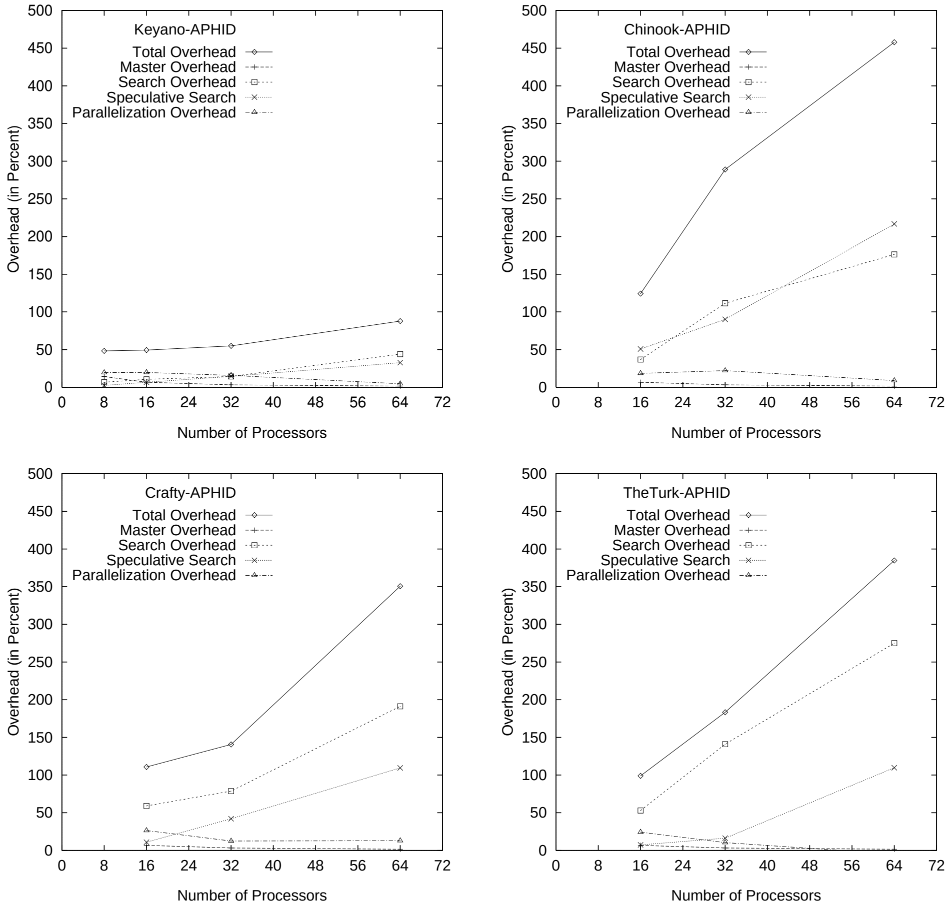

Figure 8: Overheads for All Programs

similar conditions: the same test set, same search depth, and the same size of shared memory transposition table. The performance of APHID versus YBWC can be seen in Figure 9. YBWC is rapidly starved out of work to do once 64 processors are reached. The synchronization overhead increases dramatically and the processors are only busy 60% of the time.

The search overhead in YBWC starts at a higher level than for APHID on 8 processors, but the search overhead in APHID rapidly increases past the equivalent level in YBWC. However, it is important to note that the overhead in APHID is partially due to speculative search. When APHID is run on 64 processors, a 32.65% overhead is attributed to speculative search. If the search were extended an extra ply, these speculative search results would give APHID a head start on the next iteration; YBWC would be forced to attempt the entire search from scratch. YBWC's parallelization overhead is much lower than the equivalent overheads in APHID. However, APHID

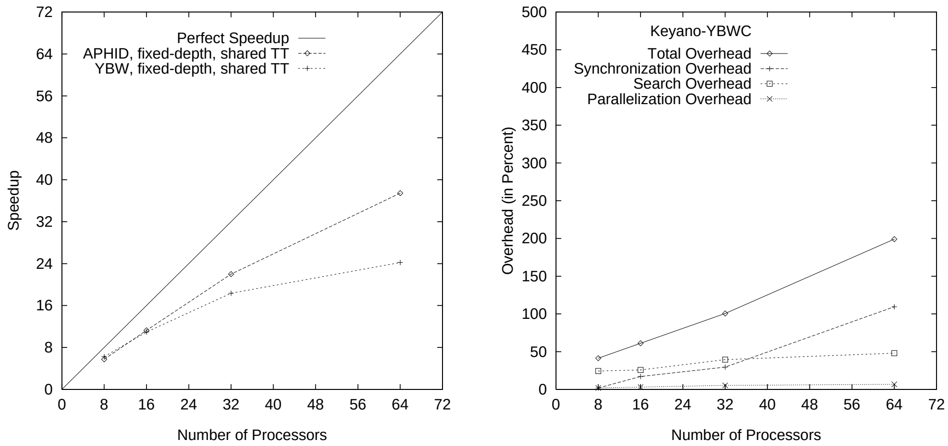

Figure 9: Speedups and Overheads for KEYANO (YBWC, Fixed-Depth, Shared Memory)

never slows down to wait for messages from another processor or to find work.

The observed speedups for YBWC are in line with other published performance numbers for this algorithm. Weill tested YBWC and ABDADA on a CM-5 using a different Othello program [17]. YBWC achieved a 9.5-fold speedup and ABDADA achieved an 11-fold speedup on 16 processors, comparable numbers to that achieved by the KEYANO implementation.

Switching to the checkers program results, the speedups for CHINOOK could be described as disappointing. However, other authors that have attempted to parallelize checkers programs with synchronous algorithms have met with limited success. The best-known speedup for a synchronous algorithm on a high-performance checkers program is 3.32 for Lu's Principal Variation Frontier Splitting with Load Balancing [12]. Thus, the observed speedup of 14.35 on 64 processors for APHID is a four-fold improvement over previously published results for any synchronous algorithm in the domain of checkers.

APHID can yield larger observed speedups than synchronous algorithms in domains with low branching factors such as checkers and Othello. Higher branching factors are represented by two computer chess programs: CRAFTY and THETURK. In computer chess algorithms, synchronous parallel algorithms achieve higher efficiencies because of the increased number of parallel alternatives available at ALL nodes. Consider Weill's tests on a 32-processor CM-5 as representative of synchronous parallel algorithm performance. Weill's implementation of YBWC for chess achieved a speedup of 12 on 32 processors, and ABDADA achieved a speedup of 16 [17]. Weill's results are similar to other reported speedups with synchronous game-tree search algorithms. The majority of these algorithms have been tested under similar conditions to those used for the chess programs in this paper. Although it is not an objective comparison between algorithms, the observed speedups in the literature for most synchronous algorithms are comparable, up to 32 processors, with the observed speedups seen for the APHID algorithm in both THETURK and CRAFTY.

Dynamic Tree Splitting, when using the same testing methodology as described in this paper, achieved a speedup of 8.81 on 16 processors [6]. However, a latter paper which used a non-standard testing methodology (using a series of related positions with transposition table data carried over from search to search) yielded a speedup of 11.1 on 16 processors [7]. It is not known whether the latter testing methodology is partially responsible for the difference in the observed results.

However, there is one important data point that is significantly larger than the reported speedups by any other author. Feldmann's Young Brothers Wait algorithm in ZUGZWANG achieves speedups of 21.83 on 32 processors and 37.34 on 64 processors for 7-ply searches for chess that take place in tournament time [3]. Results of this calibre have not been reported for any other algorithm or by any other author repeating Feldmann's work.

Why are Feldmann's observed speedups so large? One reason is that ZUGZWANG is a slow sequential program. CRAFTY searches 11- and 12-ply trees on 32 processors in the same average time that ZUGZWANG searches 7-ply trees on 32 processors. Although there is a dramatic speed difference between the SGI Origin 2000 (on which Crafty searches 130,000 nodes per second) and the Transputer T-800 (where Zugzwang visits 523 nodes per second), this does not completely account for the large difference in search depths.

Unlike other game-playing programs, ZUGZWANG spends a large amount of time doing move ordering at each node. This allows ZUGZWANG to use a network of Transputers as a shared memory system with very little loss in performance. The time spent in move ordering improves ZUGZWANG's parallel tree search: it searches the best move first 96% of the time, significantly better than most other sequential chess programs. On the other hand, most programs use less ex-

	Keyano		Chinook		Crafty		TheTurk	
n	Speedup	Total Search Over.	Speedup	Total Search Over.	Speedup	Total Search Over.	Speedup	Total Search Over.
8	5.74	9.12%	-	-	-	-	-	-
16	11.27	17.26%	8.35	87.45%	8.76	69.9%	8.48	59.26%
32	21.99	28.78%	10.82	201.6%	16.56	120.7%	12.96	157.2%
64	37.44	76.72%	14.35	393.1%	18.00	300.65%	15.96	384.8%

Table I: Speedup Data for All Programs (Fixed-Depth, Shared Memory)

pensive move ordering heuristics and only search the best more first 80-90% of the time, increasing the node rate dramatically. The increase in nodes per second easily subsumes the additional nodes that must be searched, allowing most programs to out-search sequential ZUGZWANG.

Another reason for the extraordinary speedups is that the transposition table was allowed to grow as more Transputers were added to the system. Thus, the observed speedups are illustrating the power of adding additional processors and transposition table memory, not just additional processors. For the largest search amongst the test set, ZUGZWANG searches 61 million nodes sequentially with a small transposition table, and 41 million nodes in parallel on 256 processors with a much larger transposition table. The number of nodes ZUGZWANG searches sequentially would decrease if much larger transposition tables were used during the sequential test. This would, of course, reduce the observed speedup.

With a fast sequential program, operating in an environment where messages are not fast in relation to the program, less than superb move ordering and constant memory usage in the sequential and parallel runs, it is unlikely that Feldmann's observed speedups can be duplicated with any synchronous parallel algorithm. Given a program without quiescence search, it is possible to generate equivalent speedups with asynchronous search. Earlier it was demonstrated that APHID can generate speedups of that calibre in the fixed-depth version of KEYANO. However, quiescence search is integral to modern chess programs. Hence, it is unlikely that any asynchronous parallel algorithm in chess would achieve observed speedups similar to Feldmann's results.

5 Conclusions

In this paper, the question was posed as to whether asynchronous algorithms could be competitive with synchronous algorithms in real applications searching real game trees. The results in Section 4.3, summarized in Table I, show that the APHID algorithm can deliver superior/competitive results to that achievable by a synchronous algorithm. Further, this performance is easy to achieve since APHID is easy to integrate into an existing $\alpha\beta$-searching program.

Comparing APHID to synchronous approaches, it is interesting to note that APHID's results are less dependent on the branching factor within the tree. The synchronous search results are highly dependent on the branching factor; a larger branching factor yields more parallelism and less idle time. Thus, the asynchronous algorithm in APHID is a better choice in applications with small branching factors, such as checkers and Othello.

The speedups achievable by the APHID algorithm are similar to those achieved by synchronous parallel algorithms up to 32 processors for chess. Thus, a synchronous parallel algorithm may be a reasonable choice for wide variable-depth game trees. However, when one considers that APHID is (1) easy to integrate into existing legacy code, (2) can likely achieve better speedups with more transposition table memory, and (3) has the additional benefit of speculative search on some of the key variations at the next search depth, then APHID becomes the algorithm of choice.

It is hoped that this document, in conjunction with the freely-available APHID library, will be helpful for game-tree researchers to investigate these and other ideas surrounding asynchronous game-tree search.

6 Acknowledgements

Paul Masiar and Marc Nolte arranged access to the 64-processor SGI Origin 2000 system in Eagan, Minnesota. Charles Leiserson, Aske Plaat and Boston University arranged access to a 32-processor SGI Origin 2000 system. The authors would also like to thank Darse Billings, Yngvi Björnsson, Murray Campbell, Joe Culberson, Yaoqing Gao, Andreas Junghanns, Tony Marsland, Monroe

Newborn, Denis Papp, John Samson, and the referees for their helpful suggestions.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of PPOPP '95*, pages 207–216, Santa Barbara, CA, July 1995.

[2] M. G. Brockington. *Asynchrnous Parallel Game-Tree Search*. PhD thesis, University of Alberta, Department of Computing Science, Edmonton, Canada, December 1997.

[3] R. Feldmann. *Spielbaumsuche auf Massiv Parallelen Systemen*. PhD thesis, University of Paderborn, Paderborn, Germany, May 1993. English translation available: Game Tree Search on Massively Parallel Systems.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[5] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker. The Greenblatt Chess Program. In *Proceedings of the Fall Joint Computer Conference*, volume 31, pages 801–810, 1967.

[6] R. M. Hyatt. *A High-Performance Parallel Algorithm To Search Depth-First Game Trees*. PhD thesis, University of Alabama, Birmingham, Alabama, 1988.

[7] R. M. Hyatt. The Dynamic Tree Splitting Parallel Search Algorithm. *ICCA Journal*, 20(1):3–19, 1997.

[8] A. Junghanns and J. Schaeffer. Search Versus Knowledge in Game-Playing Programs Revisited. In *Proceedings of IJCAI-97*, pages 692–697, Nagoya, Japan, August 1997.

[9] D. E. Knuth and R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(3):293–326, 1975.

[10] D. Kopec and I. Bratko. The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 57–72. Permagon Press, 1982.

[11] B. C. Kuszmaul. The StarTech Massively-Parallel Chess Program. *ICCA Journal*, 18(1):3–19, 1995.

[12] C.-P. P. Lu. Parallel Search of Narrow Game Trees. Master's thesis, University of Alberta, Department of Computing Science, Edmonton, Canada, 1993.

[13] T. A. Marsland and M. S. Campbell. Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, 14(4):533–551, 1982.

[14] M. M. Newborn. Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-10(5):687–694, 1988.

[15] J. Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6(2):90–114, 1989.

[16] J. J. Scott. A Chess-Playing Program. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 255–265. Edinburgh University Press, 1969.

[17] J.-C. Weill. *Programmes d'Échecs de Championnat: Architecture Logicielle, Synthèse de Fonctions d'Évaluations, Parallélisme de Recherche*. PhD thesis, Université Paris 8, January 1995. In French.

A The Master's Guessed Score Algorithm

APHID has all of the information about the previous depths of $\alpha\beta$ search for a node, such as whether the score was a lower bound, an upper bound or an exact minimax value. Other information includes the final minimax value from each of those depths of search, as well as a hypothetical minimax value for the current d-ply search (the closest node on the principal variation that has a certain value). All of this information is used to determine a guessed score. The full algorithm is given in Figure 10.

Starting with the information for a $(d - d' - 1)$-ply search, the algorithm checks successively shallower depths of search returned by the slave until a relevant result is found with respect to the $\alpha\beta$ search window. This is guaranteed to happen, since the master stores an exact evaluation of a leaf node (a 0-ply search) when the leaf is first generated and given to a slave.

However, the result returned by the slave should not be used without an adjustment since the minimax value at the root of the tree may vary between different depths of search. In APHID, the slave's minimax value is scaled by the difference between the value of the full search that it was generated for and the hypothetical value of the current search. For example, if the current search has a hypothetical minimax value of 5, and the result of $\leq$ 6 from a search that had a minimax value of 7 at the root of the game tree is to be used, then the difference of the minimax values (5-7) would be added to the result. Thus, the value changes from $\leq$ 6 for the earlier search to $\leq$ 4 for use as a guessed minimax value in the current search. This preserves the move ordering of the previous iteration.

This algorithm is preferred over other methods for determining an estimated evaluation because it guarantees that a guessed score will not disrupt the search tree. If the old search result ($\leq$ 6) was sufficient to cause the tree to be pruned in the earlier search, the guessed value (≤ 4) will likely be sufficient to prune the tree in the current search.

B The Slave's Window Selection Algorithm

The window selection algorithm is application-dependent, and can be modified by editing a call-back function provided to the application programmer. Based on numerous tests, a general set of rules was created for determining small, useful $\alpha\beta$ search windows in APHID.

The first recommendation is that the slave's search window should be centered around the hypothetical minimax value (as defined in Section 3.1), since this is the most likely value of the search.[8] The second recommendation is that to increase the window based on how "unstable" the principal variation may be. For example, if the master says that the highest remaining priority work to be searched is the same as the piece of work that is currently being examined, then a small search window should be chosen because the search is relatively confident of the principal variation. However, as the difference between the largest priority that the master needs completed and the priority of the slave's piece of work gets larger, the search window needs to be larger to reflect the uncertainty in the principal variation. These larger search windows are also used when starting speculative searches for future iterations, since there is no information on the value of the principal variation at the next ply.

These recommendations were used in the window selection algorithm for each of the programs tested in Section 4. The scale and variance of the evaluation function in each program was used to determine the constants used to increase or set the initial size of the slave's window.

[8]When the hypothetical minimax value is INVALID, then this must be a PV node, and an infinite search window should be chosen.

```
int SlaveScoreLowBound(position p, int d);
/* SlaveScoreLowBound returns value from slave’s alphabeta search for
position p at depth d if result was >= alpha(p,d); -INFINITY, otherwise. */

int SlaveScoreUppBound(position p, int d);
/* SlaveScoreUppBound returns value from slave’s alphabeta search for
position p at depth d if result was <= beta(p,d); INFINITY, otherwise. */

int RootMinimax(int d);
/* RootMinimax returns minimax value from previous search at depth d */

int HypotheticalMinimax();
/* HypotheticalMinimax returns highest accurate value along current PV in
current pass, returns INVALID if no PV has been explored (only happens when
first node visited during a pass by a master.) */

int GuessedScore(int alpha, int beta, position p, int d, int dprime)
{
    int depth;  /* depth of search where estimate will be derived         */
    int ub,lb;  /* scaled upper/lower bounds (if not infinite)            */
    int hmm;    /* working hypothetical minimax value for current depth   */

    for (depth = d-1; depth >= dprime; depth = depth - 1)
    {
        hmm = HypotheticalMinimax();
        if (hmm == INVALID) { hmm = RootMinimax(depth); }

        /* Scale lower and upper bounds if they are not INFINITE to change
        bias of search result from previous search depth to current depth. */
        lb = -INFINITY; ub = INFINITY;
        if (SlaveScoreLowBound(p,depth-dprime) != -INFINITY)
            lb = SlaveScoreLowBound(p,depth-dprime) + hmm - RootMinimax(depth);
        if (SlaveScoreUppBound(p,depth-dprime) != INFINITY)
            ub = SlaveScoreUppBound(p,depth-dprime) + hmm - RootMinimax(depth);

        /* Check each bound to see if scaled bounds satisfy constraints. */
        if (ub <= alpha) { return alpha; }
        if (lb >= beta) { return beta; }

        /* An exact scaled value is always satisfactory for constraints. */
        if (lb == ub) { return ub; }

        /* NOTE: Loop will always exit at or before depth = dprime, since
        master stored SlaveScoreLowBound(p,0) = SlaveScoreUppBound(p,0) =
        static evaluation, yielding equal scaled upper/lower bounds. */
    }
}
```

Figure 10: Code For Guessed Score Algorithm

Mark Brockington received a B.Math. degree in Pure Math with Computer Science from the University of Waterloo, in 1992, and received a Ph.D. in Computing Science from the University of Alberta in 1997. His research interests include artificial intelligence, distributed and parallel processing, cryptography and data security. Mark is currently working at BioWare Corp., an interactive entertainment firm in Edmonton, Alberta, Canada.

Jonathan Schaeffer received a B.Sc. degree in Computing Science from the University of Toronto, in 1978, and received an M.Math. and a Ph.D. in Computer Science from the University of Waterloo in 1980 and 1986, respectively. He is the principal author of CHINOOK, the Man-Machine World Champion checkers program. He is currently a Professor of Computing Science at the University of Alberta. His current research interests include heuristic search, parallel processing, and computer games.