University of Alberta

The APHID Parallel $\alpha\beta$ Search Algorithm

by

Mark G. Brockington and Jonathan Schaeffer

Technical Report TR 96–07
August 1996

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

The APHID Parallel $\alpha\beta$ Search Algorithm *

Mark G. Brockington, *brock@cs.ualberta.ca*
Jonathan Schaeffer, *jonathan@cs.ualberta.ca*
Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2H1
Canada

August 20, 1996

Abstract

This paper introduces the APHID (Asynchronous Parallel Hierarchical Iterative Deepening) game-tree search algorithm. An APHID search is a hierarchical search with a master controlling the top of the tree (d' ply), and the slaves searching the rest of the tree ($d - d'$ ply). The slaves asynchronously read work lists from the master and return score information to the master. The master uses the returned score information to generate approximate minimax values, until all of the required score information is available.

APHID has been programmed as an easy to implement, game-independent $\alpha\beta$ library. This has been demonstrated by parallelizing three programs (chess, checkers and Othello) on a network of workstations, each with less than a day's worth of effort.

1 Introduction

The alpha-beta ($\alpha\beta$) minimax tree search algorithm has proven to be a difficult algorithm to parallelize. Although simulations predict excellent parallel performance, many of these results are based on an unreasonable set of assumptions. In practice, knowing where to initiate parallel activity is difficult since the result of searching one node at a branch may obviate the parallel work of the other branches (a so-called *cut-off*). In real-world implementations, such as for high-performance chess, checkers and Othello game-playing programs, the programs suffer from three major sources of parallel inefficiency (a similar model is presented in [18]):

- *Synchronization Overhead*: The search typically has many synchronization points that result in a high percentage of processor idle time.
- *Parallelization Overhead*: This is the overhead of incorporating the parallel algorithm, which includes the handling of communication and maintaining structures to allow for allocation of work.
- *Search Overhead*: Search trees are really directed graphs. Work performed on one processor may be useful to the computations of another processor. If this information is not available, unnecessary search may be done.

These overheads are not independent of each other. For example, increased communication can help reduce the search overhead. Reducing the number of synchronization points can increase the search overhead. In practice, the right balance between these sources of program inefficiency is difficult to find, and one usually performs many experiments to find the right trade-offs to maximize performance.

Many parallel $\alpha\beta$ algorithms have appeared in the literature (see [1] for a comprehensive list of algorithms). The PV-Split algorithm recognized that some nodes exist in the search tree where, having searched the first branch sequentially, the remaining branches can be searched in parallel [17]. Initiating parallelism

*A part of this paper was presented at the Advances in Computer Chess VIII conference in Maastricht, June 1996 [2]. A different portion was accepted for publication at the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96), New Orleans, October 1996 [3].

along the best line of play, the *principal variation*, was effective for a small number of processors, although variations on this scheme seem to be limited to speedups of less than 8 [24].

The idea can be generalized to other nodes in the tree. At nodes where the first branch has been searched and no cut-off occurs, the rest can likely be searched in parallel. It is a trade-off – increased parallelism versus additional search overhead, since one of these parallel tasks could cause a cut-off. This idea has been tried by a number of researchers [6, 7, 10]. The best-known instance of this type of algorithm is called *Young Brothers Wait* (YBW) and was implemented by Feldmann in the ZUGZWANG chess program [5]. Feldmann achieved a 344-fold speedup using YBW on 1024 processors. Variations of this algorithm have appeared with comparable experimental results, such as Kuszmaul's Jamboree search [14] and Weill's ABDADA algorithm [29].

This class of algorithms cannot achieve a linear speedup primarily due to synchronization overhead; the search tree may have thousands of synchronization points and there are numerous occasions where the processes are starved for work. The algorithms have low search overhead, but this is primarily due to the implementation of a globally shared transposition table to share information and improve move ordering.

This paper introduces the Asynchronous Parallel Hierarchical Iterative Deepening (APHID[1]) game-tree search algorithm. The algorithm represents a departure from the approaches used in practice. In contrast to other schemes, APHID defines a frontier (a fixed number of moves away from the root of the search tree), and all nodes at the frontier are done in parallel. Each worker process is assigned an equal number of frontier nodes to search. The workers continually search these nodes deeper and deeper, never having to synchronize with a controlling master process. The master process repeatedly searches to the frontier to get the latest search results. In this way, there is effectively no idle time; search inefficiencies are primarily due to search overhead. APHID's performance does not rely on the implementation of a global shared memory, which makes the algorithm suitable for loosely-coupled architectures (such as a network of workstations), as well as tightly-coupled architectures.

Unlike most parallel $\alpha\beta$ algorithms, APHID is designed to fit into a sequential $\alpha\beta$ structure. APHID has been implemented as a game-independent library of routines. These, combined with application-dependent routines that the user supplies, allow a sequential $\alpha\beta$ program to be easily converted to a parallel $\alpha\beta$ program. Although most parallel $\alpha\beta$ programs take months to develop, the game-independent library allows users to integrate parallelism into their application with only a few hours of work.

This paper discusses the APHID algorithm, its application-independent interface and the performance of the APHID algorithm. The paper is organized into five sections. Section 2 is a brief summary of previous work in sequential and parallel game-tree search. Section 3 is primarily concerned with the details of how the APHID algorithm operates, and how the library integrates with an existing sequential $\alpha\beta$ algorithm. Section 4 describes the results of integrating the library into three different game-playing programs: KEYANO (Othello), THETURK (chess), and CHINOOK (checkers). Section 5 describes some conclusions that can be drawn from the experiments, and a glimpse at some of the things we intend on improving before APHID's general release.

2 Previous Work

This section is divided into two subsections, discussing sequential and parallel search methods.

2.1 Sequential Game-Tree Search

Most common games of thought (such as checkers, chess, Othello and Go) fit into the class of two-player, zero-sum games with perfect information. Given sufficient time and assuming that both players want to win the game, it is easy to determine a best move in any position, and determine whether or not any position is a win, loss or draw.

An initial position in a game, and all of its possible outcomes, can be represented as a game-tree. Each node in the game-tree represents a position within the game. Each arc joining a node at level l to a node at level $l+1$ represents the move required to reach the successor position. The various levels in the tree are called *plies* by game-tree researchers, where ply 0 is the current position in the game, ply 1 consists of all

[1] An aphid is a soft-bodied insect that sucks the sap from plants.

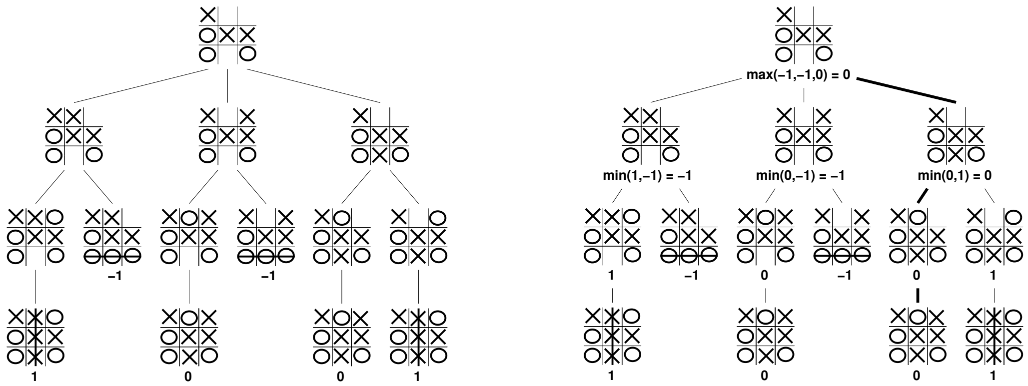

Figure 1: (a) Tic-Tac-Toe Game-Tree, (b) Game-Tree with Minimax Values

successors to the position at ply 0, and so on. At the *terminal positions* in the game-tree, the game is over and a value is used to represent the outcome of the game. For example, a win for the player to move would be represented as 1, a loss as -1, and a draw as 0. The left hand tree in Figure 1 shows a typical game-tree for the game of Tic-Tac-Toe.

The game-tree representation allows us to define a recursive depth-first tree searching algorithm to determine the *minimax value* of each node in the game-tree. The minimax value of a node represents the expected outcome of that position, given best play by both sides. The *minimax algorithm* works by searching the tree in a depth-first manner. We take the minimum of all the values for successor positions at odd-numbered plies in the tree, since this is where the opponent moves. Conversely, we take the maximum of the values of successor positions at even-numbered plies within the game-tree. The right hand tree in Figure 1 shows an example of how to derive the minimax values of each node in the game-tree.

A *principal variation* through a game-tree is a series of moves that yield optimal scores for both players. For example, there is only one principal variation in the game tree given in Figure 1, shown in bold. In general, once the tree is completely searched, an "optimal" move would be the first move from a principal variation.

For games such as chess, some game-trees are so large, the determination of a minimax value for the root is effectively intractable. Thus, a heuristic *evaluation function* is normally used to estimate the potential of winning from a position at a fixed ply in the game-tree. These are the *leaf nodes* of the truncated search tree, and this limit is called the *horizon* of the game-tree.

The *alpha-beta* ($\alpha\beta$) algorithm improves the minimax algorithm by preventing exploration of moves that can provably be demonstrated not to affect the minimax value above the given node. The name is derived from the two bounds: α and β, which are lower and upper bounds on the minimax value we are interested in. The pair of bounds is often referred to as the *search window* for a position, and the $\alpha\beta$ algorithm will return the correct minimax value if the root is searched with the window $\alpha = -\infty$ and $\beta = +\infty$.

An example of how the $\alpha\beta$ algorithm would prune a tree is given in Figure 2. We know that the left subtree has a minimax value of 4, so we set $\alpha = 4$ before we search the right subtree. Note that we set α at even ply, and β at odd ply from the root. After looking at the first leaf of the right subtree, we know that the minimax value of the right subtree will be less than 1. Since $\alpha = 4$, we do not search the other moves in the right subtree.

If we search a tree of depth d which has w moves from any position, there are $\Theta(w^d)$ nodes searched with the minimax algorithm. However, if a "best" move is always searched first, $\alpha\beta$ can reduce this size to $\Theta(w^{d/2})$ [13]. The structure of the *critical tree* evaluated in the optimal case [2] is illustrated in in Figure 3. At `ALL` and `PV` (Principal Variation) nodes, all children are searched, while only one child of each `CUT` node is searched. Note that in practice, you are never certain of the node's type *a priori*.

If a position occurs twice within the tree, we can save the work of exploring the tree underneath the node through the use of a large hash table which stores previously evaluated positions and scores. This hash table

[2] The critical tree is not necessarily the smallest tree that can be searched in practice, since transpositions are not taken into account [21].

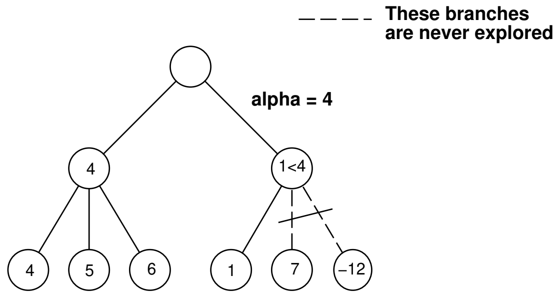

Figure 2: Example of an $\alpha\beta$ Cut-off

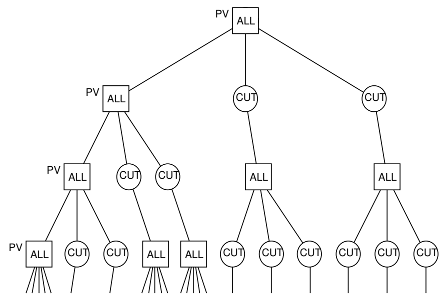

Figure 3: Structure of the Critical Game-Tree

is known as a *transposition table*.

Most competitive game-tree programs use *iterative deepening* to search the tree [26]. After a k-ply search, a $k+1$-ply search is executed on the same position. Although this seems wasteful, a transposition table and other move ordering techniques can often provide a very accurate guess at what the best move is, and the tree searched is close to the critical tree in size. Experience shows that the benefits gained by a k-ply tree as a prelude to searching a $k+1$-ply tree outweigh the costs [27].

2.2 Parallel Game-Tree Search

The idea behind the PV-Split algorithm has proven to be a fundamental building block in developing high-performance parallel game-tree algorithms [17]. Simply stated, the first move at a principal variation node must be completely evaluated before the subsequent moves can be handed out to other processors and evaluated in parallel. Parallelism occurs only at the `PV` nodes, and the nature of the algorithm ensures that an accurate search window is determined before allocating work to the slaves in parallel, which reduces search overhead. Although it is easy to control the PV-Split algorithm since only one `PV` node can be evaluated in parallel at a given moment in time, a different approach is needed if you have more processors than moves at the current `PV` node.

Newborn's UIDPABS algorithm [19] 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 window, but some of the processors may have changed their windows, based on the search results of their moves. The UIDPABS algorithm then combines the results once a predetermined time limit has been reached. The APHID algorithm uses the basic concept of how to implement asynchronous search from UIDPABS, as we shall see in Section 3.

Hsu's family of algorithms [9] are similar in design to the APHID algorithm. A host processor is responsible for maintaining a queue of critical tree and non-critical tree work to do for a d' ply search tree. The special-purpose processors are responsible for the $d-d'$ ply at the bottom of the tree, by continually requesting pieces of work from the two queues. As the work is completed and sent back to the main processor, the tree is incrementally updated up from the leaf node, and new work is added to the non-critical tree list, as necessary. Hsu has not published results with the version of Deep Blue that played against Kasparov in February 1996. The simulation results presented in his thesis predict a speedup of 350 on 1000 processors although these results have not been borne out in practice [8].

Although APHID is similar to Hsu's family of algorithms, there are important differences between the two approaches. It is important to note that the d' ply critical tree is only generated once, and then incrementally changed as results come in from the processors. Furthermore, although iterative deepening is used to accomplish the $d - d'$ ply search on the specialized processors, there is no concept of ownership of a subtree by a processor in Hsu's algorithm. Hsu's original design did not include a concept of a multi-tiered hierarchy of processors. Hsu's work priority scheme is slightly different than the one implemented in APHID. Finally, Hsu's algorithm gives no speculative work to the specialized processors in when the two work queues are empty.

The Young Brothers Wait (YBW) algorithm extends PV-Split to state that the other moves (the "young brothers") can be searched in parallel only if the first move of a node has been completely searched and has not caused the $\alpha\beta$ search to be pruned [5]. Assuming we start with an infinite search window at the root position, this is always true at `PV` nodes, and is generally true at `ALL` nodes. Thus, there are multiple potential parallel nodes at any given time when searching the tree. However, the search is still synchronized in the same way that PV-Split is synchronized. Until a search of all children of a given `PV` node is completed, the other children of the `PV` node's parent cannot be searched.

Although the *synchronization overhead* in YBW is a lot smaller than in PV-Split, workers still search for a processor that has work to do (according to the YBW criterion) by sending a message to a processor at random. This dynamic load-balancing method, "work-stealing" [14], is effective in balancing the share of work done on each processor, but periodically imposes a heavy communication load.

In the implementation of ZUGZWANG presented in Feldmann's thesis [5], a distributed transposition table was implemented with message passing across a series of Transputers to improve the results of the algorithm. On a system where only a small number of nodes can be processed per average message latency ratio, this type of distributed transposition table is practical and is useful in controlling the search overhead.

An alternative to a shared transposition table is the idea of recursive iterative deepening to achieve a better move ordering [14]. The additional search overhead of recursive iterative deepening in a work-stealing algorithm with only local transposition tables is strongly dependent on the branching factor of the trees being searched. Although he did not invent the idea [3], Kuszmaul achieved reasonable success with recursive iterative deepening in the StarTech chess program, both in the sequential program with a local transposition table and the parallel program with a shared transposition table. This is partially due to the test set being used: Kaufman's test set of tactical positions [11, 12] yields positions which have rapidly changing principal variations, and is more susceptible to showing the advantages of recursive iterative deepening. It remains to be shown whether recursive iterative deepening or a shared transposition table by themselves is a more effective heuristic in speeding up a work-stealing algorithm on a "fair" test set.

David's $\alpha\beta*$ framework uses a global transposition table to control where the processors should be searching [4]. By adding a field to the global transposition table to indicate the number of processors searching that node, each processor can pretend it is searching the tree sequentially, and make decisions on where to search based on the number of processors searching the children of the node. When any processor generates a value for the root position, the search is finished. Unfortunately, David's method of controlling where the processors should be searching was inefficient, and the scheme was hampered by the use of half of the Transputers as transposition table storage units, limiting the speedup reported to 6.5 on 16 Transputers. Regrettably, no work is reported that addresses these shortcomings.

Weill [29] recognized that the YBW criterion could be used in conjunction with the $\alpha\beta*$ framework. Weill showed the combination, ABDADA, yields comparable performance to a YBW implementation on a CM-5. On 16 processors, ABDADA yielded an 10-fold speedup for a chess program, while YBW generated a speedup of just under 8.

Unfortunately, neither of these scheduling methods deal adequately with architectures that have can process a high number of nodes per average message latency, such as a network of workstations. Using YBW on a system with only transposition tables local to each process will yield large search overheads, since there is no guarantee of where a given node will end up when we use the chaotic work-stealing scheduler

[3] Kuszmaul states that both Truscott and Berliner have used recursive iterative deepening in the past.

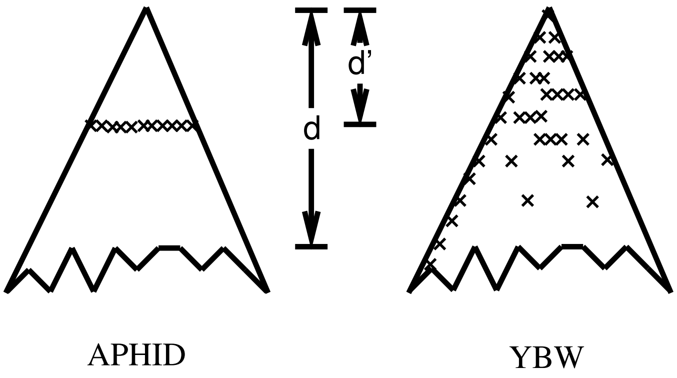

Figure 4: Location of Parallelism in Typical APHID and YBW Search

in combination with iterative deepening. Using ABDADA is infeasible since the system requires a shared transposition table, which would be extremely slow on a parallel architecture with a high number of CPU cycles per average message latency.

3 The APHID Algorithm

Young Brothers Wait and other parallel search algorithms suffer from three serious problems. First, the numerous synchronization points result in idle time. This suggests that a new algorithm must strive to reduce or eliminate synchronization altogether. Second, the chaotic nature of a work-stealing scheduler requires algorithms such as YBW and Jamboree to use a shared transposition table and/or recursive iterative deepening to achieve a good move ordering and reasonable performance. Algorithms based on the $\alpha\beta*$ framework cannot work without a shared transposition table. Third, the program may 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, Young Brothers Wait (in its simplest form) permits all of the remaining branches to be searched in parallel. However, if the second branch causes a cut-off, then all the parallel work done on the third (and subsequent) branches has been wasted. This suggests parallelism should only be initiated at nodes where there is a very high probability that all branches must be considered.

This section introduces the Asynchronous Parallel Hierarchical Iterative Deepening (APHID) game-tree searching algorithm. APHID has been designed to address the above three issues. The algorithm is asynchronous in nature; it removes all synchronization points from the $\alpha\beta$ search and from iterative deepening. Also, parallelism is only applied at nodes that have a high probability of needing parallelism. The top plies of a game-tree (near the root) vary infrequently between steps of iterative deepening [22]. This relative invariance of the top portion of the game-tree is exploited by the APHID algorithm.

In its simplest form, APHID can be viewed as a master/slave program although, as discussed later, it can be generalized to a hierarchical processor tree. For a depth d search, the master is responsible for the top d' ply of the tree, and the remaining $d - d'$ ply are searched in parallel by the slaves. Figure 4 shows where parallel activities occur in APHID and YBW. Each location marked with an `x` shows where the parallelism typically takes place. Although more parallelism could be generated in YBW, each `x` represents a potentially costly synchronization point. The parallelism is more constrained in APHID and, hence, is more likely to suffer from load imbalances than other dynamic scheduling routines (such as YBW, Jamboree, or $\alpha\beta*$).

3.1 Operation of the Master in APHID

The master is responsible for searching the top d' ply of the tree. It repeatedly traverses this tree until the correct minimax value has been determined. The master is executing a normal $\alpha\beta$ search, with the exception that APHID enforces an artificial search horizon at d' ply from the root.

Each leaf node in the master's d' ply game-tree is being asynchronously searched by the slaves. Before describing how the master knows when the d ply search is complete, we must first describe how the master searches the d' ply tree.

When the master reaches a leaf of the d' ply tree, it uses a reliable or approximate value for the leaf, depending on the information available. If a $d - d'$ ply search result is available from the slave, that will be used. (In the current implementation, we do not use deeper ply values, even if they are available. This will be discussed in Section 4.) However, if the $d - d'$ ply result is not available, then the algorithm uses the "best available" ply result that had been returned by the slave to generate a guessed minimax value[4]. Any node where we are forced to guess is marked as *uncertain*.

As values get backed up the tree, the master maintains a count of how many uncertain nodes have been visited in a pass of the tree. As long as the score at any of the leaves is uncertain, the master must do another pass of the tree. Once the master has a reliable value for all the leaves in its d' ply tree, the search of the d ply tree is complete. The controlling program would then proceed to the next iteration by incrementing d and asking the master to search the tree again.

Note that this definition of the master solves one of the problems that some algorithms have with respect to initializing parallelism too quickly at a potential `CUT` node. 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. If it seems likely that a child will generate a cut-off based on guessed values, the children of the failed `CUT` node are evaluated sequentially. If it seems unlikely that the node will be pruned due to low minimax values, the search would continue for a promising node at that branch in parallel. This is all handled automatically by the $\alpha\beta$ routine. The handling of a hypothesized `CUT` node is stronger than the equivalent scenario in the YBW algorithm, which ignores previous score information available for some branches of the failed `CUT` node. In the full version of YBW, application-dependent information is used to do what APHID handles automatically with the $\alpha\beta$ search window.

The slaves are responsible for setting their own search windows, based on information from the master. Sometimes, the information returned by the slave may not be useful to the master. For example, a slave can tell the master that the score of a given node is less than 30, but the master may want to know if the score is in between -5 and 5. In this case, a "bad bound" search is generated, and the search window parameters, α and β, must be communicated to the slave processor. Any nodes where we are waiting for "bad bound" information to be updated by the slave are considered as uncertain by the master. Eventually, the slave will return updated information that is consistent with both the original information and the search window requested[5].

3.2 The APHID Table

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

The APHID table is 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 slave or the master process of the update to the information. The master and slave only read their local copies of the information; there are no explicit messages sent between the master and the slave asking for information.

The master's half of the table is illustrated above the dotted line in Figure 5. For each leaf that has been 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 allocated to.

In our example, we can see that 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 has been visited on a previous pass of the d' ply search tree, and was not visited on the latest pass. However,

[4]Many game-tree search programs exhibit an effect based on the parity of the search depth (odd or even number of ply). Scores are stable when you look at results from the odd plies only, or even plies only, but are sometimes unstable when you mix the two. Thus, we use the deepest ply value with the same parity, instead of always using the deepest ply value available.

[5]It may happen that the original search and the "bad bound" search are inconsistent with one another, through the use of search extensions that may or may not be triggered based on the search window used. In this case, the search explicitly requested by the master overrides the information that had been previously stored.

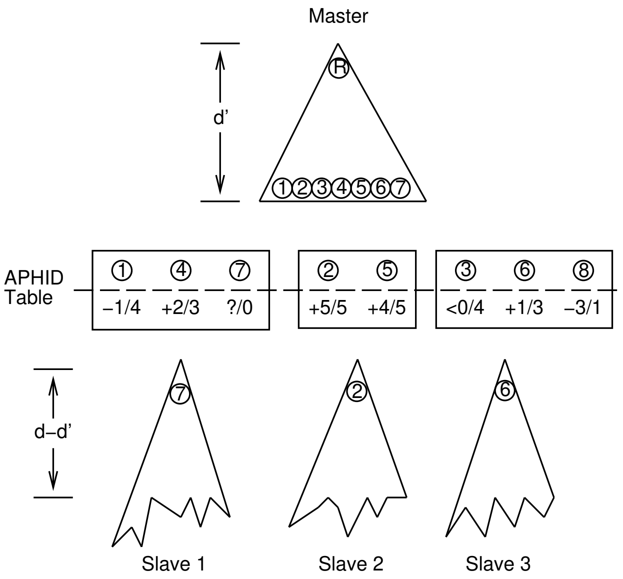

Figure 5: A Snapshot of APHID Search in Operation

the information that the slave has generated may be needed in a later pass of the tree and is not deleted by the master. 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). Although there may be better methods of allocating leaves, it has been found that this is a reasonable method of initially balancing the load on a small number of processors.

The slave's part of the table, illustrated by the area below the dotted line, contains information on the result of searching the position to various depths of search. The "best" information and the ply to which the leaf was examined is given underneath each leaf node in the tree. 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 is not necessarily an exact number. The slaves maintain an upper bound and a lower bound on the score for each ply of search depth. The score is known to be exact when the upper and lower bounds are the same.

3.3 Operation of Slave in APHID

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

1. Look in its portion of its local copy of the APHID table, and find the highest priority node to search.
2. Execute the search.
3. Report the result back to the master (getting an update to its APHID table in return).

The work selection criterion is primarily based on the depth to which the slave has already searched a node. As we can see for Slave 1 in Figure 5, leaves 1, 4 and 7 have been searched to 4, 3 and 0 ply, respectively. Thus, Slave 1 is attempting to search leaf 7 to 1 ply, and will continue to search leaf 7 up to 3 ply by using iterative deepening, if no new work arrives from the master.

The secondary criterion is the location of the node within the master's game-tree. This secondary criterion is necessary since it is usually beneficial to generate the results in a left-to-right order for the master. Children of nodes are usually considered in a best-to-worst ordering, implying that the left-most branches at a node have a higher probability of being useful 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 in the slave to 6 ply since it is further left in the tree than leaf 5.

A node that has a priority of zero (because it is no longer part of the master's tree) will not be selected for further search. For Slave 3, we notice that Leaf 8 would be searched if it had been visited by the master on the latest pass. Leaf 8 is ignored by the scheduling algorithm because it is not currently part of the master's tree.

Before a search can be executed, an $\alpha\beta$ search window must be generated by the slave. The master continually advises the slaves of the leaf's location within the master's tree, and the hypothetical value of the root of the master's tree. Although the width of the search window is application-dependent, one normally wants to center the window around this hypothesized root value, plus or minus a factor to reflect the uncertainty in it.

There are three types of update messages that a slave receives from the master: a new piece of work has been added to the slave processor's APHID table, the location of a leaf node within the master's tree has changed (changing the secondary work scheduling criterion), and a notification of a "bad bound" on a node. The bad bound message alerts the slave that a position's search information is not sufficient to save the node from being uncertain. In this case, the slave must re-search the node with the $\alpha\beta$ search window sent by the master to the ply requested.

As a performance improvement, we want to force the slave to always work on nodes for the current search depth of the master. When all the slave's work has been searched to the required depth, rather than becoming idle, it starts re-searching its work speculatively an additional ply deeper, in anticipation of the next iteration (depth $d + 1$). When speculative search is running on a processor, the slave routinely checks the communication channel for messages from the master. If the slave receives a new piece of work to do at $d - d'$ ply or less, the speculative search is immediately aborted and control is returned to the slave's scheduling algorithm.

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. In games such as chess, there are no completely reliable indicators of the "effort" required for a given search. Thus, load imbalances can occur based on the allocation of work to slaves.

The master knows all of the information about how many uncertain nodes it is waiting for from each slave. Thus, the master has information on when to move leaves from the d' ply tree from an *overworked slave* (a slave with a large number of uncertain nodes) to an *underworked slave* (a slave with no uncertain nodes). This yields a tradeoff between faster convergence for a given ply search of the tree and additional search overhead.

3.5 Implementation

The APHID algorithm has been written as an application-independent library of C routines. The library was written to provide minimal intervention into a working version of sequential $\alpha\beta$ (or its common variants: NegaScout[20, 23] and Principal Variation Search (PVS) [17]). Since the library is application-independent, a potential user must write a few application-dependent routines (such as move format, how to make/unmake moves, position format, setting a window for a slave's search, *etc.*). APHID's message passing was written using PVM [28] to allow for the maximum portability among available hardware.

To parallelize a sequential $\alpha\beta$ program, the user modifies his or her search routine as shown in Figure 6. The APHID changes are marked by shading, and easily fit into standard $\alpha\beta$ frameworks. This one piece of code functions as the search algorithm for both the master and the slave processes.

There are a few additional calls that have to be added to the iterative deepening routine that calls $\alpha\beta$, outlined in Figure 7.

Initially, it was anticipated that all users would want to search in parallel from the root of the game tree. However, there are some programs that wish to handle the root of the game tree in a different way than the other leaves of the search tree (by adding calls to the time-control mechanism, and special handling of the $\alpha\beta$ search window). APHID has been generalized to integrate with this style of searching the game tree; Figure 8 illustrates the changes necessary. The only significant change from Figure 7 is the addition of a call to `aphid_intnode_premove`.

There are two calls that are inserted into the main program, which are illustrated in Figure 9.

```
int PVS(p, alpha, beta, depth, plytogo)
position p;
int alpha, beta;
int depth, plytogo;
{
     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 for h_length ply              */
     int h_flag;    /* VALID, LBOUND or UBOUND             */
     move *h_move;  /* recommended move for h_length ply   */
     int width;     /* number of moves in move list */
     int i;         /* move counter                 */
     int value;     /* score of child PVS call      */

     /* Generate hash value and key for this position */
     generate_hash(p, p_hash, p_key);
     /* Fetch information from local transposition table */
     retrieve(p_hash, p_kley, 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 the tree */
     if (plytogo <= 0) { return(evaluate(p)); }
     if (aphid_horizon(depth)) {
         return(aphid_eval_leaf(alpha,beta,depth,p_hash,p_key));
     }

     /* Generate move list, evaluate position if no moves */
     width = generate(p, h_move);
     if (width == 0) { return(evaluate(p)); }
     if (aphid_checkalarm() != FALSE) {
         terminate_search = TRUE;
         return(0);  /* Should exit PVS quickly when alarm on */
     }

     score = -INFINITY;
     lower = alpha;  upper = beta;
     aphid_intnode_start(depth, p_hash, p_key);

     /* Loop through moves in move list */
     for(i=1; (i <= width && score <= beta); i++ ) {
         aphid_intnode_move(depth, &(move[i]));
         make_move(p, move[i]);

         value = -PVS(p, -upper, -lower, depth+1, plytogo-1);
         if (value > score && i > 1) {
             value = -PVS(p, -beta, -value, depth+1, plytogo-1);
         }

         unmake_move(p,move[i]);
         aphid_intnode_update(depth, value);
         if (value > score) {
             score = value;
             move_opt = move[i];
         }

         /* Set bounds for next search */
         lower = max(alpha, score);  upper = lower+1;
     } /* for all moves */

     if (score >= beta) { aphid_intnode_fixbound(depth); }

     /* Write information into local trans. table */
     aphid_intnode_end(depth, &score);
     h_flag = VALID;
     if (score <= alpha) { h_flag = UBOUND; }
     if (score >= beta)  { h_flag = LBOUND; }
     if (h_length <= plytogo) {
         store(p_hash, p_key, plytogo, score, h_flag, move_opt);
     }
     return(score);
 } /* function PVS */
```

Figure 6: How APHID Modifies a Typical PVS Implementation

```
aphid_initsearch(MAXDEPTH);
for(plytogo=1;(plytogo <= MAXDEPTH && done == FALSE); plytogo++) {
    /* Set up search */
    /* Search at root around value (guess) with small error (eps) */
    /* Call to aphid_rootsearch replaces call to PVS */
    score = aphid_rootsearch(0,plytogo,guess-eps,guess+eps);
    /* Print out results of search */
}
aphid_endsearch();
```

Figure 7: How APHID Modifies the Iterative Deepening Routine

```
aphid_initsearch(MAXDEPTH);
for(plytogo=1;(plytogo <= MAXDEPTH && done == FALSE); plytogo++) {
    /* Set up search with small window (eps) around guess */
    score = ROOT_PVS(root_pos,guess-eps,guess+eps,0,plytogo);
    /* Print out results of search */
}
aphid_endsearch();

ROOT_PVS(root_pos, alpha, beta, depth, plytogo)
...
{
    ...
    /* Search PV Move */
    lower = alpha;

search_best_move:
    aphid_intnode_premove(depth, &bestmove);
    make_move(p,bestmove);
    /* aphid_rootsearch replaces typical call to PVS */
    oldscore = -aphid_rootsearch(depth+1,plytogo-1,-beta,-lower);
    unmake_move(p,bestmove);

    /* Search other moves at root, and only switch if move beats */
    /* PV score (oldscore) by a small margin (delta)             */

    newscore = oldscore;
    for(i=2;(i<=width && newscore <= oldscore+delta; i++) {
        aphid_intnode_premove(depth,&move[i]);
        make_move(p,move[i]);
        /* Check if move beats PV move by more than delta */
        newscore = -aphid_rootsearch(depth+1, plytogo-1,
                                     -oldscore-delta-1, -oldscore-delta);
        unmake_move(p,move[i]);
    }

    if (newscore > oldscore+delta) {
        /* set new best move and score and research (if necc.) */
        bestmove = move[i];
        lower = newscore;
        if (newscore < beta) { goto search_best_move; }
    }
    ...
} /* function ROOT_PVS */
```

Figure 8: How APHID Modifies Special Handling of the Root of the Game Tree

```
int main(argc, argv)
int argc;
char *argv[];
{
     /* Initialization required by any process in system */

     aphid_startup(argv);
     /* Only the absolute master process gets here */

     /* Initialization required only by the master process */
     /* Play game */

     aphid_exit();
     exit(0);
}
```

Figure 9: How APHID Modifies the Main Program

A brief explanation of the parameters and function of each of these `aphid_` routines can be found in Appendix A.

Along with the additional calls added to PVS, the iterative deepening routine, and the main program, roughly 100 lines of application-dependent code have to be defined for the `aphid_stub_` routines. The APHID library is game-independent, but it needs to know some game-dependent properties such as what the name of the evaluation function is, what the name of the search routine is, how to make and unmake moves, *etc.* The `aphid_stub_` routines are also briefly described in Appendix A.

4 Experiments

The APHID game-independent library has been inserted into three different programs over the last six months. For this initial experiment, each of the programs chosen were written at the University of Alberta, and the authors of the program were assisted in implementing the APHID library into their programs. The first program was KEYANO, an Othello program written by Mark Brockington. The second program was THETURK, a chess program written by Yngvi Bjornsson and Andreas Junghanns. The final program was the current Man-Machine world champion checkers program, CHINOOK, written by a team that includes Martin Bryant, Rob Lake, Paul Lu, Jonathan Schaeffer, and Norman Treloar [25].

Parallel tests were run on up to 16 workstations on a network of SparcStation IPC computers with 12 MB of RAM, running the SunOS 4.1.4 operating system. The SparcStations are linked with 1 segment of 10 base 2 (thin net) Ethernet. One workstation in each experiment was completely occupied by the master process, while the other workstations each ran a single slave process.

More specific details will follow in the subsequent sections about the experiments with each program. However, we must discuss the general methodology used for evaluating the performance of the APHID algorithm.

Parallel and sequential algorithms often do not agree with each other about minimax values and best moves when the full version of the program is used [15]. For example, different search windows cause different search extensions to be turned on, causing different alpha-beta results. Thus, all search extensions, search reductions and null move searching were turned off for the purposes of this experiment. Since some work could be evaluated to $d+1$-ply before the d-ply result is finished, we also forced the transposition table to report only transposition table scores that had been searched to the exact depth (as in Figure 6). Although a fixed depth is enforced on the programs, quiescence search was left in THETURK and CHINOOK to prevent the evaluations from being significantly unstable. This forced the parallel and sequential programs to return identical minimax values, allowing for a fair comparison.

A suitable benchmark set was chosen for each program (see appropriate section). Each game also has different time constraints in typical positions. For example, an Othello program should take an average of 90 seconds to make a move for a midgame search, and a chess program should take about 150 seconds to complete a search. The search depth was chosen so that the average of the parallel results on 16 processors did not exceed this time constraint.

The *speedup* given in this section compares the fixed depth version of the sequential program versus the time required to find the result in parallel:

$$\text{speedup} = \frac{\text{sequential time}}{\text{parallel time}}.$$

It is important to note that neither the sequential nor the parallel algorithm being tested is the one that would be used under tournament conditions, because the search extension/reduction algorithms were turned off. For the purposes of the experiment, the parallel and sequential times for each search are summed together to determine the aggregate speedup.

The aggregate speedups will be illustrated in two graphs. The first graph will show the breakdown of the aggregate speedup at the point where the depth d search has been completed. The second graph divides the test set into three or four disjoint subsets (based on size of the sequential search), and gives the aggregate speedup for each subset. For every program tested, both graphs illustrate that larger searches yield greater parallelism than smaller searches. The second graph also illustrates some of the variance to be found in the aggregate speedup.

The overheads in the algorithm will be illustrated in a third graph. The third graph represents the same scale across all programs. For KEYANO and THETURK, a fourth graph is given to illustrate the same data on a magnified scale.

The *total overhead* represents the additional computing time required by the parallel algorithm to achieve the same result:

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

where n is the number of processors. The total overhead can also be computed by examining the overheads. The three main overheads are using a processor exclusively as a master, the effective decrease in nodes per second examined, and the additional number of nodes searched by the parallel algorithm. There is no synchronization overhead in the APHID algorithm, since the algorithm operates in an asynchronous manner. This can be expressed in the following formula[6] for total overhead:

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

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 assumed to be the parallelization overhead. This parallelization overhead is derived partially from the overhead of using PVM, and partially from the work-scheduling algorithm on each slave. In effect this includes synchronization overhead, complexity overhead and communication overhead, as used in previous parallel $\alpha\beta$ models [15, 18, 24].

The *search overhead* represents the additional nodes searched to achieve the d ply minimax value. This can be computed by dividing the nodes searched to generate d ply search results in the parallel program by the nodes searched by the sequential program. Most of the search overhead is incurred by attempting to do searches before the correct search window is available. Thus, the slaves use $\alpha\beta$ search windows that are larger than those in the sequential program. Most of the increase in search overhead as we increase the number of processors can be attributed to information deficiency, since there is no common shared data between the processes (such as a shared transposition table).

Since we only search each position to d ply, the asynchronous nature of the slaves will result in some work being done at $d+1$ ply (or more). The *speculative search* represents the amount of additional search beyond what the sequential algorithm would have done. The speculative search 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, the speculative search results were not used so that the parallel program produces the identical results as the sequential version, verifying APHID's correctness. In a real tournament game, this speculative search could be used to look an extra move ahead on some key variations, since it is highly likely that the moves extended a ply ahead would be in the left-most branches of the tree. Note that other algorithms, such as Young Brothers Wait, have processors go idle when there is no work left to do on the current iteration.

4.1 Keyano (February 1996)

The first program that the APHID library was implemented in was KEYANO, an Othello program which has competed in international Othello tournaments for the last three years.

APHID did not have load balancing at the time the algorithm was originally benchmarked; the load balancing was introduced to the evolving APHID library in March. To be consistent with results presented elsewhere [3], the February 1996 results are given here.

To test the algorithm, Keyano was programmed to search with its midgame search algorithm to a depth of $d = 12$ ply, with the master controlling the top $d' = 4$ ply of the tree.

[6]This formula is not the same as the formula presented in earlier versions of this paper.

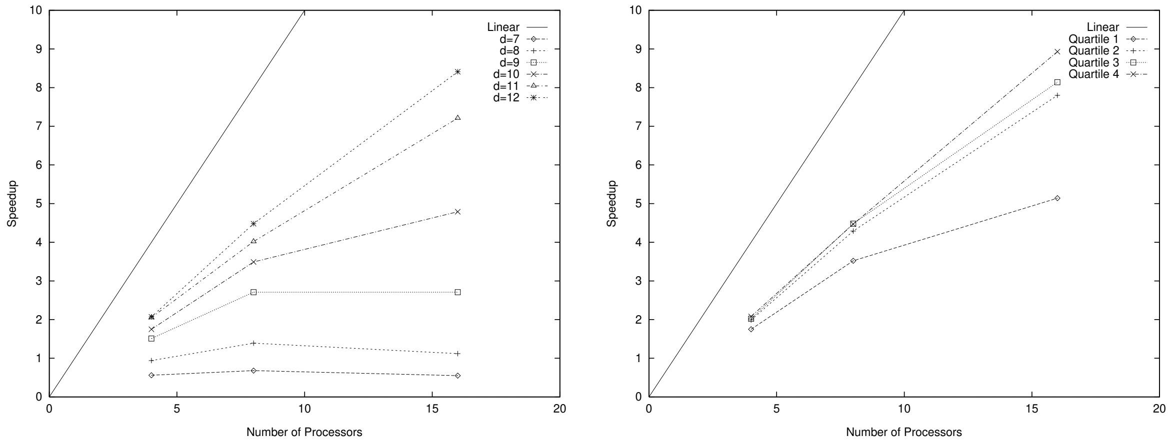

Figure 10: KEYANO - Speedups by Depth of Search and Sequential Tree Size

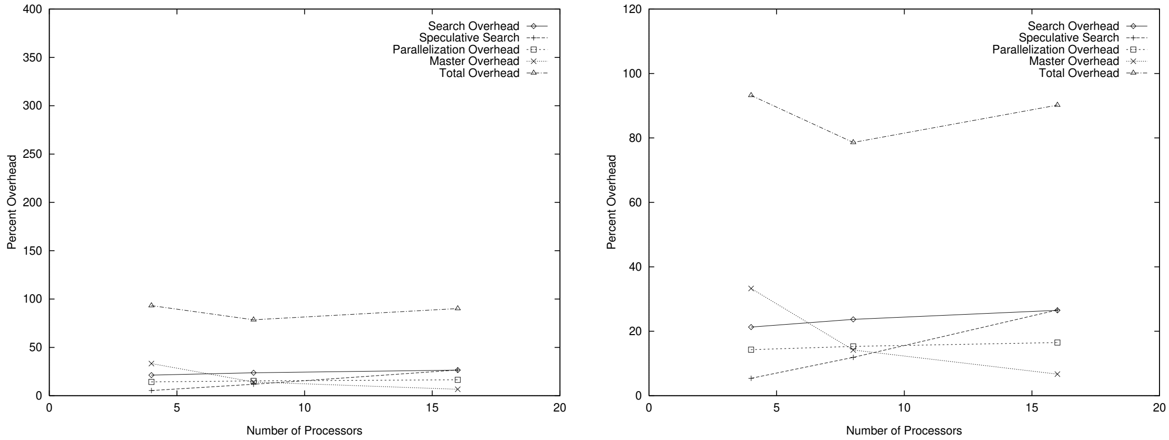

Figure 11: KEYANO - Overheads and Magnified View of Overheads

The 74 positions examined were the positions from move 2 to move 38 in the two games of the 1994 World Championship final between David Shaman and Emmanuel Caspard [7].

In Figure 10, we see the graph on the left contains the speedups broken down by ply. The graph on the right contains the 12 ply results broken down into four quartiles. The first quartile represents the smallest sequential searches, while the fourth quartile represents the largest sequential searches.

In Figure 11, we see the relative size of each of the overheads described earlier. The results for Keyano are, in many ways, the most encouraging of the three programs tested.

There is a low search overhead which increases slightly as we go from 4 to 16 processors. The parallelization overhead is not large. Keyano went from investigating 8700 nodes per second in the sequential program to investigating between 7400 to 7600 nodes per second in the APHID version of the program. The only overhead that increases rapidly is the speculative search overhead, but this is not a concern during a tournament game (as previously stated).

Weill tested YBW and ABDADA on a CM-5 using a different Othello program [29]. YBW achieved a 9.5-fold speedup and ABDADA achieved a 11-fold speedup on 16 processors. Although the APHID results are not as good, they were achieved without a shared transposition table. Both ABDADA and YBW will get poor performance on a network of workstations, since the shared transposition table is critical to effective move-ordering and limiting the search overhead.

[7] Beyond move 38, Keyano would attempt endgame searches before completing a 12-ply sequential search. Move 1 in an Othello game is completely symmetric, and Keyano does not search for the first move in the game.

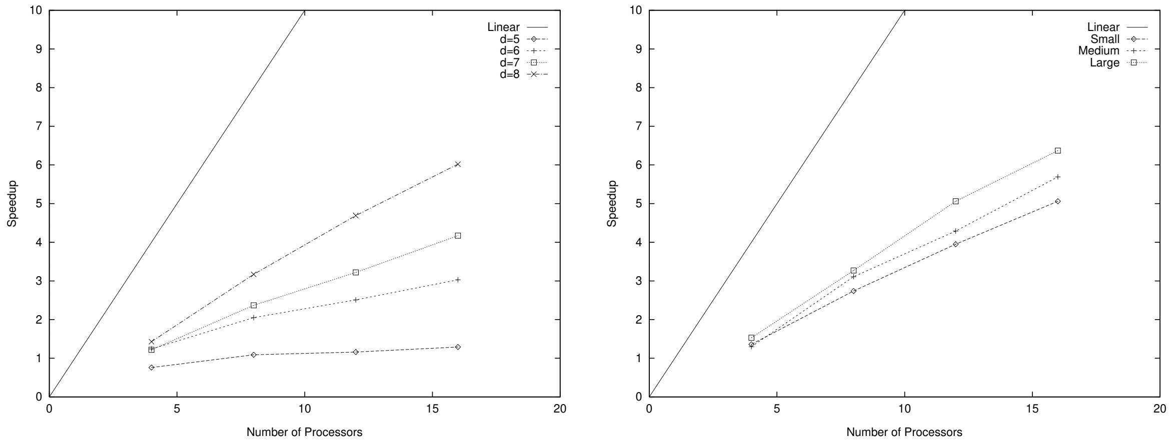

Figure 12: THETURK - Speedups by Depth of Search and Sequential Tree Size

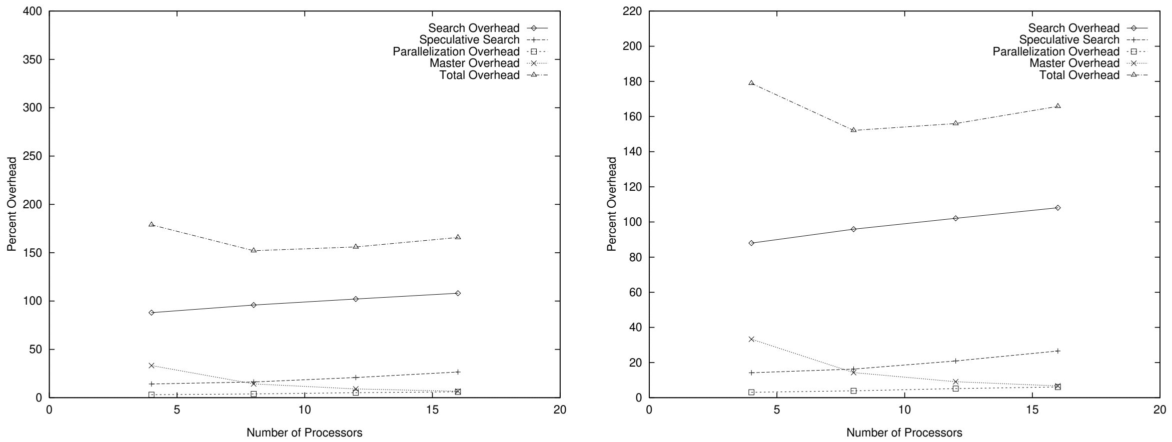

Figure 13: THETURK - Overheads and Magnified View of Overheads

4.2 TheTurk (April 1996)

The experiments with THETURK, the chess program by Andreas Junghanns and Yngvi Bjornsson, were run soon after the load balancing was inserted into the APHID library. Although it was unclear whether load balancing would be necessary with the APHID library when testing the program with KEYANO, the necessity for load balancing in other games such as chess was clear to the authors.

To test the program, the 24 Bratko-Kopec positions were used. Although there are known problems with these positions, using this test set allows for comparison with other work. Each position was searched to depth $d = 8$, while the master controlled the top $d' = 3$ ply of the tree, since $d' = 4$ was not feasible given the amount of memory on the SparcStations. Figure 12 represents the speedups generated by THETURK, while Figure 13 gives the overheads in the parallel program.

Although the results presented here for chess are promising, we believe that better results can be achieved. The main difference between the results for KEYANO and THETURK revolve around the size of the search overhead.

We have come up with three possible hypotheses for this difference. The first is that KEYANO does not use quiescent search. As a consequence, the sizes of the various pieces of work are more predictable in size, and Keyano's results did not need load balancing. Load balancing causes search overhead since it forces a new processor to repeat searches done by a different processor. The load balancing is partially responsible for the difference in search overhead. The second possible reason is that the search windows in the slaves can, in some cases, be larger than the minimal window used in the sequential program. Although this larger window is common to both parallel implementations, it has a much larger effect on the size of the trees searched in

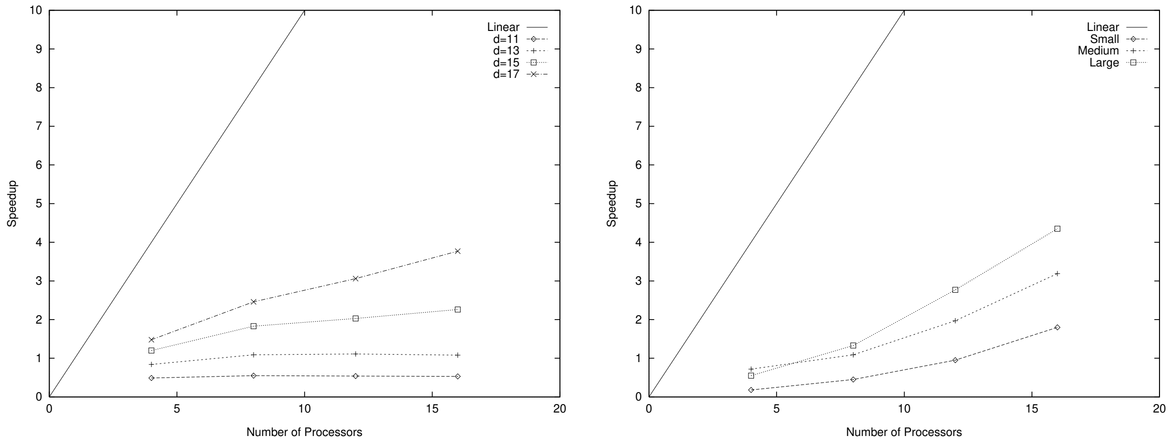

Figure 14: CHINOOK - Speedups by Depth of Search and Sequential Tree Size

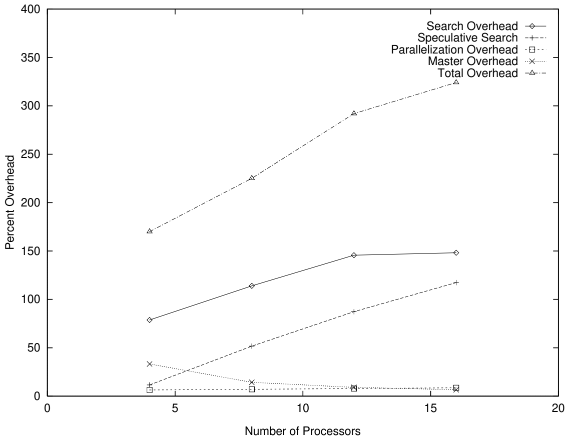

Figure 15: CHINOOK - Overheads

THETURK than in KEYANO. The third hypothesis is that the game of chess has more transpositions than the game of Othello. Thus, the information deficiency caused by doing the search on multiple processors is greater in chess than in Othello.

Other results for parallel search algorithms on a network of workstations have been presented for the game of chess [24]. A distributed transposition table was used to improve the performance of the PARAPHOENIX chess searches, and a speedup of 7 was achieved on 16 processors. Although these results are marginally better than the results presented here, the scalability of the algorithm was extrapolated to a speedup of at most 8 on 32 processors. Based on the graphs presented earlier in this section, we believe the APHID algorithm will outperform on a larger number of processors. The beneficial effects of the distributed transposition table are derived primarily in the top plies of the search tree, and these benefits are duplicated in APHID by maintaining the top d' ply exclusively on the master processor.

4.3 Chinook (June 1996)

Due to resource limitations, we could not run the experiments with CHINOOK until late June. The benchmark positions used are a series of positions which CHINOOK has misevaluated in previous tournament games. Each of these positions was searched to $d = 17$ ply, while the master controlled the top $d' = 6$ ply of the tree.

Figures 14 and 15 show the speedups and overheads for parallelism in CHINOOK.

This overhead model heavily relies on the fact that the average node cost is constant over the entire test set. For KEYANO and THETURK, this is generally true; the number of nodes per second visited varies by no more than 5%. However, this is not true for CHINOOK. The number of nodes per second visited varies by at least 30%, due to numerous time-saving tricks in the code. Thus, the analysis of the overheads for CHINOOK is not as reliable as it is for the other programs given in this technical report.

In any case, there are obvious problems with the parallelization in CHINOOK. Unlike the previous two programs, we see that the APHID algorithm in CHINOOK incurs a lot of additional speculative search, and a rapidly increasing search overhead. The number of pieces of work at depth $d' = 6$ was insufficient to keep a large number of slaves busy. However, when testing $d' = 8$ ply, we discovered that serious inflexibilities were encountered, preventing us from running a full test set without removing the transposition table in CHINOOK. As well, the impact of transpositions in the game of checkers is much more important than in either chess or Othello. This causes the search overhead to increase at a much faster rate for CHINOOK than either of the previous test programs.

Although the speedups are very low in magnitude, parallelizing a high-performance checkers programs is a very difficult task. Paul Lu spent over a year working on the tournament version of parallel CHINOOK and achieved speedups of 3.32 using a shared-memory system with 16 processors [16]. The very thin game trees of checkers do not yield easily to parallelism based on the critical tree. The authors believe that when the APHID library is modified to address these problems, the APHID library should yield a better speedup for Chinook.

5 Conclusions and Future Work

The APHID algorithm yields good speedups on a network of workstations without the necessity of a shared transposition table. Although the authors are pleased with these preliminary results, a lot of work is left to be done.

As mentioned in Section 4.3, the secondary interface and underlying code within the APHID library must be fixed to allow for greater parallelism.

Our current implementation of APHID uses a fixed-depth horizon for the master's tree. As mentioned in our discussion of load balancing, all positions are not equal in the amount of search effort they require for a fixed depth of search. APHID will be generalized to support a more dynamic search horizon in the master.

Instead of basing the results of the algorithm on the speedup, we are also interested in the quality of the results returned by the algorithm. This will be necessary when the algorithm is tested beyond the fixed-depth limitations given here. The benefits and disadvantages of these alternative views of parallel performance are currently being explored.

The results reported here are based on a simple master/slave relationship. As the number of processors increases, the master increasingly becomes a bottleneck. APHID has been generalized to work in a hierarchical process tree, although this aspect of the algorithm has not been tested here. Mid-level processes would behave as a slave toward their master, and as a master toward their slaves. The scalability of the algorithm has yet to be demonstrated on architectures of more than 16 processors, due to resource limitations during the academic year.

Perhaps the biggest contribution of APHID is that it easily fits in to an existing sequential $\alpha\beta$ program. Although the first stage of the experiment uses programs written at the University of Alberta, the authors believe that the same ease of integration will be demonstrated in the second stage of the experiment, when the APHID library is given to beta-testers outside of the University of Alberta.

6 Acknowledgements

Many people have contributed to the success of this project. We would like to thank the other members of the University of Alberta GAMES group for numerous discussions and improvements to the presentation of this paper and the library: Yngvi Bjornsson, Yaoqing Gao, Andreas Junghanns, Tony Marsland and Aske Plaat. We would also like to thank Paul Lu for his comments on the APHID library, and the Computing Science Instructional Support Group at the University of Alberta for giving us the computer time for running the experiments. We would also like to thank the anonymous referees that have helped us improve the presentation of portions of this document.

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] M. G. Brockington. A Taxonomy of Parallel Game-Tree Search Algorithms. *ICCA Journal*, 19(3), 1996. In press.

[2] M. G. Brockington and J. Schaeffer. APHID Game Tree Search. In *Advances in Computer Chess VIII*, Maastricht, The Netherlands, June 1996. In press.

[3] M. G. Brockington and J. Schaeffer. The APHID Parallel $\alpha\beta$ Search Algorithm. In *Proceedings of IEEE SPDP '96*, New Orleans, Louisiana, October 1996. In press.

[4] V. David. *Algorithmique parallèle sur les arbres de décision et raisonnement en temps contraint - Etude et application au minimax*. PhD thesis, ENSAE, Toulouse, France, 1993.

[5] R. Feldmann. *Spielbaumsuche mit massiv parallelen Systemen*. PhD thesis, Universität-Gesamthochschule Paderborn, Paderborn, Germany, May 1993. In German – English translation available.

[6] E. W. Felten and S. W. Otto. Chess on a Hypercube. In G. Fox, editor, *Proceedings of The Third Conference on Hypercube Concurrent Computers and Applications*, volume II-Applications, pages 1329–1341, Passadena, CA, 1988.

[7] C. Ferguson and R. E. Korf. Distributed Tree Search and its Application to Alpha-Beta Pruning. In *Proceedings of AAAI-88*, pages 128–132, Saint Paul, MN, August 1988.

[8] F. Hsu. Private communication, 1996.

[9] F.-h. Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study*. PhD thesis, Carnegie Mellon University, Pittsburgh, U.S.A., 1990. Also Tech. Rept. CMU-CS-90-108, Carnegie Mellon University, Feb. 1990.

[10] R. M. Hyatt, B. W. Suter, and H. L. Nelson. A Parallel Alpha/Beta Tree Searching Algorithm. *Parallel Computing*, 10(3):299–308, 1989.

[11] L. Kaufman. Rate Your Own Computer. *Computer Chess Reports*, 3(1):17–19, 1992. (Published by ICD, 21 Walt Whitman Rd., Huntington Station, NY 11746.).

[12] L. Kaufman. Rate Your Own Computer - Part II. *Computer Chess Reports*, 3(2):13–15, 1993.

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

[14] B. C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1994.

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

[16] P. Lu. Private communication, 1996.

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

[18] T. A. Marsland, M. Olafsson, and J. Schaeffer. Multiprocessor Tree-Search Experiments. In D. Beal, editor, *Advances in Computer Chess 4*, pages 37–51. Permagon Press, Oxford, 1985.

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

[20] J. Pearl. Asymptotic Properties of Minimax Trees and Game-Searching Procedures. *Artificial Intelligence*, 14:113–138, 1980.

[21] A. Plaat. *Research Re: search & Re-search*. PhD thesis, Erasmus University, Rotterdam, NL, 1996.

[22] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting Graph Properties of Game Trees. In *Proceedings of AAAI '96*, volume 1, pages 234–239, Portland, Oregon, August 1996.

[23] A. Reinefeld. An Improvement to the Scout Tree-Search Algorithm. *ICCA Journal*, 6(4):4–14, 1983.

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

[25] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53(2-3):273–290, 1992.

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

[27] D. J. Slate and L. R. Atkin. Chess 4.5 - The Northwestern University Chess Program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, New York, 1977.

[28] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[29] 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.

A Description of APHID Interface

A.1 Types of Processes within APHID

- `absolute master`: The single process which invokes all other processes within the APHID process hierarchy, which occupies the highest level in the process hierarchy.
- `slave`: Any process that must report its search results to a process above it in the hierarchy. With this definition, a process is either the absolute master or a slave, and cannot be both.
- `master`: Any process that has processes underneath it in the hierarchy. Note that it is possible for a process to be both a master and a slave if the process hierarchy has multiple levels.

A.2 Constants To Be Defined

Since the library is application-independent, some definitions of how things are implemented must be given to the APHID library.

- `APHID_HASHTYPESIZE`: The size (in bytes) of the hash value used in your program.
- `APHID_HASHKEYSIZE`: The size (in bytes) of the "lock" used to guarantee that two positions sharing the same hash value are the same.
- `APHID_MOVESIZE`: The size (in bytes) of the representation of a move in the application.
- `APHID_MINUSINF` and `APHID_PLUSINF`: Specifications of values smaller than the minimum, and larger than the maximum, that the "evaluation" could possibly return, respectively.
- `APHID_INVALIDSCORE`: A value that is outside the range represented by the minimum and maximum values specified previously.
- `APHID_LOG2_TABSIZE`: The size of the APHID table that you intend on using to share between the master and the slaves, taken to a base 2 logarithm. For example, a value of 14 indicates an APHID table with $2^{14} = 16384$ entries.
- `APHID_MAXSLAVEPLYSEARCH`: The maximum "plytogo" value that you could possibly expect to hand to a slave to search. Note that this value should not be excessively large, since this and `APHID_LOG2_TABSIZE` are the leading determinants in the amount of memory used by the APHID library.
- `APHID_MAXMASTERPLYSEARCH`: The maximum depth that we expect the master should reach.
- `APHID_SLAVENODESTOCOMM`: This should be set to the number of nodes that the sequential program visits on the type of processor being used, divided by 10.

A.3 Standard Variables Used

- `argv`: Standard argument list. Used to instantiate the slaves with the same parameters as the initial program.
- `depth`: The number of ply the current node is away from the root of the game tree.
- `plytogo`: The number of ply until we reach the bottom of the game tree. In a search with no extensions or forward pruning, `plytogo + depth` should be constant.
- `&move[i]` or `&bestmove`: A pointer to an area of `APHID_MOVESIZE` bytes which specifies the move being played.
- `p_hash`: A pointer to an area of `APHID_HASHTYPESIZE` bytes that contains the current hash value.
- `p_key`: A pointer to an area `APHID_HASHKEYSIZE` bytes that contains a lock which can "guarantee" the board stored in the location of the hash table is correct.
- `alpha` and `beta`: Search window used by $\alpha\beta$ implementation; they are expected to be 32-bit integers.
- `value` and `score`: Values of leaf nodes; they are expected to be 32-bit integers for this library.

A.4 Stubs To Be Written

- `int aphid_stub_encodeinit(char *msg)`: Provides the absolute master with a 4000-byte buffer to store all pertinent information about the root of the game tree, such as the position and the game history (if this is relevant to the search algorithm). The return value is the number of bytes used in the string `msg`.
- `int aphid_stub_decodeinit(int msg_ln, char *msg)`: All other processes in the system, aside from the absolute master, receive the message length and the message encoded by `aphid_stub_encodeinit`, and should set the root of their game tree, accordingly.
- `void aphid_stub_movedownpath(int num_moves, char *movepath)`: Called by the slaves, a series of `num_moves` moves is given in `movepath`, with each move being `APHID_MOVESIZE` bytes long. The routine should play through the moves given in `movepath` from the root of the game tree.
- `void aphid_stub_moveuppath(int num_moves, char *movepath)`: Called by the slaves, should completely undo any changes made by `aphid_stub_movedownpath`.
- `int aphid_stub_iterativedeepening(int depth, int last, int max)`: Called by the slaves, this routine should return the search depth for the subsequent search of a leaf node. For most programs that do iterative deepening in 1 ply steps, this should simply return `last + 1`.
- `void aphid_stub_preparesearch(int depth, int plytogo, int winstats[], int *alpha, int *beta)`: Called by the slaves, this routine sets the initial window searched by the slaves. Statistics on previous searches and the current guessed score at the root are passed in via the `winstats[]` array. The routine should set `*alpha` and `*beta` before terminating. Furthermore, this routine should ensure that any global "alarm" states have been turned off.
- `int aphid_stub_alphabeta(int depth, int plytogo, int alpha, int beta)`: Called by both the master and the slaves, this routine should call your implementation of $\alpha\beta$, and return the minimax value back to the APHID library.
- `int aphid_stub_evaluate(int depth, int alpha, int beta)`: Called by the master the first time it visits a leaf of its tree, this routine should simply call your evaluation routine and return the score for the position reached at `depth` ply within the tree.
- `int aphid_stub_stopsearch(int pass_stats[])`: Called only by the absolute master process, this routine should check your timer and determine if your time limit has been exceeded for a search. If the time limit has been exceeded, this routine should return 1; otherwise, 0. A number of parameters are passed in to this routine via the pass_stats array, such as the number of uncertain nodes outstanding for a given search, to make the decision to terminate a search more robust.
- `int aphid_stub_visited()`: Called by the slaves, this should return a global count of the number of nodes visited by the process.

A.5 Interface Calls Used by Master and Slave

- `void aphid_startup(argv)`: In the first process run, PVM is spawned on the machines specified, and a slave process is spawned with the same argument list (`argv`) as the master, as specified by the `aphid.config` file. In a spawned process, the APHID library never exits this function call.
- `void aphid_exit()`: This routine removes a process from the PVM group, and it should be called before any process exits (due to errors or normal completion). If the process is the absolute master, completion of this routine ensures that all of the spawned processes have been shut down successfully.
- `int aphid_master()`: Returns 1 if the process is a master in the hierarchy, and 0 otherwise.
- `int aphid_slave()`: Returns 1 if the process was spawned, returns 0 if it is the absolute master process which spawned the other processes. Note that a process can be both a slave and a master depending on the hierarchy specified in `aphid.config`.

A.6 Interface Calls Used by Masters Only

- `void aphid_initserach(int maxdepth)`: Called by the absolute master process, this procedure prepares to start a search in parallel. This routine calls `aphid_stub_encodeinit`, and then informs all of the other processes of the current state of the game. The parameter indicates the maximum depth that any process is nominally allowed to search, *not including search extensions.*

- `int aphid_rootsearch(int depth, int plytogo, int alpha, int beta)`: This routine is called by the absolute master, instead of calling the typical $\alpha\beta$ implementaiton. It allows a master process to do multiple passes of the tree until the search is completed, or the alarm has been signalled via `aphid_stub_stopsearch`. If the search is allowed to complete, this routine returns the minimax value of the tree that it has been asked to search.

- `int aphid_intnode_premove(int depth, char *moveptr)`: Called by the absolute master, this routine stores moves made before a call to `aphid_rootsearch` in the move list.

- `void aphid_endsearch()`: The absolute master should call this routine when it is finished searching a tree that has been called ; it stops the slaves from working on the leaves of the game tree and prepares the slaves to receive a new root position.

- `void aphid_horizon(int depth)`: A master process calls this routine to determine if it has reached its artificial horizon.

- `void aphid_eval_leaf(int alpha, int beta, int depth, char *p_hash, char *p_key)`: This routine determines a score value for the leaf, based on the best information available to the master. `p_hash` and `p_key` are required to determine if this node has been previously visited.

- `void aphid_intnode_start(int depth, char *p_hash, char *p_key)`: Called by a master process, this routine initializes bound gathering information for an interior node within the game tree. `p_hash` and `p_key` are required to determine if this node has been previously visited.

- `void aphid_intnode_move(int depth, char *moveptr)`: Called by a master process, this routine inserts the move pointed to by `moveptr` into a hidden move list that will eventually be sent to a slave in `aphid_eval_leaf`.

- `void aphid_intnode_update(int depth, int value)`: Called by the master process, this routine takes the value returned by the child $\alpha\beta$ call and uses it to update the hidden "bound" information gathered for every node in the master's tree.

- `void aphid_intnode_fixbound(int depth)`: In the case of a cut-off, this routine will fix the hidden bound information when not all moves have been fully explored.

- `void aphid_intnode_end(int depth, int *score)`; This routine is called by a master for every internal node within the tree, and ensures that the score returned is consistent with previously gathered information about the node.

A.7 Interface Calls Used by Slaves Only

- `int aphid_checkalarm()`: This routine checks to see if a search should be terminated. If the value returned is equal to 0, the search should continue. If the value returned is not equal to 0, the current search has been interrupted, and we should terminate it in a "nice" way.

A.8 The "aphid.config" File Description

- The first line contains two integers: The number of levels in the process hierarchy, and the minimum depth of work that can be handed out to a slave.

- The next lines each contain one integer, each integer representing the value of d' for a level in the process hierarchy.

- The last lines indicate what machines the processes should be spawned on, and what executable is to be run, one line per process to be spawned. By default, APHID will spawn an executable with the same program name as the original program unless a `#` character appears on the line, which indicates a separation between the machine name and the program name. The absolute master process is not listed in this hierarchy. A process hierarchy can be specified using tabs to move the start of the machine name over in the file.