APHID Game-Tree Search

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

June 13, 1996

Abstract

This paper introduces the APHID (Asynchronous Parallel Hierarchical Iterative Deepening) game-tree search algorithm. An APHID search is controlled by a master and a series of slave processors. The master searches the first d' ply of the game-tree repeatedly. The slaves are responsible for the bottom plies of the game-tree. 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, and was implemented into a chess program with one day of programming effort. APHID yields reasonable performance on a network of workstations, an architecture where it is extremely difficult to use a shared transposition table efffectively.

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:

- *Synchronization Overhead*: The search typically has many synchronization points that result in a high percentage of processor idle time.
- *Communication Overhead*: Processes must communicate information between them; the impact of communication depends on the frequency of messages and the communication latency.
- *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 [1, 2, 3, 9, 13, 22]. 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 [16]. Initiating parallelism along the best line of play, the *principal variation*, was effective for a small number of processors, although variations on this scheme seemed limited to speedups of less than 8 [21].

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 cutoff. 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 Jamboree search [12] and ABDADA [24].

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 [8].

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 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 preliminary results of integrating the library into a chess program, and Section 5 describes some of the research directions that we are currently working on.

---

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

2 Previous Work on Parallel $\alpha\beta$ Algorithms

The idea behind the PV-Split algorithm has proven to be a fundamental building block in developing high-performance parallel game-tree algorithms [16]. 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 [17] 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.

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 $\alpha\beta$ window to be cut off [5]. This is always true at `PV` nodes, and is generally true at `ALL` nodes, assuming we start with an infinite search window at the root position. 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" [12], 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 with a low CPU cycle to message latency ratio, this type of distributed transposition table is practical and is useful in controlling the search overhead.

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 [24] 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 high CPU cycle/message latency ratios, 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 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 CPU cycle/message latency ratio.

3 The APHID Algorithm

Young Brothers Wait and other 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 to achieve reasonable performance. Algorithms based on the $\alpha\beta*$ framework require a shared transposition table to function. 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 cutoff, 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 [19]. 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 1 shows where parallel activities occur in APHID and YBW. Each location marked with an `x` shows where the parallelism typically takes place. Although more work 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'

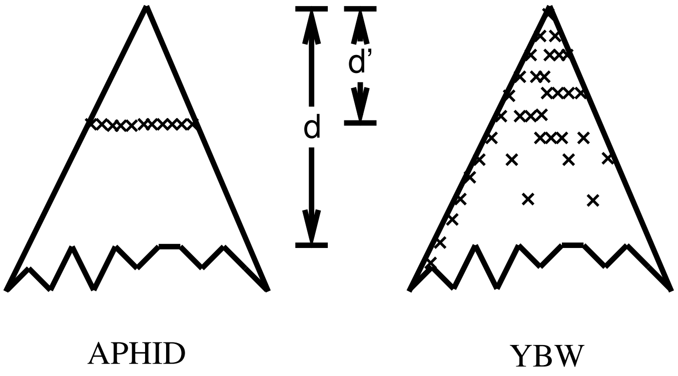

Figure 1: Location of Parallelism in Typical APHID and YBW Search

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[2]. Any node where we are forced to guess are 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 cutoff at a failed `CUT` node. If it seems likely that a child will generate a cutoff based on guessed values, the children of the failed `CUT` node are evaluated sequentially. If it seems unlikely that the node will be cut off 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

---

[2] 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 largest ply value with the same parity, instead of always using the largest ply value available.

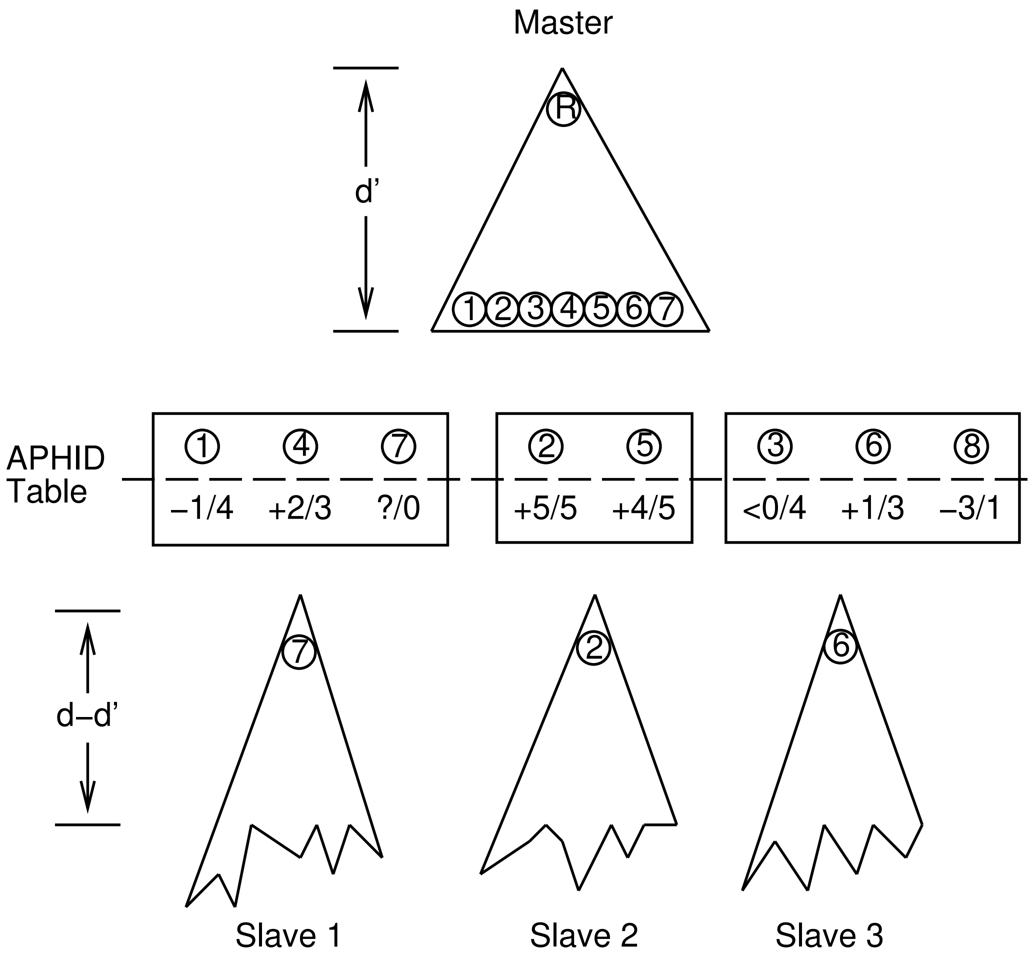

Figure 2: A Snapshot of APHID Search In Operation

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 are considered as uncertain by the master, even though we have a score bound for the $d - d'$ ply search. Eventually, the slave will return updated information that is consistent with both the original information and the search window requested[3].

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 2 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

---

[3] 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.

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 2. 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 touched 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 touched on the latest pass. However, 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 2, 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 2, 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 touched by the master. 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 an additional ply deeper, in anticipation of the next iteration (depth $d+1$). When this is happening, 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 search is immediately aborted and control is returned to the slave's scheduling algorithm.

3.4 Load Balancing

Although the master attemps 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, as discussed in Section 4.

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[18, 20] and Principal Variation Search (PVS) [16]). 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 [23] 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 3.

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 4.

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 5 illustrates the changes necessary. The only significant change from Figure 4 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 6.

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

4 Experiments

The APHID game-independent library was inserted into TheTurk chess program, written by Yngvi Bjornsson and Andreas Junghanns at the University of Alberta. With less than one day of programming effort, we had a running parallel program based on the APHID algorithm that could run on a series of workstations. Along with the additional calls added to PVS, the iterative deepening routine, and the main program, roughly 100 lines of code were defined for the `aphid_stub_` routines, whose purpose is briefly described in Appendix A.

To test the algorithm, TheTurk was programmed to search the 24 Bratko-Kopec positions [11] to a depth of $d = 8$ ply, with the master controlling the top $d' = 3$ ply of the tree.

Parallel tests were run on 4, 8, 12 and 16 workstations on a network of SUN 4s. One workstation in each experiment was completely occupied by the `master` process, while the other workstations each ran a `slave` process. On the available hardware, TheTurk visits approximately 2500 nodes per second.

Parallel and sequential algorithms are notoriously poor when it comes to exact convergence when the full version of the program is used [14]. For example, different search windows cause different search extensions to be turned on, causing different alpha-beta results. Thus, all search extensions and null move searching were turned off for 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 3). Quiescence search was left in TheTurk to prevent the evaluations from being significantly unstable.

Although this list of changes is incomplete to guarantee the same minimax value is returned by both the sequential and parallel programs, this list constitutes the changes made to TheTurk without doing serious damage to the performance of the program. In the version tested, over 90% of the parallel searches returned the same minimax value as the sequential search, and the principal variation was correct for all but 1 of the 96 searches executed in parallel.

Figure 7 illustrates the average speedups for 5 to 8 ply searches, which are the searches that were executed in parallel. The graph shows that as the depth of the search increases, so does the speedup.

```
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 3: 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 4: 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 5: 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 6: How APHID Modifies The Main Program

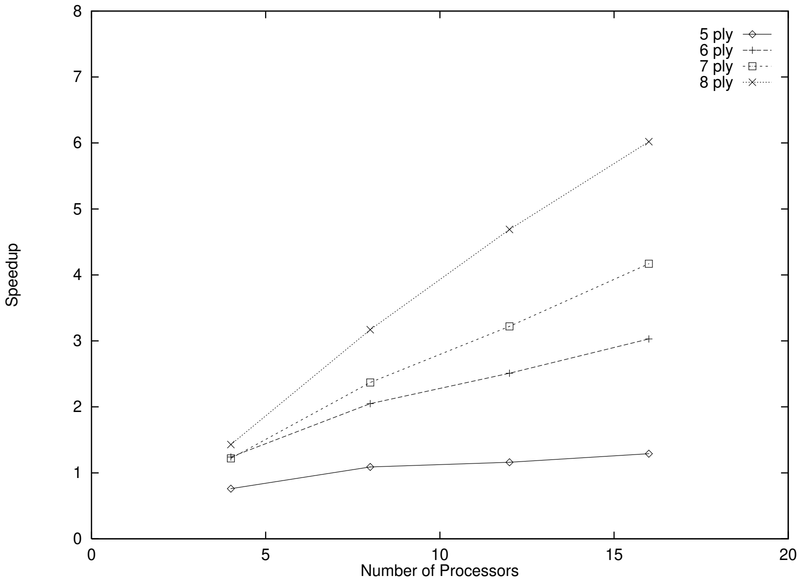

Figure 7: Speedups at Various Depths for APHID in THETURK

The 24 positions have been sorted by the size of the sequential search tree, and divided into three groups of equal size. Figure 8 illustrates the average speedup at 8 ply for each third of the complete test set.

The overheads in the algorithm are illustrated in Figure 9 as percentages of the sequential time required to search all 24 positions. The *total overhead* represents the additional computing time required by the parallel algorithm to achieve the same result:

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

where n is the number of processors. The total overhead is also a sum of the four overheads:

$$\text{total overhead} = \text{master overhead} + \text{parallelization overhead} + \text{speculative search} + \text{search overhead}.$$

The *master overhead* is the approximate penalty incurred by having a single processor being allocated completely to the handling of the master. This is not simply $\frac{1}{n}$, but is the time taken away by the master:

$$\text{master overhead} = \frac{\text{parallel time}}{\text{sequential time}}.$$

The *parallelization overhead* is the penalty incurred by the APHID-library on the speed of the slaves. In the parallel algorithm, the slaves were visiting 2000 nodes per second. The difference between this rate and the sequential program's node rate is derived partially from the overhead of using PVM, and partially from the work-scheduling algorithm on each slave. In the authors' experience, this complexity overhead is approximately the same as implementations of YBW on similar hardware.

The *search overhead* starts at 70% and increases very gradually as we increase from 4 to 16 processors. 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

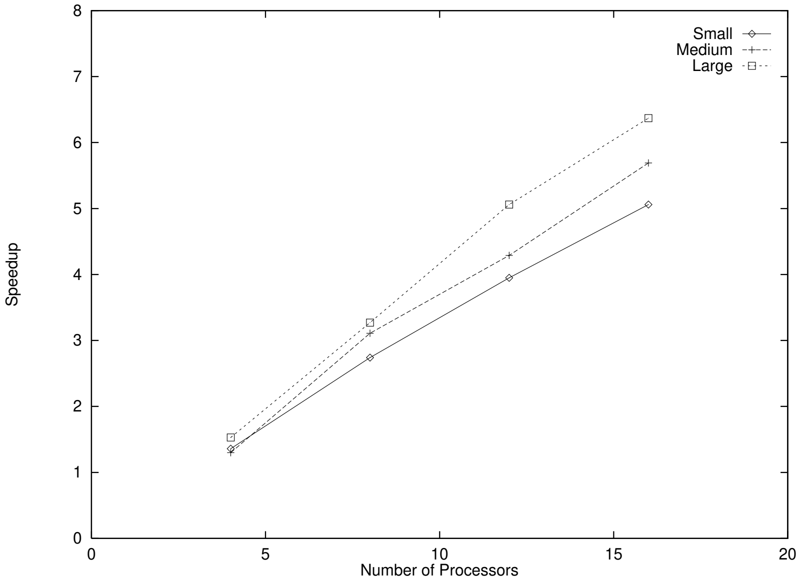

Figure 8: Speedups for Different Size Problems at 8 Ply in TheTurk

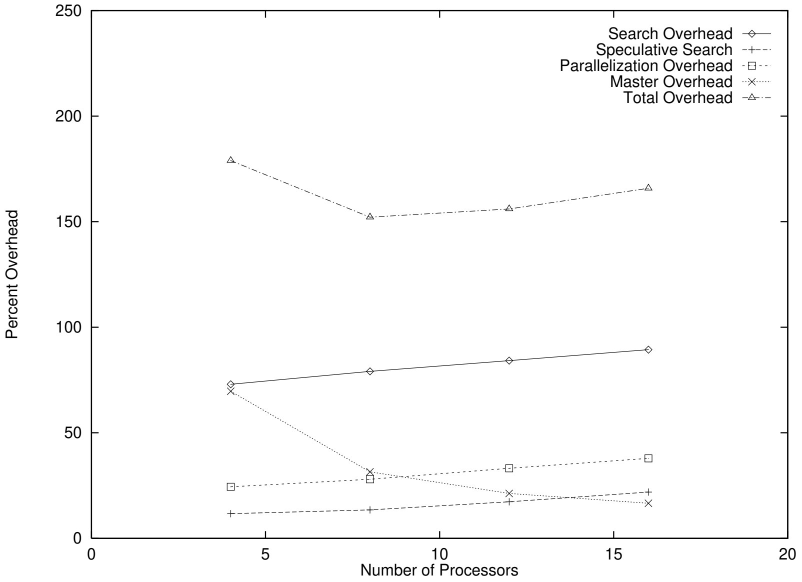

Figure 9: Overheads at 8 Ply for APHID in TheTurk

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 8 ply, the asynchronous nature of the slaves will result in some work being done at 9 ply (or more). The *speculative search* line represents the amount of additional search beyond what the sequential algorithm would have done. However, 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.

As discussed earlier, load balancing was used for generating the results given here. This is based on the assumption that the primary metric to measuring a parallel program's performance is the speed with which it achieves a d-ply result. However, one could argue that the quality of the result achieved is equivalent, irrespective of the speedup. The additional speculative search done in cases where the load balance is incorrect could be used to improve the quality of the result returned.

Other results for parallel search algorithms on a network of workstations have been presented for the game of chess [21]. 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.

Although the results presented here for chess are promising, we believe that better results can be achieved. The results for parallelizing Keyano, an Othello program, yield a 9-fold speedup with the APHID library without the use of load balancing. The main difference between the results is that the search overhead for Keyano is less than half of the TheTurk's 70% search overhead.

The authors have come up with two possible hypotheses for this discrepancy. The first is that Keyano does not use quiescent search. As a consequence, the sizes of the various pieces of work are more predictable; hence, load balancing is not necessary. 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 TheTurk than in Keyano.

The APHID algorithm can support a shared transposition table, but the algorithm does not depend on its presence. Thus, the algorithm gets good performance on a loosely-coupled network of workstations and will perform even better on tightly-coupled processors.

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.

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 is being generalized to support a dynamically changing 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 alternative view of parallel performance is 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. Not only has APHID been integrated into THETURK and KEYANO, APHID has been integrated into the checkers program CHINOOK with one afternoon of effort.[4] In contrast, the version of CHINOOK used in the 1994 match versus Tinsley took 4 to 6 weeks to implement [15].

6 Acknowledgements

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

References

[1] S. G. Akl, D. T. Barnard, and R. J. Doran. Design, Analysis and Implementation of a Parallel Tree Search Algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(2):192–203, 1982.

[2] G. M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1978. Available as Tech. Rept. CMU-CS-78-116.

[3] V.-D. Cung. *Contribution à l'Algorithmique Non Numérique Parallèle: Exploration d'Espaces de Recherche.* PhD thesis, Université Paris VI, April 1994.

[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.

---

[4] The results for CHINOOK were not available at the time of writing.

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

[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] 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, Oxford, 1982.

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

[13] G. Lindstrom. The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm. Technical Report UUCS 83-101, University of Utah, Department of Computer Science, Salt Lake City, UT, March 1983.

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

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

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

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

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

[19] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Nearly Optimal Minimax Tree Search? Technical Report 94-19, Department of Computing Science, University of Alberta, Edmonton, Alta., December 1994.

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

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

[22] I. R. Steinberg and M. Solomon. Searching Game Trees in Parallel. In *Proccedings of the 1990 International Conference on Parallel Processing (vol. 3)*, pages 9–17, University Park, PA, 1990. Penn. State University Press.

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

[24] 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 can not 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. 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 next search of a leaf node from a master's tree. 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.
- `int aphid_stub_visited()`: Called by the slaves, this should return a global count of the number of nodes visited by the program.

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 cutoff, 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 granularity of work that we are allocated in parallel.
- 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.