The APHID Parallel $\alpha\beta$ Search Algorithm

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

Abstract

This paper introduces the APHID (Asynchronous Parallel Hierarchical Iterative Deepening) game-tree search algorithm. APHID represents a departure from the approaches used in practice. Instead of parallelism based on the minimal search tree, APHID uses a truncated game-tree and all of the leaves of that tree are searched in parallel. APHID has been programmed as an easy to implement, game-independent $\alpha\beta$ library, and has been tested on several game-playing programs. Results for an Othello program are presented here. The algorithm yields good parallel performance on a network of workstations, without using a shared transposition table.

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, most 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.

In real-world implementations, such as 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 [6]).

The first is *synchronization overhead*. The search typically has many synchronization points where there is no work available, which results in a high percentage of idle time.

The second is *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.

The third is *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 (a more complete list is available elsewhere [1]). 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 [5]. 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 [7].

The idea can be generalized to other nodes in the tree. At nodes where the first branch has been searched and no cut-off occurred, 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, such as Jamboree search [4] and ABDADA [9]. The best-known instance of this type of algorithm is called *Young Brothers Wait* (YBW) and was implemented in the $Zugzwang$ chess program [3]. YBW achieved a 344-fold speedup using a network of 1024 Transputers.

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, with the absolute performance being strongly linked to the quality of the move ordering within the game-tree.

This paper introduces the Asynchronous Parallel Hierarchical Iterative Deepening (APHID) 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 distributed transposition table, which makes the algorithm suitable for loosely-coupled architectures (such as a network of workstations), as well as tightly-coupled architectures.

Unlike some 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 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.

2. The APHID Algorithm

Young Brothers Wait and other similar algorithms suffer from three serious problems. First, the numerous synchronization points and occasions where there is little or no work to be done in parallel result in idle time. This suggests that a new algorithm must strive to reduce or eliminate synchronization and small work lists. Second, the chaotic nature of a work-stealing scheduler requires algorithms such as YBW and Jamboree to use a shared transposition table to achieve good move ordering and reasonable performance. ABDADA requires a shared transposition table to function correctly. 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, for example, causes a cut-off, then all the parallel work 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*[1] of a game-tree near the root vary infrequently between steps of iterative deepening. 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.

2.1. 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 the master's stopping condition, 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. However, if the $d - d'$ ply result is not available, then the algorithm uses the deepest search result that has been returned by the slave to generate a guessed minimax value. 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 over the tree. As long as the score at any of the leaves is uncertain, the master must do another pass over 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.

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 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.

2.2. The APHID Table

If a 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 1 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 dashed line in Figure 1. For each leaf that has been visited

---

[1] The ply of a node is its depth within the game-tree, starting with ply 0 at the root of the game-tree.

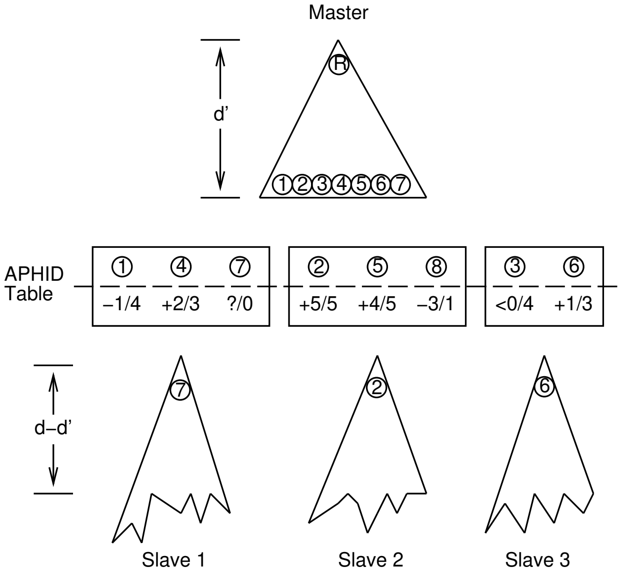

Figure 1. A Snapshot of an APHID Search

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 there are approximately the same number of leaves which 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, but 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 allocated to the slaves in a round-robin manner. Although there may be better methods of allocating leaves, it has been found that this is a reasonable method of balancing the work on a small number of processors.

The slave's part of the table, illustrated by the area below the dashed 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.

2.3. 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 steps until the master tells it that the search is complete. The slave looks in its portion of its local copy of the APHID table, and finds the highest priority node to search. The slave then executes the search, and reports 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. The secondary criterion, if the primary criterion is the same, is based on 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.

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 2, 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 likely 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 nodes 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.

2.4. 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. Since the library is application-independent, a potential user must write a few application-dependent routines (such as move format, how

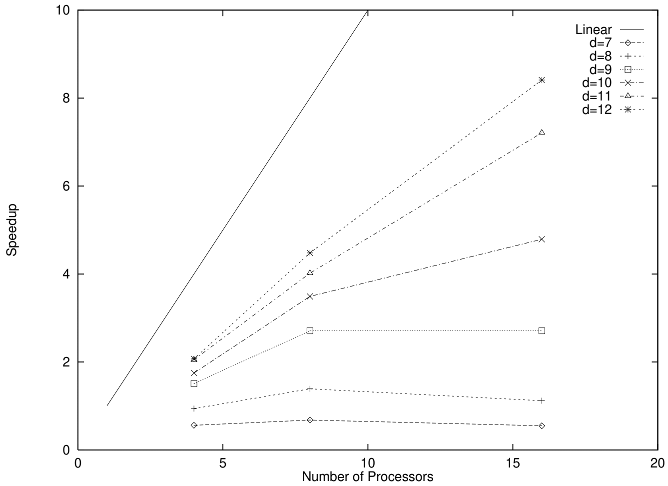

Figure 2. APHID Speedups in Keyano

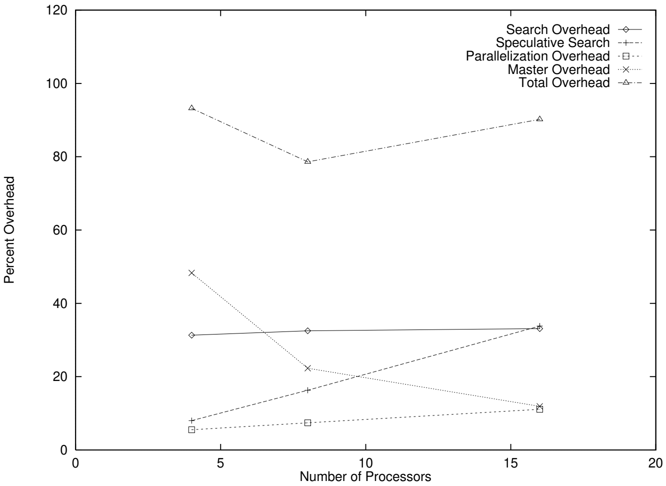

Figure 3. 12 Ply Overheads

to make/unmake moves, position format, setting a window for a slave's search, *etc.*). APHID's message passing was written using PVM [8] to allow for the maximum portability among available hardware.

To parallelize a sequential $\alpha\beta$ program, the user modifies their sequential search routine to add approximately 20 lines of code. This search routine functions as the search algorithm for both the master and the slave processes. There are a few additional procedure calls that have to be added to the iterative deepening routine that calls the search routine. A complete explanation of the current APHID interface can be found elsewhere [2].

3. Experiments

The APHID game-independent library was inserted into the Othello program, Keyano, which has frequently finished in the top five in international computer Othello tournaments over the last three years.

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. The search depth is typical of what the parallel program could achieve within the tight time constraints of Othello competitions (typically 30 minutes per game). Deeper searches will yield better speedups, but are not indicative of what can be achieved in real time. 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.

Parallel tests were run on 4, 8 and 16 workstations on a network of SparcStation IPC computers with 12 MB of RAM, running the SunOS 4.1.4 operating system. The computers 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 slave process. Figure 2 illustrates the average speedups for 7 to 12 ply searches. The graph shows that as the depth of the search increases, so does the speedup.

The overheads in the algorithm are illustrated in Figure 3 as percentages of the sequential time required to search all 74 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: master overhead, parallelization overhead, speculative search and 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: the parallel time divided by the sequential time.

The *parallelization overhead* is the penalty incurred by the APHID library on the speed of the slaves. Keyano visits approximately 8700 nodes per second on a single SparcStation IPC. Using the APHID library, the slaves were visiting 7600 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 parallelization overhead is similar to implementations of YBW on equivalent hardware (the results have not been presented here because Keyano has been rewritten since the original experiment with YBW in 1994).

The *search overhead* starts at 30% 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. Part of the search overhead is due to information deficiency, since there is no common shared data between the slave processes.

Since we only search each position to 12 ply, the asynchronous nature of the slaves will result in some work being done at 13 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.

Weill tested YBW and ABDADA on a CM-5 using a different Othello program [9]. 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 the performance of the algorithms.

Other results for parallel search algorithms on a network of workstations have been presented for the game of chess [7]. There is more parallelism available in the wider chess trees, which results in better speedups in comparison to Othello [9]. Although, a distributed transposition table was used to improve the performance of the parallel chess searches, a speedup of 7 was possible on 16 processors. These results were extrapolated to a speedup of 8 on 32 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.

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.

4. 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.

The load balancing of the APHID algorithm has not been addressed in this paper. More intelligent schemes than round-robin allocation of work are currently being investigated.

Our current implementation of APHID uses a fixed-depth horizon for the master's tree. All positions are not equal in the amount of search effort they require. APHID is being generalized to support a dynamically changing horizon in the master.

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. 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.

Perhaps the biggest contribution of APHID is that it easily fits into an existing sequential $\alpha\beta$ program. As a validation of this, APHID has been integrated into a chess and checkers program with one afternoon of effort. The application-dependent code is roughly 150 lines, and took less than an hour to write. Although all of the programs in the original testing were designed at the University of Alberta, the second stage of the experiment will be to release the code to interested members of the high-performance games community outside of the University of Alberta.

5. Acknowledgements

The authors would like to thank the other members of the GAMES group at the University of Alberta for their assistance and constructive criticisms: Yngvi Bjornsson, Andreas Junghanns, Yaoqing Gao, Tony Marsland and Aske Plaat. The authors would also like to thank the anonymous referees whose comments were valuable in improving the presentation of the original paper. 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*, 1996. In press.

[2] M. G. Brockington and J. Schaeffer. The APHID Parallel $\alpha\beta$ Search Algorithm. Technical Report 96-07, Department of Computing Science, University of Alberta, July 1996.

[3] R. Feldmann. *Spielbaumsuche mit massiv parallelen Systemen*. PhD thesis, Universität-Gesamthochschule Paderborn, Paderborn, Germany, May 1993.

[4] B. C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, M.I.T., Cambridge, MA, 1994.

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

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

[7] J. Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6(2):90–114, 1989.

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

[9] J.-C. Weill. *Programmes d'échecs de championnat: architecture logicielle synthèse de fonctions d'évaluations, parallélisme de recherche*. PhD thesis, Université Paris 8, 1995.