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'