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.

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

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