A TAXONOMY OF PARALLEL GAME-TREE SEARCH ALGORITHMS

Mark G. Brockington [1]

Edmonton, Alberta

1 INTRODUCTION

In the last twenty years, a number of articles and theses have been written that contain innovative parallel game-tree search algorithms. The authors of the parallel algorithms have shown how their work is unique and interesting. In some cases, this has been shown by classifying other algorithms by listing implementation details (Bal and van Renesse, 1986; Ciancarini, 1994). To the author's knowledge, no attempt has been made to classify the algorithms based solely on their algorithmic properties. A taxonomy would make it easy to ascertain what has and has not been accomplished in parallel game-tree search. The presentation of this type of taxonomy is the main contribution of this paper.

The taxonomy will be broken into two major categories: $\alpha\beta$-based algorithms, and algorithms based on other search paradigms ($SSS*$, ER, and theoretical methods). For the former category, a table is given to isolate the fundamental differences between the algorithms. The table is divided into two parts: the first part contains characteristics of the $\alpha\beta$-based algorithms, while the second part contains details about an implementation of each algorithm. Section 2 describes the various columns given in the table, and then gives some brief details on the algorithms contained therein.

The algorithms based on other search paradigms are given in Section 3. Due to the varied nature of the methods, a brief description is given for each of the algorithms and no attempt has been made to categorize them to the same extent as the $\alpha\beta$-based algorithms. The implementation details have not been organized into a table, since some of the algorithms given are of a theoretical nature and have not been implemented or simulated.

The final section deals with some conclusions that can be drawn from the taxonomy.

2 $\alpha\beta$-BASED PARALLEL GAME-TREE SEARCH

This section discusses algorithms based on $\alpha\beta$ and will be split into four subsections. Sections 2.1 and 2.2 will be used to describe the columns in Tables 1 and 2, respectively. The description of the algorithms has been divided into two eras. Section 2.3 covers work first submitted up to June, 1987. Most of the work from this era describes algorithms to be used on a limited number of processors. Section 2.4 covers work that was first submitted on or after July, 1987. The majority of the algorithms from this era are designed for implementation on massively-parallel hardware systems, which were unavailable to most researchers before 1987.

2.1 Comparison of the $\alpha\beta$ Algorithms

Table 1 summarizes and classifies the various $\alpha\beta$-based algorithms, and is divided into five columns.

The first column gives the name of the algorithm, and the reference that contains the most details about the algorithm. For example, the Young Brothers Wait concept has been described in many papers, but all of the algorithm's details are located in Feldmann's thesis (1993).

---

[1] Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1.
Email: `brock@cs.ualberta.ca`

Algorithm (Reference)	Date First Described	Processor Hierarchy/ Control Distribution	Parallelism Possible At These Nodes	Synchronization Done At These Nodes
Parallel Aspiration Search (Baudet, 1978)	1978	Static/ Centralized	Root ($\alpha\beta$ window)	Root
Mandatory Work First (Akl, Barnard and Doran, 1982)	1979	Static/ Centralized	Type 1+3+Left-most child of 3	Bad Type 2
Tree Splitting (Finkel and Fishburn, 1982)	1980	Static/ Centralized	Top k-ply	Root
PV-Split (Marsland and Campbell, 1982)	1981	Static/ Centralized	Type 1	Type 1
Key Node (Lindstrom, 1983)	1983	Static/ Centralized	Type 1+3+Left-most child of 3	Bad Type 2
UIDPABS (Newborn, 1988)	1986	Static/ Centralized	Root	None
DPVS (Schaeffer, 1989)	01/1987	Dynamic/ Centralized	Type 1+3+2	Type 1+3+Bad 2
EPVS (Hyatt, Suter and Nelson, 1989)	06/1987	Dynamic/ Centralized	Type 1+3	Type 1+3
Waycool (Felten and Otto, 1988)	1987	Dynamic/ Distributed	All, except Type 2 with no TT entry	Nodes with TT & no cutoff
Young Brothers Wait (Feldmann, 1993)	10/1987	Dynamic/ Distributed	Type 1+3+Bad 2	Type 1+Bad 2
Dynamic Tree Splitting (Hyatt, 1988)	1988	Dynamic/ Distributed	Type 1+3+Bad 2	Type 1+Bad 2
Bound-and-Branch (Ferguson and Korf, 1988)	08/1988	Dynamic/ Distributed	Type 1+3+Bad 2	Type 1+Bad 2
Delayed Branch Tree Expansion (Hsu, 1990)	1990	Static/ Centralized	Type 1+3	Bad Type 2
Frontier Splitting (Lu, 1993)	1993	Dynamic/ Distributed	All	Root
$\alpha\beta$* (David, 1993)	1993	Dynamic/ Distributed	Type 1+3	Type 1+3+Bad 2
CABP (Cung, 1994)	1994	Static/ Centralized	Type 1+3	Bad Type 2
Jamboree (Kuszmaul, 1994)	1994	Dynamic/ Distributed	Type 1+3+Bad 2	Type 1+Bad 2
ABDADA (Weill, 1995)	1995	Dynamic/ Distributed	Type 1+3+Bad 2	Type 1+Bad 2
Dynamic Multiple PV-Split (Marsland and Gao, 1995)	1995	Dynamic/ Distributed	Nodes within PV set	Nodes within PV set
APHID (Brockington and Schaeffer, 1996)	1996	Static/ Centralized	Top k-ply	None

Table 1: Comparison of Parallel $\alpha\beta$-based Game-Tree Search Algorithms

The second column gives the date that the algorithm was first published or received by a journal. This information has been used to order the algorithms into chronological order.

The third column contains information on both the processor hierarchy and the distribution of control within the algorithm. *Processor Hierarchy* categorizes algorithms based on the rigidity of the processor tree. If the processor tree is *static*, one or more processors are designated as masters, and control the other slave processors. This hierarchy is fixed throughout a search of the game-tree. A *dynamic* processor tree changes based on the distribution of busy and idle processors. *Control Distribution* describes whether the control of the algorithm is *centralized* on a small number of masters (e.g. PV-Split), or could be *distributed* amongst all processors (e.g. Young Brothers Wait).

The fourth column describes the typical nodes of the game-tree where parallelism could occur. The description is based on the classification of the minimal game-tree by Knuth and Moore (1975). Note that the type 2 (cut nodes) have been differentiated into two sub-classes. When a type 2 node has not been pruned after searching the first move, this node is called a *bad type 2* node due to its incorrect move ordering. Similarly, *good type 2* nodes are considered to be type 2 nodes that cause a cutoff after examining the first move (i.e. the move ordering is correct).

For example, PV-Split only implements parallelism at type 1 (principal variation) nodes, while the Young Brothers Wait algorithms allow for parallelism at type 1, type 3 (all nodes) and bad type 2 nodes. At good type 2 nodes, the Young Brothers Wait algorithm will search the first move, achieve a cutoff, and none of the other children will be evaluated.

The fifth column indicates which nodes of the game-tree might have parallelism constrained by waiting for the first k children to be evaluated. For example, bad type 2 nodes are synchronization points for Akl *et. al*'s Mandatory Work First algorithm, while type 1 and bad type 2 nodes are synchronization points for Ferguson and Korf's Bound-and-Branch algorithm.

2.2 Comparison of the $\alpha\beta$ Implementations

Table 2 summarizes an implementation of each algorithm given in Table 1.

The first column gives the name of the algorithm, and the reference to the paper that contains the details about the implementation. In some cases, this paper may be different than the paper which best describes the algorithm.

The second column describes the underlying hardware used to host the selected implementation. A software *simulation* of hardware is denoted in this column.

The third column describes the type of game-trees explored by the algorithm. If the game-trees are not generated by game-playing programs, they are considered to be *artificial trees.* In terms of average branching factor, chess trees are wider than Othello trees, and both are wider than checkers trees.

The fourth column denotes which of the sequential game-tree searching algorithms was parallelized: $\alpha\beta$ (Brudno, 1963; Knuth and Moore, 1975), PVS (Marsland, 1983) or NegaScout (Reinfeld, 1983). Some programs are more efficient when using a different sequential algorithm, depending on the nature of the evaluation function and the strength of the move ordering techniques in the sequential program. Thus, the choice of sequential algorithm to compare the parallel algorithm against is a factor to consider when evaluating a parallel algorithm's results.

The fifth column describes what type of transposition table has been implemented for the algorithm. Efficient sharing of transposition table information is crucial to the performance of a parallel game-tree search algorithm. The two main methods are a *distributed* message-passing transposition table and a *shared-memory* transposition table. Special hardware is required to implement a shared-memory transposition table, but is

Algorithm (Reference)	Hardware Used	Test Domain	Sequential Algorithm	Trans-Position Table	Speedup Obtained
Parallel Aspiration Search (Baudet, 1978)	Simulation	Artificial Trees	$\alpha\beta$	none	$\leq$ 6 for large n (simulated)
Mandatory Work First (Akl, Barnard and Doran, 1982)	Simulation	Artificial Trees	$\alpha\beta$	"score table"	$\leq$ 6 for large n (simulated)
Tree Splitting (Finkel and Fishburn, 1982)	LSI-11 & Simulation	Checkers	$\alpha\beta$	none	2.34 (n=3) 5.12 (n=27,sim)
PV-Split (Marsland, Olafsson and Schaeffer, 1985)	Sun 3 Network	Chess	PVS	local	3.75 (n=5)
Key Node (Lindstrom, 1983)	Simulation	Artificial Trees	$\alpha\beta$	none	12.57 (n=20)
UIDPABS (Newborn, 1988)	Data General (mixed procs.)	Chess	$\alpha\beta$	local	3.94 (n=8)
DPVS (Schaeffer, 1989)	Sun 3 Network	Chess	NegaScout	TT. Mgr - messages	7.64 (n=19)
EPVS (Hyatt, Suter and Nelson, 1989)	Sequent Balance	Chess	$\alpha\beta$	shared memory	5.93 (n=16)
Waycool (Felten and Otto, 1988)	Hypercube	Chess	$\alpha\beta$	distributed messages	101 (n=256)
Young Brothers Wait (Feldmann, 1993)	Transputers	Chess	NegaScout	distributed messages	142 (n=256) 344 (n=1024)
Dynamic Tree Splitting (Hyatt, 1988)	Sequent Balance	Chess	$\alpha\beta$	shared memory	8.81 (n=16)
Bound-and-Branch (Ferguson and Korf, 1988)	Hypercube	Othello	$\alpha\beta$	distributed messages	12 (n=32)
Delayed Branch Tree Expansion (Hsu, 1990)	Simulation	Chess	$\alpha\beta$	none	350 (n=1000, simulated)
Frontier Splitting (Lu, 1993)	BBN TC2000	Checkers	NegaScout	shared memory	3.32 (n=16)
$\alpha\beta$* (David, 1993)	Transputers	Chess	NegaScout	distributed messages	6.5 (n=8+8TT)
CABP (Cung, 1994)	Sequent Balance	Artificial Trees	$\alpha\beta$	shared memory	4.6 (n=9)
Jamboree (Kuszmaul, 1994)	CM-5	Chess	NegaScout	distributed messages	≈50 (n=512)
ABDADA (Weill, 1996)	CM-5	Chess (& Othello)	NegaScout	distributed messages	15.85 (n=32)
Dynamic Multiple PV-Split (Marsland and Gao, 1995)	AP-1000	Artificial Trees	PVS	none	≈32 (n=64)
APHID (Brockington and Schaeffer, 1995)	Sparc 2 Network	Chess	NegaScout	local	6.04 (n=16)

Table 2: Comparison of Parallel $\alpha\beta$-based Game-Tree Search Implementations

generally faster than distributed transposition tables based on message passing. Local transposition tables are maintained separately on each processor, but no transposition table information is shared between the processors.

The final column gives the *speedup* of the implementation on a large number of processors. There will be a tendency to read down this column and use the absolute speedup or relative efficiencies to determine something about a pair of algorithms. It is very hard and misleading to compare two algorithms in this manner. There are five major concerns when comparing speedup numbers.

The first concern deals with simulation results. It is often difficult to determine how a parallel algorithm will behave on a large number of machines, due to over-simplification of the simulation model being used.

The second major concern deals with the test domain. Artificial trees rarely reflect the unique properties of searching a real chess/checkers/Othello tree, and might be geared towards illustrating the strength of the parallel algorithm (Plaat *et. al*, 1996).

The third major concern is that we cannot take the real trees at face value either, since the sequential algorithm used to generate the tree may not be an efficient searcher (with respect to the minimal tree size). This could be caused by leaving out key move ordering techniques, such as a sufficiently large transposition table, iterative deepening or killer moves. A poor searcher will yield more opportunities for parallelism, and may increase the speedup achieved by the algorithm.

The fourth concern deals with the varied branching factor of the game-trees in different test domains. The average branching factor in chess (38) is higher than the average branching factor in checkers (8 for non-capture positions). Taking capture positions into account, the average branching factor of checkers is less than 3 (Lu, 1993). The breadth of checkers trees yield speedups that are much smaller in magnitude than the same algorithm implemented on a chess program.

The final concern deals with the speed of the processor versus the speed of the network. If the algorithm was tested on slow processors with a fast network linking them, the algorithm may not yield the same performance when using faster processors and/or a slower network.

In short, it is nearly impossible to objectively compare speedups or efficiencies for the different implementations given in Table 2.

2.3 Descriptions of Earlier Algorithms (1979-June 1987)

Baudet's work (1978) described a method of doing *parallel aspiration search*. Aspiration search can be described as the shrinking of the initial $\alpha\beta$ window to a small range. If the minimax value lies within the smaller range, the correct minimax value could be returned while visiting less leaf nodes than would have been visited by using the larger range.

In Baudet's work, the initial $\alpha\beta$ window is subdivided into p disjoint windows, where p is the number of processors used. Each processor searches the game-tree with those smaller windows. When a processor is finished, it can use the result of the search (if it is a fail low or fail high) to further reduce the size of the windows examined. Once a processor determines the minimax value, all processors are stopped immediately.

Akl, Barnard and Doran (1982) were the first to propose and simulate a *mandatory work first* algorithm. The idea of the algorithm is to explore in parallel those leaves that would be examined if the tree was perfectly ordered. There are two categories of nodes that correspond to the cut and all nodes from the minimal tree. Left-hand nodes are similar to all nodes, and all of their successors are evaluated at different processors in parallel. Right-hand nodes are similar to cut nodes, and only one successor process can be spawned from them at a time.

The first branch to be evaluated from a right-hand node might establish a score that signifies a cutoff. The process controlling a right-hand node is forced to stall and find out the value of the sibling left-hand node. Once this sibling left-hand node has a value, the two values are compared and the program determines whether or not the right-hand node can be pruned based on the first branch that was searched. If there is no cutoff yet for the right-hand node, the subsequent branches in the right-hand node are examined one after the other sequentially. This will stop when the right-hand node gets pruned, or the right-hand node establishes a value higher than the sibling left-hand node after exploring all branches. This allows the scheme to determine most of the direct shallow cutoffs that would occur in the sequential $\alpha\beta$ algorithm, but neglects some of the deep cutoffs possible.

Finkel and Fishburn (1982) introduced the concept of *tree splitting*. In their algorithm, a static tree of processors is overlaid on top of a game-tree. The root of the game-tree is given to the root of the processor tree. The processor root generates all the moves at ply 1 of the game-tree, and hands them over to the first ply of the processor tree. This process continues until we reach the leaves of the processor tree, where the processors execute the sequential $\alpha\beta$ algorithm to the required search depth. The nodes in the first k levels of the tree, where k is the depth of the processor tree, can be evaluated in parallel. The only synchronization point occurs at the root of the tree between searches at different depths.

The *PV-Split* algorithm (Campbell, 1981; Marsland and Campbell, 1982) is a natural extension of tree splitting, based on the regular structure of the minimal game-tree as we travel along the principal variation (PV). The first stage of the algorithm involves a recursive call to itself as it travels down the principal variation. Once the left subtree of a PV node has been examined, all of the other subtrees below that PV node are searched in parallel using tree splitting. After all of the subtrees have been explored, that PV node can return a score to the PV node above it. At any one time, only one node's subtrees are being examined in parallel by the PV-Split algorithm.

In the original implementation (Marsland and Campbell, 1982), the PV-Split algorithm did not use minimal windows or other search enhancement techniques common in game-playing programs. To simulate these ordering techniques, strongly-ordered game-trees were artificially created. The PV-Split algorithm was shown to have a better speedup than tree splitting for the simulated strongly-ordered trees. There are numerous published experiments with the PV-Split algorithm (Marsland and Popowich, 1985; Newborn, 1985; Marsland, Olafsson and Schaeffer, 1985). The best reported efficiency in these implementations was a speedup of 3.75 on 5 processors for Marsland, Olafsson and Schaeffer (1985). The major problem in the implementation of PV-Split is a large synchronization overhead, since many processors are often forced to wait for long periods of time while the last unevaluated branch of a PV node is evaluated. Newborn suggested that idle processors should assist those processors that are still busy by creating a new split node that the idle processors can work on (Newborn, 1985).

The *Key Node* method (Lindstrom, 1983) attempts a different method for attacking the tree. The mandatory work first tree is dynamically evolved and stored within a centralized message queue. Each processor takes a message from the queue, creates new messages based on the type of message, and adds the information into the tree, as required. For example, if a message is sent to a leaf node within the tree, the node is evaluated, and the value is sent to the parent. At type 1 and type 3 nodes, messages for each of the children can be sent out at the same time (yielding nearly ideal parallelism). Synchronization occurs at bad type 2 nodes, where each move is tried in turn in an attempt to find the cutoff.

The Key Node method was simulated using artificial trees, where the score of the parent was similar to the score of the children. The method was compared against the classical $\alpha\beta$ algorithm. In the simulation, each message was assumed to be processed in unit time, and some efforts were made to simulate contention for the nodes within the tree. Over 10 test runs, the Key Node method achieved a speedup of 12.57 on 20 processors, using a tree depth of 5 and a breadth of 4.

Newborn's (1988) algorithm, *Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search*, 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. Some of the moves may have been evaluated to larger depths than those on other processors, which may yield a better quality move choice.

Schaeffer's *Dynamic PV-Split* algorithm (1989) is an enhancement that allows for dynamic processor trees in the PV-Split framework. Instead of the fixed processor tree mechanism that was used in PV-Split, processors in Dynamic PV-Split (DPVS) are allowed to dynamically attach themselves to other busy processors, which each run the PV-Split algorithm. This allows for parallelism along the pseudo-principal variation (the leftmost branch) being searched by any processor, and allows for multiple split nodes. The process of choosing the new split node started by allocating branches at type 1 or type 3 nodes, and allowed parallelism at type 2 nodes once the branches from all type 1 and 3 nodes were allocated. All requests for work went through a Controller process, which was used to balance the dynamic processor tree amongst the processors that had the most work to do, as well as assign work from the current node on the principal variation.

Unfortunately, the increase in search overhead compensated for the decrease in synchronization overhead. The search overhead arose from additional processors, once they had been reassigned, attempting to search some subtrees without the benefit of the ordering information from the searched sibling subtrees. By allowing a shared Table Manager to handle all transposition table requests, along with a mechanism for rebroadcasting history table information, the speedup for ParaPhoenix was improved to 7.64 on 19 processors. The mechanism described in the paper tapered off once more than 10 processors were involved; the overhead of going through a single Table Manager increased linearly as more processors were added.

The *Enhanced PV-Split* algorithm (Hyatt, 1988; Hyatt, Suter and Nelson, 1989) is a different type of dynamic allocation to the PV-Split algorithm. In the Enhanced PV-Split (EPVS) algorithm, when a processor became idle, all of the other processors were stopped and a new split node would be created two ply further down the tree of one of the busy processors. All processors would then start to work on the smaller subtree. The transposition table ensured that the smaller subtree had not been explored yet.

Using a Sequent Balance 21000 computer, the speedup of EPVS was 5.93 on 16 processors. On the same machine and test set, PV-Split achieved a speedup of 4.57 on 16 processors. The authors point out at the end of their paper that the average branching factor of a chess tree is 38; their algorithm could not use 64 or more processors effectively since all processors co-ordinate at one split node at any given time. Thus, to use massively parallel architectures (with hundreds or thousands of processors), a greater number of split nodes must be available for parallel work.

2.4 The Advent Of Massive Parallelism (July 1987-present)

Felten and Otto (1988) implemented the first parallel $\alpha\beta$ algorithm that played chess on more than 32 processors. Their WayCool program decided on the type of parallelism to be applied at a node based on whether there was a transposition table entry in the system. If a transposition table entry was available, the move stored would likely be the best move, and it was worth waiting for a bound to be returned from that transposition table move. Once that bound had been returned (assuming that the node is not immediately pruned), all of the other successors could be explored in parallel. If there is no transposition table information, all subtrees could be computed in parallel.

The processors are hierarchically organized into a tree structure at the start of the search, but this processor tree would be restructured as necessary. The scheme relied on a globally shared transposition table and a load balancing scheme that is similar to the one used in EPVS. The load balancing scheme reorganized searchers into new teams that search a “hot spot” in parallel.

Feldmann *et. al* implemented a parallel $\alpha\beta$ algorithm on a large network of Transputers for the chess program ZUGZWANG (Vornberger and Monien, 1987; Feldmann *et. al*, 1989; Feldmann, 1993). The algorithm involves the use of the *Young Brothers Wait Concept* (YBWC) to determine when nodes can be given out in a parallel manner.

In a game-tree that has near perfect ordering, there is a high probability that a node is an all node if we evaluate the leftmost branch and we have not pruned the search below that node. The basic Young Brothers Wait Concept states that the leftmost branch (the eldest brother) must be evaluated before any other branches (the young brothers) can be distributed to other processors. This is not necessarily limited to the principal variation (i.e. PV-Split) or a pseudo-principal variation (i.e. DPVS or EPVS); it can happen at any node within the game-tree. The algorithm given in Tables 1 and 2 is YBWC*. This variation does not wait for young brothers at type 3 nodes and forces sequential evaluation of all “reasonable” moves at type 2 nodes, as described in Feldmann's Ph.D. thesis (1993).

Hyatt introduced *Dynamic Tree Splitting* (DTS) in his Ph.D. thesis (1988). One processor is given the root position and the others must try to find a processor that has work to steal. If a processor has work, DTS hands out a branch from the lowest type 3 node. Type 1 nodes that have bound information (i.e. the leftmost child has been evaluated) are considered as type 3 nodes. Failing this, the processor will hand out any node from a type 1 node that does not have any bound information. Finally, the processor would hand out branches from type 2 nodes that have not been pruned after the first child has been completely evaluated. This makes the algorithm similar (in parallelization and synchronization characteristics) to YBWC*, but allocates the work in a different order than YBWC* would.

In the implementation, split points were placed in shared memory so that other processors had an opportunity to take branches without a processor continually looking for work to do. Using a Sequent Balance 21000, DTS generated an average speed-up of 8.81 over 16 processors.

Bound-and-Branch (Ferguson and Korf, 1988) is a processor allocation scheme in the Distributed Tree Search framework – a general framework for distributed search – to search $\alpha\beta$ trees generated by an Othello program. If no cutoff bound exists at a node, all processors are assigned to the first child to generate a cutoff bound as quickly as possible. If a cutoff bound exists, or has been established by completing the search of the first child, the processors are allocated in a breadth-first manner to all remaining children. Effectively, this scheme gives the same parallelism and synchronization pattern described in the YBWC* algorithm. For the Bound-and-Branch processor allocation scheme, Ferguson and Korf get speedups of 12 while studying Othello trees using a 32-processor hypercube. The search is facilitated by a distributed “game-tree representation” which is similar to a transposition table, and iterative deepening.

Hsu described a *queued processor array model* for implementing a parallel $\alpha\beta$ algorithm within the second version of Deep Thought (1990). The host workstation traverses the tree according to the algorithm until the parallelization horizon is reached. The subproblems are then placed on to a queue that can be accessed by a large number of specialized VLSI processors. All of the processors are connected by the same bus to this queue. The processors take away the subproblems placed on the queue, run the silicon-encoded $\alpha\beta$ routine on the chip, and return the results to another queue on the bus that goes in to the host processor. The results are then added to the tree representation in the host computer.

Hsu also introduced the *delayed branch tree expansion* (DBTE) algorithms in his thesis. These algorithms generate two queues of nodes. The first queue is a set of nodes that correspond to the $\alpha\beta$ minimal tree, in a left-to-right order. The second queue contains nodes that are not in the $\alpha\beta$ minimal tree, because of poor move ordering at type 2 nodes. The minimal tree queue is used only when the queue of additional work at failed type 2 nodes is empty. This setup allows for parallelism at type 1 and 3 nodes.

There is a family of DBTE algorithms based on the choice of CUT nodes to re-expand. One of the algorithms, the Leftmost First algorithm, is shown to be asymptotically optimal on best-first trees as well as dominating weak $\alpha\beta$ (a version of the $\alpha\beta$ algorithm which only has shallow cutoffs). The Leftmost First algorithm causes synchronization to occur at bad type 2 nodes. Simulations report that a speedup of 350 with 1000 processors is possible, once the machine is completely constructed.

Lu (1993) implemented a pair of improvements to the basic PV-Split algorithm for use in the checkers program CHINOOK. To prevent starvation when exploring checkers trees, which have an average branching factor of 3 instead of the 38 found in chess trees, *frontier splitting* was proposed and tested. Frontier splitting creates multiple split nodes further up the variation being explored by the Controller process as they are required. This is different than algorithms like YBWC*, which concentrate on creating new split nodes underneath the current split node. Split nodes are created first at all nodes and only at cut nodes when there is no parallelism left at any all nodes. The drawback is that the search might be started without any bound information. Essentially, this removes the synchronization for a given depth search, and allows for parallelism at any node within the tree.

An implementation of dynamic load balancing, similar to the EPVS method, was also presented. *Straggler preemption* gathers a group of idle processors and assigns them to a subtree that has been worked on by one processor for a long period of time. Both of the improvements were tested and the speedup on the test set improved from 1.92 on 16 processors for the basic PV-Split algorithm to 3.31 on 16 processors. The average branching factor of the trees being explored was 2.78; although the magnitude of the increase was small, the fact that the speedup is larger than the branching factor is significant.

David's $\alpha\beta$* (1993) is a new type of architecture for dealing which requires a shared transposition table. All processors start at the root of the tree and start travelling down the tree. However, the processors explore different parts of the tree based on results from the shared transposition table. Each entry contains a counter of how many processors are exploring the subtree rooted at that node. Thus, the processor can discover which nodes have been evaluated.

A depth-limited search is executed at non-PV nodes to determine whether the node is a type 2 or a type 3 node. Type 2 nodes are searched sequentially, as in other algorithms. At type 3 nodes, the number of processors allowed to explore a subtree is limited by a constant factor of the number of processors that are currently at the type 3 node. For the purposes of $\alpha\beta$*, once the leftmost child of a type 1 node has been evaluated, the type 1 node effectively becomes a type 3 node. Once a processor has visited a node, it may not go back above that node until the node is evaluated (i.e. the correct $\alpha\beta$ information is known about the value at that node). This means that as soon as one processor has "evaluated" the root, the search is completed. One advantage of the $\alpha\beta$* algorithm is the parallel code involves only a small number of changes to the sequential code (Weill, 1995). In his thesis, David achieved a speedup of 6.5 on 16 Transputers. 8 of the Transputers were used to control the shared transposition table, while the other 8 were used as tree searchers.

CABP is an algorithm by Cung (1994), based on the Deep Thought design presented by Hsu (1990). The algorithm was designed for a shared memory system, and maintains a shared "score tree" for the entire game-tree and the two lists of work: critical nodes one ply above the leaves, and non-critical children of failed cut-nodes. At failed cut-nodes, the non-critical children are added to the list k at a time. (In the simulation results given in the thesis, $k = 1$.) In his Ph.D. thesis, Cung shows the CABP algorithm generating a speedup of 4.6 on strongly-ordered trees with a branching factor of 40 using 9 processors on a Sequent Balance 8000.

Kuszmaul (1994) presents *Jamboree search* in his Ph.D. thesis as an algorithm for testing MIMD scheduling algorithms on the CM-5. Jamboree search is a parallelization of NegaScout search which behaves with only a few minor differences to the work done by Feldmann *et. al* on the Young Brothers Wait algorithm. In the Young Brothers Wait algorithm, when a subtree is given to a processor and the search fails high, the slave processor immediately proceeds to work out the value with the full search window without informing the master processor. In Jamboree search, a fail high value is returned to the master processor. This prevents any younger subtrees from executing a full window search until the new bound α can be established by a full window search.

Weill introduced an improvement to the $\alpha\beta^*$ algorithm in his Ph.D. thesis (1995). Weill suggested and tested a decision method based on the Young Brothers Wait, instead of the depth-limited search and constant factor at type 3 nodes tested by David. At any node in the $\alpha\beta^*$ algorithm, if the leftmost child isn't evaluated, all processors must evaluate the leftmost child. Once the leftmost child is evaluated, processors are allocated to non-evaluated idle children first, and then allocated in a balanced manner to the other non-evaluated children in the tree. Although both $\alpha\beta^*$ and YBWC* use the same decision method for allowing or denying parallelism, $\alpha\beta^*$ uses a shared transposition table to keep the processors working on different parts of the tree, while YBWC* uses master-slave relationships.

In a later paper (Weill, 1996), the combined method was called *Alpha-Bêta Distributé avec Droit d'Aînesse*, or ABDADA. He showed that ABDADA yields greater speedups than YBWC on a CM-5 when studying chess trees. ABDADA also yields similar speedups to YBWC when studying $\alpha\beta$ trees generated by an Othello program.

Dynamic Multiple Principal Variation Splitting (DM-PVSplit) is a variation of the PV-Split algorithm that allows for greater parallelism near the start of a search (Marsland and Gao, 1995). To understand the algorithm, it is necessary to define the *PV set*. The root is a member of the PV set. At subsequent levels, nodes are part of the PV set if the parent is a member of the PV set, and they are generated by the first k candidate moves in the move list of the parent. The determination of k is given by a function, not a fixed number. Thus, the PV set is a right-pruned version of the game-tree. An appropriate choice of the heuristic guess allows for greater parallelism without adversely increasing the search overhead, since the correct move at a PV set node is highly likely to appear in its PV set children. By always selecting only one candidate move, DM-PVSplit generalizes into PV-Split. Since the PV set does not respect the structure of the minimal tree, the last two columns in Table 2 reflect this by referring to the PV set, and not Knuth and Moore's classification of minimal tree nodes.

The algorithm is designed for use on strongly ordered trees. In the paper, Marsland and Gao show the results of an experiment where DM-PVSplit generates a speedup of approximately 32 over 64 processors, using an artificially generated tree of width 32 and depth 8.

Asynchronous Parallel Hierarchical Iterative Deepening is an attempt to extend the ideas of Newborn's UIDPABS algorithm for use on a network of workstations, where the cost of communication is prohibitive (Brockington and Schaeffer, 1996). Instead of using a single ply of the game-tree to generate work lists, APHID uses a truncated game-tree of user-defined depth, and all leaves of the truncated game-tree are allocated to slave processors. The master processors within the hierarchy continually search the truncated game-tree to add, delete and update the priority of work for the slaves. The slave processors search their work lists with minimal synchronization from the master. The slaves determine their own work schedule and inform the master processors of the search results. Slave processors can continue searching the next level of the iterative deepening search speculatively if they have no work left to do at the current level.

The APHID algorithm has been implemented as a game-independent library that can be easily inserted into a sequential game-playing program. The library has been tested on both KEYANO (an Othello program) and THETURK (a chess program). At time of writing, the APHID algorithm achieves speedups of 8.41 and 6.02 for KEYANO and THETURK (respectively) on a network of 16 Sparc-2 workstations.

3 OTHER SEARCH PARADIGMS FOR PARALLEL GAME-TREE SEARCH

Although the emphasis has been on $\alpha\beta$-based methods, there are other search strategies that can be used to generate the minimax value. They have been subdivided into three subsections. The first will deal with parallel algorithms based on the original formulation of $SSS*$. The second subsection will deal with the ER method, while the final subsection will deal very briefly with purely theoretical models for game-tree search.

3.1 Parallel Search based on $SSS*$

Stockman (1979) introduced the $SSS*$ algorithm. Initially, it was believed that the algorithm dominated $\alpha\beta$ in the sense that $SSS*$ will not search a node if $\alpha\beta$ did not search it. A perceived problem with the algorithm is that a list structure (the `OPEN` list) must be maintained, which could grow to $b^{d/2}$ elements, where b is the branching factor and d is the depth of the tree to be searched. This space requirement was, at the time, considered to be too large for a practical chess-playing program. Furthermore, even if the space requirement was not a problem, the maintenance of the `OPEN` list slowed down the algorithm to make it slower than $\alpha\beta$ in practice.

A theoretical parallel $SSS*$ algorithm was proposed (Campbell, 1981; Campbell and Marsland, 1983), based on breaking the tree into stages to reduce the cost of maintaining the `OPEN` list. At the end of a stage in the search, the node would be handed to a slave processor in the tree hierarchy. This limited the depth d, thus preventing the `OPEN` list from becoming too large. The staged $SSS*$ algorithm is shown to be marginally faster than either tree splitting or PV-Split on randomly ordered trees in Campbell's work.

Leifker and Kanal (1985) propose the HYBRID algorithm that is based on the problem heap from $SSS*$, but contains no details pertaining to an implementation. Vornberger and Monien presented the results of a parallel $SSS*$ algorithm (1987), but the results were disappointing when compared to the parallel $\alpha\beta$ algorithm (later to be called "Young Brothers Wait"). Their implementation of parallel $SSS*$ on a local area network of PCs had a search overhead of over 300 percent when using 16 processors. Diderich described an implementation of Synchronized Distributed State Space Search ($SDSSS*$) (1992) and achieved a speedup of 11.40 using 32 processors, searching a 5 ply tree with a branching factor of 16.

There are some other $SSS*$ algorithms by Hiromoto *et al.* (1987), Kraas (1990), and Shinghal and Shved (1991), and all of the work deals with how to parallelize work based on the `OPEN` list. Although the work has shown some promise in eliminating the difficulties of dealing with an ordered problem heap, all of the work on handling the `OPEN` list has been made obsolete. This is due to the revelation that $SSS*$ can be implemented as a series of null-window $\alpha\beta$ calls, using a transposition table in the place of the `OPEN` list (Plaat, 1995).

3.2 Parallel Search based on ER

Steinberg and Solomon (1990) presented the ER method of searching game-trees. ER stands for Evaluate-Refute, and it attempts to evaluate some mandatory work before attempting to refute the other moves within the tree. At a node to be evaluated (an e-node) within the tree, the ER algorithm evaluates the elder grandchildren (concurrently, if possible), and then chooses the child with largest elder grandchild to be the e-child. This e-child is evaluated, and then the other children of the e-node are refuted. The method is less efficient at searching trees than the $\alpha\beta$ algorithm since it misses some deep cutoffs. Furthermore, the algorithm was not tested with iterative deepening or minimal windows when refuting e-nodes.

The ER method and PV-Split were implemented as problem-heap algorithms on a Sequent Symmetry multiprocessor. Steinberg and Solomon found that they achieved a better efficiency with the parallel ER algorithm than with PV-Split. The ER method achieved a speedup of 10 with 16 processors, and a speedup of 14.7 with 27 processors.

3.3 Theoretical Parallel Search Methods

There are a number of theoretical algorithms that, to the author's knowledge, haven't been programmed. Karp and Zhang (1989) and Althöfer (1993) have proposed algorithms that will yield linear speedups in the number of processors, given trees of $O(N \log N)$ or $O(N)$ depth, respectively. Broder *et al.* (1990) have shown that for any parallel tree searching algorithm, there exists a tree instance that does not run in polylog parallel run-time. Broder *et al.* also show and prove an upper bound on the run-time of the ParHope algorithm.

4 CONCLUSIONS

The taxonomy given in section 2 shows that there are a number of algorithms which are similar to one another. There are only subtle differences between a large number of the $\alpha\beta$-based algorithms. Most of these differences are due to the different architectures and game trees studied. This comparison is easily made through the separation of the implementation and algorithmic details.

As the number of processors available to the programmers has increased, dynamic algorithms have taken the place of the earlier algorithms which offered limited parallelism on a large number of processors. It is the author's belief that speculative parallelism is necessary to yield near-linear performance on parallel architectures, and is the subject of on-going research at the University of Alberta.

5 ACKNOWLEDGEMENTS

The author would like to thank Yngvi Bjornsson, Rainer Feldmann, Yaoqing Gao, Andreas Junghanns, Tony Marsland, Aske Plaat, Jonathan Schaeffer, Jean-Christophe Weill and the editorial staff of the ICCA Journal for their helpful comments on the revisions of this paper. The author is being funded by an NSERC Postgraduate Scholarship, for which the author is extremely grateful.

The author would appreciate any corrections, suggestions and improvements that you may have.

References

Akl, S. G., Barnard, D. T., and Doran, R. J. (1982). Design, Analysis and Implementation of a Parallel Tree Search Algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, No. 2, pp. 192–203.

Althöfer, I. (1993). A Parallel Game Tree Search Algorithm with a Linear Speedup. *Journal of Algorithm*, Vol. 15, pp. 175–198.

Bal, H. E. and van Renesse, R. (1986). A Summary of Parallel Alpha-Beta Search Results. *ICCA Journal*, Vol. 9, No. 3, pp. 146–149.

Baudet, G. M. (1978). *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA. Available as Tech. Rept. CMU-CS-78-116.

Brockington, M. G. and Schaeffer, J. (1996). APHID Game-Tree Search. Presented at *Advances in Computer Chess 8*, Maastricht.

Broder, A., Karlin, A., Raghavan, P., and Upfal, E. (1990). On the Parallel Complexity of Evaluating Game-Trees. Technical Report RR RJ 7729, IBM T. J. Watson Research Center, Yorktown Heights, New York.

Brudno, A. L. (1963). Bounds and Valuations for Abridging the Search for Estimates. *Problems of Cybernetics*, Vol. 10, pp. 225–241. Translation of Russian original in *Problemy Kibernetiki*, 10:141–150, May 1963.

Campbell, M. S. (1981). *Algorithms for the Parallel Search of Game Trees.* M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Alta. Available as Tech. Rep. TR 81-8, Dept. of Computing Science.

Campbell, M. S. and Marsland, T. A. (1983). A Comparison of Minimax Tree Search Algorithms. *Artificial Intelligence*, Vol. 20, pp. 347–367.

Ciancarini, P. (1994). Distributed Searches: A Basis for Comparison. *ICCA Journal*, Vol. 17, No. 4, pp. 194–206.

Cung, V.-D. (1994). *Contribution à l'Algorithmique Non Numérique Parallèle: Exploration d'Espaces de Recherche.* Ph.D. thesis, Université Paris VI.

David, V. (1993). *Algorithmique parallèle sur les arbres de décision et raisonnement en temps contraint - Etude et application au minimax.* Ph.D. thesis, ENSAE, Toulouse, France.

Diderich, C. G. (1992). Evaluation des Performances de l'Algorithme SSS* avec Phases de Synchronisation sur une Machine Parallèle à Mémoires Distribuées. Technical Report LITH-99, Swiss Federal Institute of Technology, Lausanne, Switzerland.

Feldmann, R. (1993). *Spielbaumsuche mit massiv parallelen Systemen.* Ph.D. thesis, Universität-Gesamthochschule Paderborn, Paderborn, Germany.

Feldmann, R., Monien, B., Mysliwietz, P., and Vornberger, O. (1989). Distributed Game Tree Search. *ICCA Journal*, Vol. 12, No. 2, pp. 65–73.

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

Ferguson, C. and Korf, R. E. (1988). Distributed Tree Search and its Application to Alpha-Beta Pruning. In *Proceedings of AAAI-88*, pp. 128–132, Saint Paul, MN.

Finkel, R. A. and Fishburn, J. P. (1982). Parallelism in Alpha-Beta Search. *Artificial Intelligence*, Vol. 19, No. 1, pp. 89–106.

Hiromoto, U., Masafumi, Y., Masaharu, I., and Toshihide, I. (1987). Parallel Searches of Game Trees. *Systems and Computers in Japan*, Vol. 18, No. 8, pp. 97–109.

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

Hyatt, R. M. (1988). *A High-Performance Parallel Algorithm To Search Depth-Frist Game Trees.* Ph.D. thesis, University of Alabama, Birmingham, U.S.A.

Hyatt, R. M., Suter, B. W., and Nelson, H. L. (1989). A Parallel Alpha/Beta Tree Searching Algorithm. *Parallel Computing*, Vol. 10, No. 3, pp. 299–308.

Karp, R. M. and Zhang, Y. (1989). On Parallel Evaluation of Game Trees. In *Proceedings of SPAA '89*, pp. 409–420, New York, NY. ACM Press.

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 3, pp. 293–326.

Kraas, H.-J. (1990). *Zur Parallelisierung des SSS*-Algorithmus.* Ph.D. thesis, TU of Braunschweig, Braunschweig, Germany.

Kuszmaul, B. C. (1994). *Synchronized MIMD Computing.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

Leifker, D. B. and Kanal, L. N. (1985). A Hybrid SSS*/Alpha-Beta Algorithm for Parallel Search of Game Trees. In *Proceedings of IJCAI-85*, pp. 1044–1046.

Lindstrom, G. (1983). 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.

Lu, C.-P. P. (1993). *Parallel Search of Narrow Game Trees.* M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Canada.

Marsland, T. A. (1983). Relative Efficiency of Alpha-Beta Implementations. In *Proceedings of IJCAI-83*, pp. 763–766, Karlsruhe, Germany.

Marsland, T. A. and Campbell, M. S. (1982). Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, Vol. 14, No. 4, pp. 533–551.

Marsland, T. A. and Gao, Y. (1995). Speculative Parallelism Improves Search? Technical Report 95–05, Department of Computing Science, University of Alberta, Edmonton, Alta.

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

Marsland, T. A. and Popowich, F. (1985). Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-7, No. 4, pp. 442–452.

Newborn, M. M. (1985). A Parallel Search Chess Program. In *Proceedings of the ACM Annual Conference*, pp. 272–277.

Newborn, M. M. (1988). Unsynchronized Iterative Deepening Parallel Alpha-Beta Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-10, No. 5, pp. 687–694.

Plaat, A., Schaeffer, J., Pijls, W., and de Bruin, A. (1995). Best-First Fixed-Depth Game-Tree Search in Practice. In *Proceedings of IJCAI-95*, volume 1, pp. 273–279, Montreal, Quebec.

Plaat, A., Schaeffer, J., Pijls, W., and de Bruin, A. (1996). Best-First Fixed-Depth Minimax Algorithms. To appear in *Artificial Intelligence.*

Reinefeld, A. (1983). An Improvement to the Scout Tree-Search Algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14.

Schaeffer, J. (1989). Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, Vol. 6, No. 2, pp. 90–114.

Shinghal, R. and Shved, S. (1991). Proposed Modifications to Parallel State Space Search of Game Trees. *International Journal of Pattern Recognition and Artificial Intelligence*, Vol. 5, No. 5, pp. 809–833.

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

Stockman, G. C. (1979). A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence*, Vol. 12, pp. 179–196.

Vornberger, O. and Monien, B. (1987). Parallel Alpha-Beta versus Parallel SSS*. In *Proceedings of the IFIP Conference on Distributed Processing*, pp. 613–625. North Holland.

Weill, J.-C. (1995). *Programmes d'échecs de championnat: architecture logicielle synthèse de fonctions d'évaluations, parallélisme de recherche.* Ph.D. thesis, Université Paris 8.

Weill, J.-C. (1996). The ABDADA Distributed Minimax-Search Algorithm. *ICCA Journal*, Vol. 19, No. 1, pp. 3–16.