University of Alberta

Library Release Form

Name of Author: Mark Gordon Brockington

Title of Thesis: Asynchronous Parallel Game-Tree Search

Degree: Doctor of Philosophy

Year this Degree Granted: 1998

Mark Gordon Brockington
10649 85th Avenue, Apt. 201
Edmonton, Alberta
Canada, T6E 2K6

Date: ____________________

University of Alberta

Asynchronous Parallel Game-Tree Search

by

Mark Gordon Brockington

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Computing Science

Edmonton, Alberta
Spring 1998

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Asynchronous Parallel Game-Tree Search** submitted by Mark Gordon Brockington in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Jonathan Schaeffer

Joseph Culberson

T. Anthony Marsland

John Samson

Murray Campbell

Monroe Newborn

Date: ____________________

Abstract

Tree searching is a fundamental and computationally intensive problem in artificial intelligence. Parallelization of tree-searching algorithms is one method of improving the speed of these algorithms. However, a high-performance parallel two-player game-tree search algorithm has eluded researchers. Most parallel game-tree search approaches follow synchronous methods, where the work is concentrated within a specific part of the tree, or a given search depth. This thesis shows that asynchronous game-tree search algorithms can be as efficient as synchronous methods in determining the minimax value.

A taxonomy of previous work in parallel game-tree search is presented. A theoretical model is developed for comparing the efficiency of synchronous and asynchronous search algorithms under realistic assumptions. APHID, a portable parallel game-tree search library, has been built based on the asynchronous parallel game-tree search algorithm proposed in the comparison. The library is easy to implement into a sequential game-tree searching program. APHID has been added to four programs written by different authors. APHID yields better observed speedups than synchronous search methods for an Othello and a checkers program, and yields comparable observed speedups to synchronous methods on two chess programs.

Acknowledgements

I have been told if you do not thank your supervisor first, he will not sign any nice reference letters for you. A big thanks to Jonathan Schaeffer, both as a supervisor and a friend.

Thank you to the other members of my thesis committee, Tony Marsland, Joe Culberson and John Samson, for many discussions and keeping me on track. Thanks to my external examiners, Murray Campbell and Monroe Newborn, for some extremely valuable feedback on the original version of the thesis.

The author would like to acknowledge the people who have made the experiments in this thesis possible. The author would like to thank Paul Masiar and Marc Nolte at the SGI office in Calgary for access to the 64-processor SGI Origin 2000 system in Eagan, Minnesota. The author would also like to thank Charles Leiserson, Aske Plaat and Boston University for access to a lab of UltraSPARC machines and the 32-processor SGI Origin 2000 system.

Thank you to the other members of the University of Alberta GAMES Group, past and present, for many helpful discussions and some valuable proofreading: Darse Billings, Yngvi Björnsson, Yaoqing Gao, Andreas Junghanns and Denis Papp.

Thank you to Michael Buro for many opportunities to play against Logistello, and the knowledge that I still need to do some more work. Thank you to Igor Durdanovic, Louis Geoffroy, Kevin Hall, David Parsons, Martin Piotte, Colin Springer, David Summers, Jean-Christophe Weill, and the other regulars on the Internet Othello Server for providing challenging opponents for Keyano over the last four years.

Thanks to Annette Gottstein, Sean Halliday, Kathleen Kippen, Randal Kornelsen, Steve MacDonald, David McCaughan, Diego Novillo, Ian Parsons, Kevin Wiebe, David R! Woloschuk, Debbie and Jerry Yee, and all of the other graduate students and friends who have had to put up with my fascination with computer games for the last five years in Edmonton.

Thank you to Savak, for crushing any aspiration I might have had about being a professional musician. Thank you to my family who have supported and encouraged me over the years.

Last and definitely not least, a big thank you to my wife, Jennifer.

Brought to you by the letters ζ and ξ.

Mark Brockington November 18, 1997

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

1.1.1 Artificial Intelligence and Games

Making computers play games in a skillful manner, comparable to that of a good human player, is an interesting and challenging problem that has attracted the attention of many computer scientists over the last forty years. It is viewed as an interesting test bed for other fields of artificial intelligence because of the well-defined rules and the wealth of heuristic information that can be applied to achieve "intelligent" play.

Many researchers have explored the development of algorithms to play games such as checkers, backgammon, Othello [1], Monopoly [2], Scrabble [3], and Go. However, the majority of the game-playing research in the western world still focuses on chess.

Chess can be considered as one of the original *Drosophila* [4] of artificial intelligence [67]. In the early 1950s, it was believed that a computer that could play chess at a master level would also be able to solve economic and philosophical problems. Unfortunately, chess can be programmed algorithmically with little insight into life,

[1] Othello is a registered trademark of Tsukuda Original, licensed by Anjar Co.

[2] Monopoly is a registered trademark of Parker Brothers.

[3] Scrabble is a registered trademark of Milton Bradley Company, a division of Hasbro, Inc.

[4] *Drosophila melongaster* is the scientific name for the common fruit fly. It is popular in biological experiments because of its short life cycle.

the universe, and everything. In 1997, a home computer can play a game of chess that is capable of beating 99.9% of all humans by using a tree searching algorithm and a few heuristics to guide the evaluation function. In May 1997, Deep Blue, an IBM machine utilizing custom hardware and massive parallelism, defeated Garry Kasparov, the World Chess Champion, in a 6-game match. Although we have made great advances in the field of computer chess, both the home computer and Deep Blue are a long way from understanding Descartes.

The research in computer chess is widely applicable to other two-player games with perfect information, such as Othello and checkers. The $\alpha\beta$ algorithm can be used for any two-player zero-sum game with perfect information. Furthermore, almost all of the research into making the $\alpha\beta$ algorithm search a chess tree more efficiently can be applied directly to other games. In return, other games have contributed many alternative methods of handling some search problems in the game of chess.

A game-playing program that can outsearch its opponent has a high probability of winning. It has been shown that there is a strong correlation between the search depth and the relative strength of chess [90], checkers and Othello programs. Thus, game programmers continually attempt to search larger game trees while staying within the time constraints imposed by the rules of the game.

1.1.2 Parallel Algorithms for Game-Playing

A sequential game-tree search algorithm uses only a single processor to search the game tree. One method of achieving greater search depths involves using many processors to speed up the search. A parallel game-tree search algorithm uses multiple processors simultaneously to determine the value of the game tree.

In the field of parallel game-tree search, the most important measure of performance is speed: How fast can we search the game tree in parallel in comparison to our sequential search? We can observe how much faster a program searches a game tree in parallel than sequentially. This ratio is usually defined as the observed speedup. If we divide the observed speedup by the number of processors, this is referred to as the observed efficiency of the parallel algorithm. The ideal parallel game-tree search algorithm has a high parallel efficiency on a large number of processors.

For the game of chess, almost all of the researchers have used synchronous parallel game-tree search algorithms to speed up sequential $\alpha\beta$-based algorithms. Synchronous parallel game-tree search algorithms force work on one part of the tree to be completed before work on the rest of the tree can be carried out. One could say that a synchronous algorithm has global synchronization points during the game-tree search that all processors must reach before any process is allowed to proceed beyond the synchronization point.

In some synchronous algorithms, the work is synchronized at every choice along the hypothesized best move sequence, commonly known as the principal variation. In all synchronized algorithms, the work is synchronized at the root of the game tree. Most game-playing programs use iterative deepening; the program searches the game tree to positions d moves ahead (commonly known as a *d-ply game tree*) before searching the game tree $d+1$ moves ahead. The synchronization at the root of the game tree implies that the d-ply game-tree search must be complete before any process can proceed to the $(d+1)$-ply game-tree search.

The advantage of the synchronous parallel approaches is that the parallel algorithm can use the value of the principal variation without uncertainty, in the same way that a sequential game-tree search algorithm uses the value of the principal variation. These synchronous parallel algorithms are successful at keeping the amount of work required to determine the value of the game tree near the sequential search size, assuming that processors are able to share search information in an efficient manner. The common method of doing this is a shared table, which stores all of the important search results from each processor.

However, there are fundamental problems with synchronous algorithms for parallelizing $\alpha\beta$-based game-tree search. The first problem is that there are many times when there is insufficient parallelism to keep all the processors busy. If we have more processors than work to do before a synchronization point, some processors must go idle. This idle time is not that important on a small number of processors, but increases in magnitude (and importance) as the number of processors increases. This problem is exacerbated in games that have a smaller average number of move choices than chess, such as Othello and checkers. For example, the best observed speedup for a state-of-the-art checkers program using a synchronous parallel algorithm is only 3.32 on 16 processors, with further processors being of no benefit.

A second problem is that all of the synchronous parallel $\alpha\beta$-based algorithms require an efficient implementation of a shared table between the processes to achieve a high parallel efficiency. Synchronized algorithms in the domains of computer chess and checkers exhibit poor parallel efficiency without an efficient shared table implementation, due to the requirement of replicating search results on multiple processes. Most of the synchronous algorithms are tested on machines with fast networks and slow processors, while the majority of parallel architectures available to researchers today are distributed networks of workstations with much faster CPUs relative to the network speed. On the latter systems, creating a shared table with distributed memory is not as efficient, and the efficiency portrayed in the literature is not achievable.

The third problem is that the introduction of these algorithms into existing sequential code can force serious implementation difficulties. In general, one wants the parallel algorithm and the sequential algorithm to agree on the game-tree value as well as the principal variation of the search. When we implement a parallel search algorithm in an existing sequential program, it can take a large amount of time to insert the algorithm into the program and verify that the parallel program is working in the same way that the sequential program does.

Newborn suggested the use of asynchronous search for parallelizing $\alpha\beta$-based algorithms [69]. Each process is given a subset of the moves from the root of the game tree to search independent of all other processes. When the time committed to the search has been exhausted, all of the results are combined and the move leading to the best game-tree value can be determined. At no point during the search are any of the processes waiting for another process. Newborn's UIDPABS algorithm is limited by the number of moves available at the root of the game tree, since processes are not allowed to co-operate on searching a single move from the root. The idea behind UIDPABS could be generalized to use massively parallel hardware efficiently.

The goal of this thesis is to answer the question: In the area of game-tree search, can asynchronous parallel algorithms outperform synchronous parallel algorithms?

1.2 Contributions

In this thesis, we will show that it is possible for asynchronous parallel algorithms to outperform synchronous parallel algorithms in the area of game-tree search.

A taxonomy of parallel $\alpha\beta$-based game tree search algorithms is presented. The taxonomy shows that many synchronous algorithms that seem dissimilar on the surface are, in fact, using the same underlying algorithm. The taxonomy also shows that asynchronous search algorithms, in the field of game-tree search, have been overlooked aside from Newborn's UIDPABS algorithm.

We develop a model for comparing a typical synchronous game-tree search algorithm to an asynchronous game-tree search algorithm. We show that it is possible for the asynchronous algorithm to outperform the synchronous algorithm on realistic game trees.

The APHID (Asynchronous Parallel Hierarchical Iterative Deepening) algorithm has been developed and is presented in this thesis. APHID is a new asynchronous $\alpha\beta$-based search algorithm based on the proposed asynchronous algorithm used in the theoretical model.

APHID keeps all processors busy by allowing the processes to schedule work for themselves, with minimal advice from an overseeing process. APHID attempts to keep communication between the overseer and its child processes to a minimum, allowing the system to exhibit reasonable performance on both a network of workstations and massively parallel hardware.

The APHID algorithm is written in a application-independent manner so that it may be inserted into many different game-tree searching programs to generate parallel game-tree searching programs. In this work, we exhibit implementations of APHID in many different programs:

- Chinook, the World Man-Machine checkers champion, developed by a team led by Jonathan Schaeffer,
- Crafty, Robert Hyatt's freeware chess program,
- TheTurk, a chess program written by Yngvi Björnsson and Andreas Junghanns, two graduate students at the University of Alberta, and
- Keyano, an Othello program written by the author.

For the message-passing between processes, APHID uses the Parallel Virtual Machine (PVM) library [35]. Thus, the system is portable across many different architectures. The APHID system has been tested on many different architectures over the last two years, including a network of Sun SPARCstation IPC workstations, an SGI Power Challenge array, a series of 8-processor Ultra-SPARC Enterprise machines, as well as 32 and 64-processor SGI Origin 2000 systems.

APHID has been written as a library of application-independent routines that can be linked into a game-playing program. The library uses a small API to send information to and from the the sequential application. Instead of forcing the user to rewrite existing search code to fit the parallel algorithm, we place a small number of calls to APHID routines in the sequential application. The library achieves access to application-dependent functions via a number of call-back functions that the programmer must provide.

The library allows the APHID algorithm to be integrated into an existing minimax-based game-tree search algorithm with little effort. We used the APHID library to generate all of the parallel applications tested in this thesis. Each of the parallel implementations took less than a day of programming time to achieve a parallel program that executed in the same way as the sequential program, and a few days of additional tuning to achieve the presented parallel speedups.

A synchronous parallel game-tree search algorithm generates greater speedups in games that have a greater number of move choices in each position. However, APHID's results are independent of the average number of move choices. This allows APHID to outperform synchronous algorithms on game trees with a small number of move choices. For example, CHINOOK yields an observed speedup of 14.35 on 64 processors when combined with the APHID library. This is four times the speedup achieved using a highly-tuned synchronous algorithm in the same application. KEYANO, an Othello program, yields observed speedups with APHID on 64 processors that are 50% larger than the equivalent speedups for YBWC*, the synchronous game-tree search algorithm that has exhibited the best parallel efficiency for any computer game on more than 64 processors. The observed speedups with APHID in the two chess programs are 12.96 (THETURK) and 16.56 (CRAFTY) on 32 processors. The speedups are equivalent to many reported speedups in the game of chess for synchronous parallel algorithms.

The analysis of earlier results has led to numerous insights into the structure of the games studied. For example, chess and checkers yield low speedups without using shared tables. One of the key differences between these games and Othello is the relative need for a shared table. Through our experience with APHID, we will show why the good parallel speedups in computer chess have an efficient shared table implementation. We will also show 16-processor speedups for all of the applications on local, distributed and shared tables.

The final contribution of this research is to give other researchers a starting point from which to advance the study of asynchronous game-tree search algorithms. The code and methods used in APHID are described within this document, and the APHID library is freely available to any interested game-tree search researcher at the APHID home page[5].

[5] `http://www.cs.ualberta.ca/~games/aphid/`

1.3 Organization

Chapter 2 gives a summary of game-tree searching techniques that are employed on a single processor. The chapter is intended to be a brief survey of the field, and is for the reader who knows little about searching game trees. Chapter 3 contains a summary of the previous work done in parallel tree-search algorithms. An explanation of some parallel terminology required to comprehend the summaries is also given. Chapter 4 contains a theoretical comparison of a typical synchronous $\alpha\beta$ algorithm to an asynchronous $\alpha\beta$ algorithm.

Chapter 5 describes the APHID algorithm in detail, along with some illustrative examples of how to add APHID into existing sequential game-tree search code. Chapter 6 shows the experimental results of implementing APHID in the four applications. Chapter 7 presents the conclusions, and poses some additional questions that still need to be solved.

The test positions used for each of the applications can be found in Appendix A. An in-depth description of the current interface to the APHID library can be found in Appendix B.

1.4 Publications

A large part of Chapter 3 has been published by the ICCA Journal [13]. Preliminary versions of the APHID library have been presented [15, 16]. However, the library has been changed substantially since the initial publication. Furthermore, the results have improved for all of the test domains used in the first series of tests.

Chapter 2

Sequential Game-Tree Search

Before one can understand how multiple processors can be used by a game-tree searching program, it is important to understand the mechanics of sequential game-tree searching programs. This chapter deals exclusively with sequential game-tree search, and how it is used in current practice. The emphasis will be on minimax-based and $\alpha\beta$-based algorithms, since these are the algorithms that can be parallelized with the APHID library. This chapter is intended as a brief introduction to the field, and not a substantive review of all research.

Section 2.1 is a brief introduction to the theory of two-player zero-sum games with perfect information. The motivation and evolution of the $\alpha\beta$ algorithm is given in Section 2.2. A non-exhaustive survey of additions and improvements to fixed-depth $\alpha\beta$ algorithms can be found in Section 2.3. Section 2.4 deals with algorithms and heuristics that change a fixed-depth game tree into a variable-depth game tree. Section 2.5 briefly describes other game-tree search algorithms (SSS*, B*, BPIP and conspiracy numbers), and Section 2.6 will give us the opportunity for some closing remarks on sequential game-tree search.

2.1 Game Theory and the Minimax Algorithm

Game theory deals with the mathematical analysis of competitive situations. The two fields where game theory are regularly employed are economics and military conflicts. We could view games as a contest with fixed rules that is decided by skill, strength or luck.

Von Neumann and Morgenstern's book on game theory [93] commences by partitioning games into many classes. The games we are interested in, such as chess, Othello and checkers, belong to a class of games known as *two-player zero-sum games with perfect information.*

Two-player games are those in which two competing forces are attempting to obtain their goals. A player could be a single person, an entire army, or a computer program. As long as all members of the force are attempting to achieve the same goal, they can be viewed as a single player. A *zero-sum* game indicates that when all payoffs are taken into account, the net sum must equal zero. In a zero-sum game, if one player wins, the other player must lose by an equivalent amount. If neither player wins anything, the game ends in a draw. A game with *perfect information* requires that all of the information about the current state of the game is visible to each player. Poker and Scrabble are examples of games with imperfect information, since some of the cards or tiles are hidden from view and cannot be used in formulating a strategy.

We can also simplify matters by stating that the game should have no chance moves – it does not rely on a probabilistic event to determine the moves available in the game. For example, backgammon and Monopoly can be considered as two-player zero-sum games with perfect information and chance moves, because the die roll determines the mobility of the game pieces.

Although many two-player zero-sum games with perfect information and no chance moves may seem dissimilar on the surface, they belong in the same class of games. Furthermore, strategies can be formulated using the same method. The optimal move for any position, given best play by the opponent, can be determined from a *game* *tree.*

The root of a game tree represents the current state of the game. Each node in the tree can have any number of child nodes. Each child of a node represents a new state after a legal move from the node's state. This continues until we reach a *leaf*, a node with no child nodes, in the game tree. We assign a payoff vector to each leaf in the game tree. In a generalized game tree, this payoff vector represents the *utility* of the final position to both players. In general, winning a game represents a positive utility for a player, while losing a game represents a negative utility. Since the game is a two-player zero-sum game, the utility for the first player must equal the negative of the utility for the second player. The utility for the side to move at the root of the tree is usually the only one given to save space.

In Figure 2.1, an example of a game tree for a game of Naughts and Crosses (or Tic-Tac-Toe) is given. Note that the two players take alternating turns at different levels of the tree. X moves at the root, while the opponent, O, moves at the first level below the root. A position is normally categorized by how many levels down in the game tree it is located. The common term for this is *ply*. The root is said to be at ply 0, while the immediate successors of the root are said to be at ply 1, *et cetera*.

The orientation of the game tree in Figure 2.1 is important to understanding the common terminology used in the literature. When we talk about descending a search tree, we are travelling away from the root of the game tree towards the leaves. Note that the ply values increase as we descend the game tree. Travelling upwards in a search tree is the opposite: we travel towards the root of the game tree and the ply values decrease.

Naughts and Crosses, like chess and checkers, has only three possible outcomes for a player: win, loss or draw. Normally, we assign the payoff of +1, 0 and -1 to a win, draw or loss for the player to move, respectively. These payoffs are given in Figure 2.1 at the bottom of each leaf position, with respect to the player with the crosses.

We will give names to each player to simplify our discussion. Let us call the player to move in the initial position Max and the opponent Min. At each node in the tree

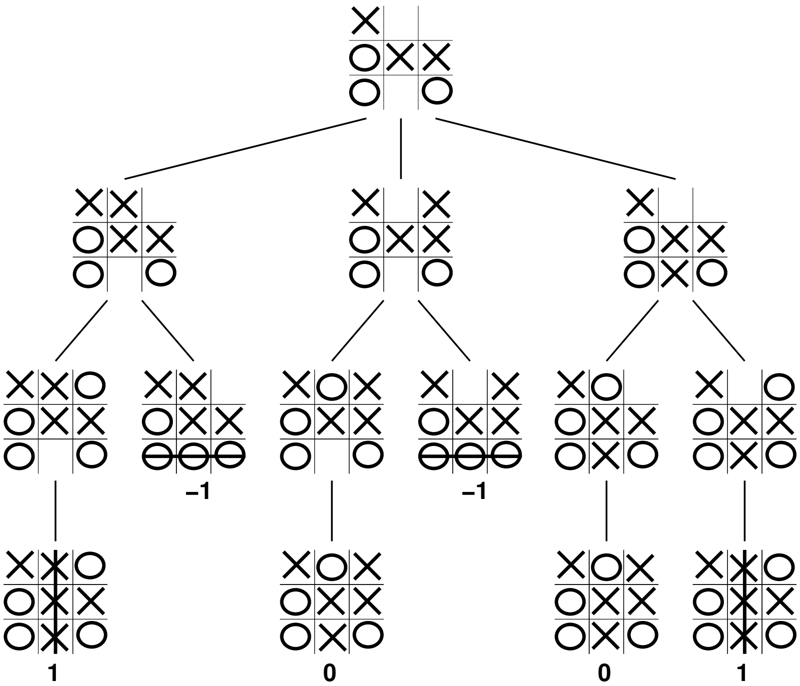


Figure 2.1: Example of Naughts and Crosses Game Tree

where Max has to move, Max would like to play the move that maximizes the payoff. Thus, Max will assign the maximum score amongst the children to the node where Max makes a move. Similarly, Min will minimize the payoff to Max, since that will maximize Min's payoff. The maximum and minimum scores are taken at alternating levels of the tree, since Max and Min alternate turns.

In this way, all nodes in the tree can be assigned a payoff or *minimax value*, starting from the leaves of the tree and moving up the tree towards the root. In Figure 2.2, we give minimax values for all nodes in our Naughts and Crosses game tree (Figure 2.1). These minimax values tell us what the best possible outcome for Max is in any position within the game tree, given that Min will do its best to foil Max's plans.

Once the root of the game tree has been assigned a minimax value, a *best move* for Max is defined as a move which leads to the same minimax value as the root of the tree. We can trace down the tree, always choosing moves that lead to the same minimax value. This path of moves gives us an optimal line of play for either player,

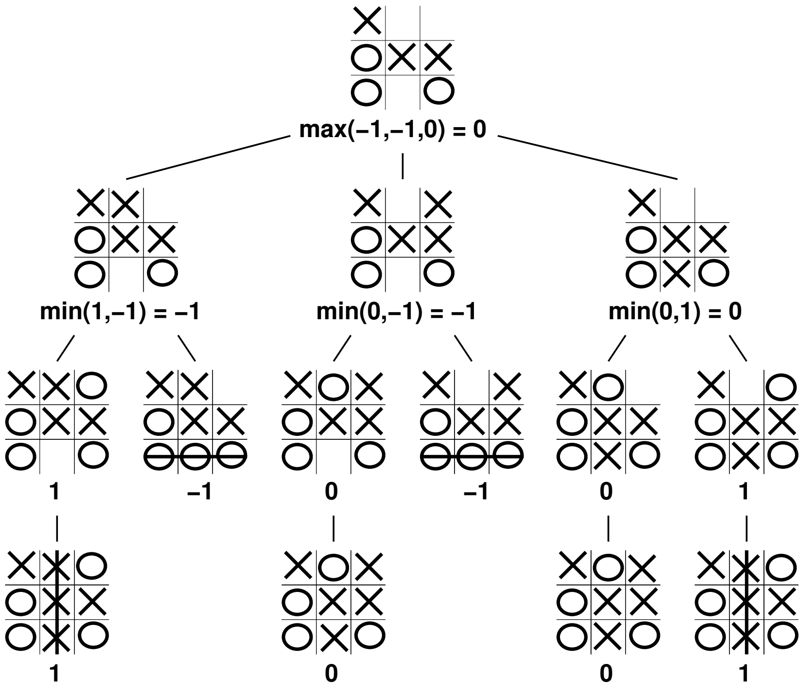


Figure 2.2: Game Tree with Minimax Values Added

and is known as a *principal variation*. Note that in our game of Naughts and Crosses, the side playing the Crosses will draw the game, but only if an X is played in the lower central square. Playing to either square in the top row can lead to a loss for the Crosses, if the opponent plays the best move.

To compute the minimax value of a position, we can use any algorithm which searches the whole game tree. A depth-first search will search each child of a node completely before exploring any other children. A breadth-first search will attempt to search all children of a node at the same time. A best-first search algorithm attempts to search and expand the "best" node at every step, irrespective of its location within the tree. A depth-first search uses less memory than a best-first or breadth-first tree search algorithm, so it is preferred in current game-tree search programs. In Figure 2.3, we show two C functions which are the basis of a recursive depth-first search of a game tree. By calling `Maximize` with a position `p`, we will get the minimax value of position `p` as the output of the function after the entire game tree has been searched.

In Figure 2.3, we have left out some of the details. For example, we have not

```
int Maximize(position p) {

   int numOfSuccessors;       /* total moves */
   int gamma;             /* current maximum */
   int i;                    /* move counter */
   int sc;      /* score returned by search */

   if (EndOfGame(p)) { return(GameValue(p)); }
   gamma = -∞;
   numOfSuccessors = GenerateSuccessors(p);
   for(i=1; i <= numOfSuccessors; i++) {
      sc = Minimize(p.succ[i]);
      gamma = max(gamma, sc);
   }
   return(gamma);

} /* Maximize */

int Minimize(position p) {

   int numOfSuccessors;       /* total moves */
   int gamma;             /* current minimum */
   int i;                    /* move counter */
   int sc;      /* score returned by search */

   if (EndOfGame(p)) { return(GameValue(p)); }
   gamma = +∞;
   numOfSuccessors = GenerateSuccessors(p);
   for(i=1; i <= numOfSuccessors; i++) {
      sc = Maximize(p.succ[i]);
      gamma = min(gamma, sc);
   }
   return(gamma);

} /* Minimize */
```

Figure 2.3: An Implementation of the Minimax Algorithm

defined what a position is, since this is game-dependent. `min` and `max` are functions which choose the smallest or largest of the two parameters, respectively. There are three additional functions that would be required to implement the minimax search: (1) `EndOfGame`, which determines whether the game is over at the input position, returning a Boolean value of `TRUE` if the game is over; (2) `GameValue`, which accepts a position as a parameter, determines who has won the game, and returns the payoff with respect to the player Max; and (3) `GenerateSuccessors` which generates an array of successor positions (`p.succ[]`) from the input position, and returns the number of successors to the calling procedure.

Note that `Maximize` and `Minimize` recursively call one another until a position is reached where the `EndOfGame` function returns `TRUE`. As each successor of a node is explored, `gamma` maintains the current assessment of the position, based on all of the moves that have been searched so far. Once all successors have been examined, the minimax value for that position has been computed and stored in `gamma`, which can be returned to a higher level within the tree.

The minimax algorithm can also determine which move yields the score `gamma`, and return that up the tree as well. However, there is only one place we are interested in the move choice: the root of the game tree. We could write a special version of `Maximize` that returns a best move and the minimax value. However, there are other methods of storing the best move and minimax value. We shall discuss these methods in Section 2.3.

All two-player zero-sum games with perfect information and no chance moves can be solved using this algorithm. If we already know how to solve all games in this class, why do we not know the best moves to play in the game of chess or checkers? Unfortunately, these games have very large search trees. Although computers are fast, it would take a desktop computer billions of years to compute the final result for the game of chess from the initial position.

The magnitude of this problem has forced programmers of this algorithm to make some compromises. It is computationally infeasible in most positions to have all of the branches of the game tree terminated by a position where we can assign the perfect payoff vector. Thus, branches in the game tree must be stopped at arbitrary points in the search. Since we are not necessarily at the end of the game, we need to assign values for positions that approximate the chance of winning for the player to move.

We must rely on an *evaluation function* to approximate this chance of winning the game. As the chance of winning the game increases, the evaluation function should increase in a similar manner. A winning position is usually given a value of $+\infty$ in an evaluation function, while a lost position has the value of $-\infty$. How to generate, train, and update an evaluation function is beyond the scope of this document. However, we will define some components frequently found in evaluation functions.

A sample evaluation function for a game may count the number and type of pieces on the board, otherwise known as *material*. Material is the most important component of chess and checkers evaluation functions. As an example from the game of checkers, assume that each checker is worth 1 point, and each king is worth 2.5 points. By computing the number of points associated with all of the checkers and kings that you have, and subtracting that by the number of points for the checkers and kings that your opponent has, we have an rudimentary evaluation function for any position in the game of checkers. In chess, the typical evaluation function states that a pawn is worth 1 point, bishops and knights are worth 3 points, a rook is worth 5 points and a queen is worth 9 points.

Mobility, the number of moves available in a position, may be computed for both players. Mobility is the dominant factor in the evaluation function of top Othello programs, and is an important factor in most chess and checkers programs. For example, if we take an Othello position and count the number of moves you have and subtract that by the number of moves your opponent has, the result is a good evaluation function for the game of Othello.

We continue our discussion of the search algorithm by re-examining the minimax search algorithm from Figure 2.3. It is poor software engineering to use two pieces of code that do similar things when one would suffice. A different formulation of the

```
int Negamax(position p) {

   int numOfSuccessors;       /* total moves */
   int gamma;             /* current maximum */
   int i;                    /* move counter */
   int sc;       /* score returned by search */

   if (EndOfSearch(p)) { return(Evaluate(p)); }
   gamma = -∞;
   numOfSuccessors = GenerateSuccessors(p);
   for(i=1; i <= numOfSuccessors; i++) {
      sc = -Negamax(p.succ[i]);
      gamma = max(gamma, sc);
   }
   return(gamma);

} /* Negamax */
```

Figure 2.4: The Negamax Formulation of the Minimax Algorithm

functions `Maximize` and `Minimize` removes this problem. This formulation is called the *negamax* formulation and is shown in Figure 2.4. The main differences between the minimax formulation given in Figure 2.3 and the negamax formulation are: (1) the `Evaluate` function returns the evaluation of a position for the player to move at that node, instead of `GameValue`, which returned the payoff for Max and could only be used at the end of the game; (2) the `EndOfSearch` function is written to terminate the depth-first search before the search progresses too deep into the tree (to ensure completion of the search); (3) at each level, the scores are negated so that nodes where we would minimize can maximize as well. This compacts the code into one recursive function.

Let us now analyze the search executed by the minimax algorithm. At each node in the game tree, we must look at all of the children of that node. The average number of children we must look at is called the *branching factor*. The branching factor of a particular node in the game tree is dependent on the position that the node represents, but we shall simplify our analysis by assuming a constant branching factor, b, at all non-leaf nodes.

One of the conditions that is commonly used to stop recursing down the tree is the depth of the position in the tree. Whenever that depth is reached during the search, the position is immediately evaluated. Let us assume that this occurs at d ply within the game tree. Thus, all possible alternatives for the next d moves are examined from the position at the root of the game tree.

Using the branching factor and depth of the tree, we can calculate the size of the game tree. There are two important metrics that are commonly used in the literature to measure the size of a game tree: the *number of bottom positions*, NBP [86] and the *node count*, NC. A bottom position is another name for a leaf in the game tree, as determined by the `EndOfSearch` function. These bottom positions are not necessarily terminal nodes within the game tree, although some bottom positions may be terminal nodes if we have reached the end of the game. In general, the number of bottom positions measures how many times the function `Evaluate` is called over the course of a search. The number of bottom positions visited by the minimax algorithm in a game tree with uniform branching factor b and depth d is:

$$\mathrm{NBP}_{\mathrm{MINIMAX}} = b^d.$$

The node count in a game tree, under the same conditions is:

$$\mathrm{NC}_{\mathrm{MINIMAX}} = \frac{b^{d+1} - 1}{b - 1}.$$

The number of bottom positions is the more popular of the two measures. One reason for this is that many game-playing programs spend the majority of the CPU time in the `Evaluate` routine when executing a search. Thus, the number of bottom positions can be indicative of how much time a search will take. Another reason is that the number of bottom positions is generally more stable than the node count when averaged over time.

The exponential growth of these metrics as we increase the depth of search is bothersome, since there is strong evidence of a positive correlation between depth of search and playing strength in chess, Othello and checkers programs. Research into finding the minimax value (and hence, the principal variation) by evaluating a smaller number of bottom positions is critical, due to the limited availability of computing resources.

In the late 1950s, it was realized that it was not necessary to search all of the nodes to ascertain the minimax value of a game tree [70]. Knuth and Moore [47] have shown that there is a theoretical minimum number of nodes that the minimax algorithm must visit to determine the minimax value. If we have a game tree with uniform branching factor b, and all leaves of the game tree are at depth d, the minimax algorithm must visit at least $\mathrm{NBP}_{\mathrm{MIN}}$ bottom positions, where:

$$\mathrm{NBP}_{\mathrm{MIN}} = b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1.$$

There are many algorithms that exhibit this best-case for uniform game trees. The $\alpha\beta$ algorithm satisfies this best-case property and will be described in the next section.

2.2 The $\alpha\beta$ Algorithm

The first published account of the $\alpha\beta$ algorithm can be found in an article by Brudno [18]. However, there are numerous claims by the early researchers in computer games as to who developed the method first. These claims have been summarized by Knuth and Moore [47].

The $\alpha\beta$ algorithm is a modification of the minimax algorithm. Two bounds are used at each node in the tree, α and β, and these bounds are passed down as we traverse the tree in a depth-first manner. At any node, α represents the smallest node value that can affect the minimax value above that point in the tree, while β represents the largest node value that can affect the minimax value. Thus, α and β are often referred to as the *search window*. The search window is usually written as (α,β) and that notation will be used throughout this document.

The parameter α represents the largest minimax value of evaluated branches at Max nodes along the path to the node, including the node itself. As each subtree underneath a Max node is evaluated, α may be increased if we see a new maximal branch. Thus, as we descend the search tree or explore more successors at a specific

```
int AlphaBeta(position p, int alpha, int beta) {

   int numOfSuccessors;       /* total moves */
   int gamma;             /* current maximum */
   int i;                    /* move counter */
   int sc;      /* score returned by search */

   if (EndOfSearch(p)) { return(Evaluate(p)); }
   gamma = alpha;
   numOfSuccessors = GenerateSuccessors(p);
   for(i=1; i <= numOfSuccessors; i++) {
      sc = -AlphaBeta(p.succ[i],-beta,-gamma);
      gamma = max(gamma, sc);
      if (gamma >= beta) { return(gamma); }
   }
   return(gamma);

} /* AlphaBeta */
```

Figure 2.5: A Negamax Formulation of the $\alpha\beta$ Algorithm

node, α is monotonically increasing. Similarly, β represents the smallest minimax value of a node's evaluated branches at Min nodes along the path to the node, including the node itself. Thus, β is monotonically decreasing as we descend the search tree or explore more branches at a Min node.

When we reach a point where $\alpha \geq \beta$, we know that there is a better path for one of the players closer to the root of the tree. There is no need to search the subtree underneath a node where $\alpha \geq \beta$, and we can return to our parent node immediately. In effect, this *prunes* parts of the tree that cannot contribute to the minimax value. It has been shown that the $\alpha\beta$ algorithm will return the correct minimax value if the root position is searched with the search window ($-\infty$,$+\infty$) [47].

The $\alpha\beta$ algorithm, using the negamax framework, is given in Figure 2.5. As we pass `alpha` and `beta` down to the next level, we must negate and swap the two parameters so that the bound to be changed is always maintained in `alpha`. If we examine the negamax formulation carefully, we note that we increase `alpha` at even plies (max nodes) within the tree, and decrease `beta` at odd plies (min nodes), fulfilling the requirements of the $\alpha\beta$ algorithm.

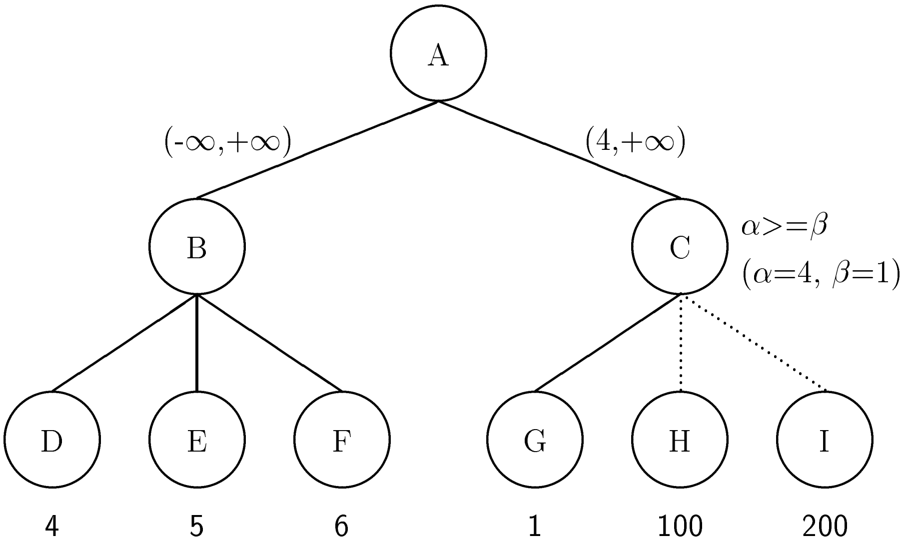


Figure 2.6: Example of an $\alpha\beta$ Shallow Cutoff

If we refer to the game tree in Figure 2.6, we can see an example of a *shallow cutoff*. If we proceed in a left-to-right manner through the tree as we execute our depth-first search, we explore the left branch of the root first and discover that the first move generates a minimax value of 4 for the maximizing side. Thus, when we start searching the right-hand branch, α is already set to 4. Now, as we explore the right branch, we get a minimax value of 1 at node G. Thus, β will be reduced to 1 at node C. We have shown that the minimizing side can keep the score down to at most 1 at node C, while Max will get a score of at least 4 by playing the move leading to node B. Thus, there is no point in exploring the other children of node C. The maximizing side should choose the move leading to B over the move leading to C, and the two branches leading to nodes H and I are never explored.

The $\alpha\beta$ algorithm might also generate *deep cutoffs*. A deep cutoff is caused by information gathered at a depth d in the tree being used to prune moves at a depth greater than d. For example, the bound information generated by exploring the subtree below node B cuts off the search at node J in Figure 2.7.

Before we continue discussing the $\alpha\beta$ algorithm, there is an interesting feature in Figure 2.6 that should be emphasized. The optimal play in the shallow cut example

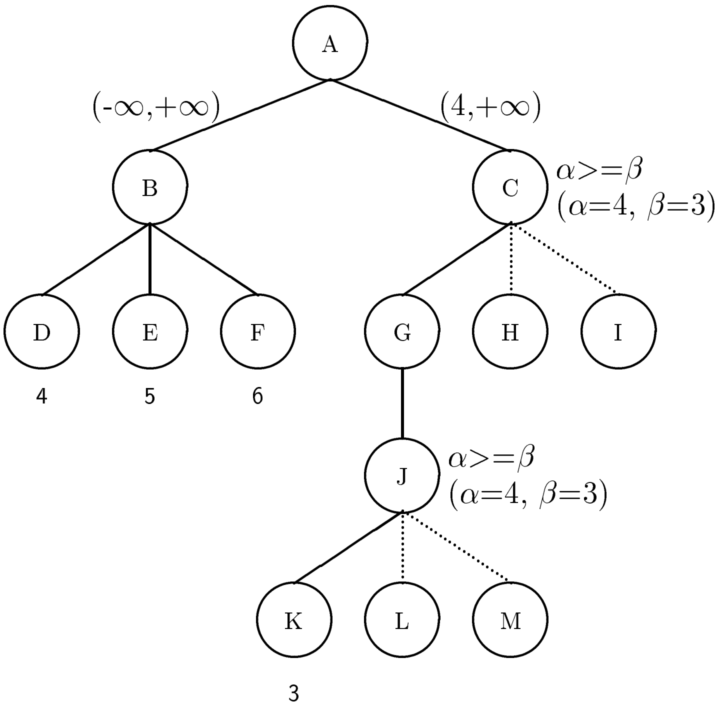


Figure 2.7: Example of an $\alpha\beta$ Deep Cutoff

is to play the move leading to node B. However, this definition of optimality assumes that the opponent will also play optimally with respect to the evaluation function. If the opponent is fallible, it is likely that the move leading to node C will provide better winning chances. Thus, the optimal move (with respect to the evaluation function) is not always the best move to play against a fallible opponent.

Earlier, we stated that the $\alpha\beta$ algorithm can visit the optimal number of nodes ($\mathrm{NBP}_{\mathrm{MIN}}$). When does the $\alpha\beta$ algorithm search $\mathrm{NBP}_{\mathrm{MIN}}$ nodes? We have seen that cutoffs occur when the moves leading to better minimax scores are searched before those that lead to worse minimax scores. At every node, if we can expand the move that leads to the best minimax value (largest at Max nodes, smallest at Min nodes) first, the $\alpha\beta$ algorithm searches only $\mathrm{NBP}_{\mathrm{MIN}}$ bottom positions. A tree of this sort is called a *perfectly-ordered* game tree. The proof that the $\alpha\beta$ explores $\mathrm{NBP}_{\mathrm{MIN}}$ bottom positions in a perfectly-ordered game tree was first given by Slagle and Dixon [86].

In a *worst-ordered* tree, the minimax values of the successors are arranged from

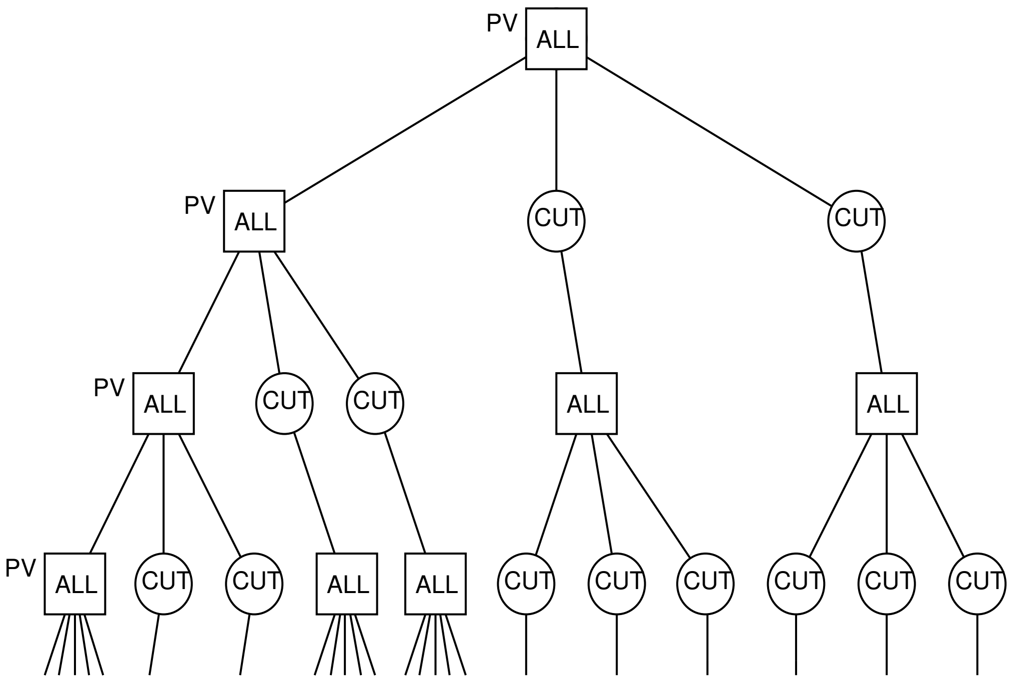


Figure 2.8: The Critical Tree

worst to best for every node. This prevents the $\alpha\beta$ algorithm from performing any cutoffs, and $\alpha\beta$ will explore the full minimax tree. Most game-playing programs generate *strongly-ordered* game trees; in practice the best move is searched first 90% to 95% of the time [74]. A lot of research has focused on getting better move orderings, and some heuristics for generating this ordering will be given in Section 2.3.3.

It is important to note that there is a portion of the tree that must always be searched, irrespective of move ordering, to determine the minimax value. The *critical tree* is defined as the perfectly-ordered tree that is generated when $\alpha\beta$ is started with the search window $(-\infty,+\infty)$. Figure 2.8 shows the structure of the critical tree. Nodes marked ALL have all of their successors explored by the $\alpha\beta$ algorithm, while nodes marked CUT have only one successor explored before the $\alpha\beta$ algorithm cuts off the rest of the successors.

The principal variation (PV on the diagram) of an $\alpha\beta$ critical tree is the first branch searched. We do not know *a priori* where the principal variation is to be located within a game tree. However, in the case of the $\alpha\beta$ critical tree, we are fortunate enough to search the principal variation first. All of the PV nodes are searched with the window $(-\infty,+\infty)$. Thus, we search all children at PV nodes and they become ALL nodes.

Subtrees that lie off of the leftmost branch in the critical tree are attempts to prove that a move on the PV branch is not optimal. Assume that it is player A's turn to move at the PV node, and player B is his opponent. Thus, we are attempting to prove that player A's choice of move at the PV node is not correct by evaluating a different subtree. Inside this subtree, we explore all possible moves for player A; these are the internal ALL nodes within the subtree. However, when player B must move inside this subtree, we need only look at one response to refute player A's choice. The refutation exists, since the first move of player A at the PV node is in a principal variation; all other move choices for A lead to lower or equivalent minimax values.

The common terminology in the literature to describe PV, CUT, and ALL nodes are type-1, type-2 and type-3 nodes, respectively. The numerical names come from Knuth and Moore's original description of the structure of the critical tree [47].

We should note that if the game tree has a constant branching factor b, a d-ply $\alpha\beta$ critical tree will have exactly $\mathrm{NBP}_{\mathrm{MIN}}$ bottom positions. Knuth and Moore originally defined the critical tree as the *minimal tree* because it is the smallest tree which determines the minimax value. However, the critical tree is not necessarily minimal in practice. These confusing and seemingly contradictory statements will be addressed in Section 2.3.5 after we describe some of the enhancements to the $\alpha\beta$ algorithm.

2.3 Improving the $\alpha\beta$ Algorithm

This section will deal will some of the numerous improvements that have been made to the $\alpha\beta$ and other minimax-based algorithms over the last thirty years. This is not intended to be an exhaustive survey of all improvements. However, those methods that are mentioned in other chapters of this work will be described and defined here.

2.3.1 Fail-Soft and Aspiration Searching

The first improvement that needs introduction is the *fail-soft* improvement [33], and only requires two changes to the algorithm. In the $\alpha\beta$ algorithm given in Figure 2.5, `gamma = alpha` is replaced by `gamma = -`∞. The other change is to modify the recursive `AlphaBeta` call so that the last parameter changes from `-gamma` to `-max(gamma,alpha)`.

By itself, this change does not do anything for the searching efficiency of the $\alpha\beta$ algorithm. Any node evaluated by the fail-soft $\alpha\beta$ algorithm will also be evaluated by the strict $\alpha\beta$ algorithm; the converse is also true. However, this is of importance when an *aspiration search* is attempted.

An aspiration search is an attempt to guess at the minimax value of the game tree before doing the search. In aspiration $\alpha\beta$, we call `AlphaBeta` with a lower bound on the guessed minimax value in the `alpha` parameter, and an upper bound on the guessed minimax value in the `beta` parameter. For example, if the program determined with some certainty that the minimax value would be between 5 and 15, the initial search window or *aspiration window* could be changed from (-∞,+∞) to (5,15).

If this aspiration window contains the minimax value, it will cause cutoffs at earlier points in the tree than the full search window. Hence, $\alpha\beta$ with an aspiration window will evaluate fewer bottom positions than the $\alpha\beta$ algorithm started with an infinite search window.

If the window used does not contain the minimax value, we will know because the value returned will not lie between our guesses for `alpha` and `beta`. If the value returned is less than or equal to `alpha` the search is said to have *failed low*. Similarly, if the value returned is greater than or equal to `beta`, the search has *failed high*.

In the case of failing low in $\alpha\beta$ without the fail-soft improvement, we must re-search the root with the window (-∞,`alpha`) to find the correct minimax value. In fail-soft $\alpha\beta$, it has been shown that if f is the value that is returned by a fail low search, then the minimax value must be less than or equal to f [34]. The re-search need only be done in the window (-∞,f) for fail-soft $\alpha\beta$. The smaller window (since $f \leq$ `alpha`) means that fewer nodes need to be searched by fail-soft $\alpha\beta$ to determine the minimax value. The corresponding assertions hold true when a fail high occurs.

2.3.2 Iterative Deepening

Iterative deepening was an idea suggested by Scott [84] for his chess program, although it would take a few years to discover the full extent of the benefits. The idea is that the $\alpha\beta$ algorithm should be limited to exploring a small search depth k by forcing evaluations of nodes once they reach that depth. Once that search is done, the limit k can be moved forward by a step s, and the search can be repeated to a depth of $k + s$. In chess programs, k and s usually equal 1. Thus, the program does a 1-ply search before doing a 2-ply search, which occurs before the 3-ply search *et cetera*.

Scott noted that there is no way of predicting how long an $\alpha\beta$ search will take, since it depends heavily on the move ordering [84]. However, by using iterative deepening, one can estimate how long a $(k + 1)$-ply search will take, based on the length of the preceding k-ply search. Unfortunately, the prediction may be far off the accurate value. In some cases, a real time constraint (such as a time control in a chess game) may necessitate aborting the current search. Without iterative deepening, if a program has not finished a search when the time constraint interrupts the search, the program may play a catastrophic move. With iterative deepening, we can use the best move from the deepest search that was completed.

Other benefits were explored by Slate and Atkin in their Chess 4.5 program [87]. They discovered that there were many statistics that could be gathered from a search iteration, including the principal variation. The principal variation of a k-ply search is a good starting place to look for a principal variation of a $(k + 1)$-ply search, so the principal variation from the k-ply search is searched first at depth $(k + 1)$. This improves the ordering of the moves in the $(k + 1)$-ply search. Usually, the number of bottom positions explored for all of the searches up to depth d with iterative deepening is significantly smaller than attempting a d-ply search without iterative deepening.

On top of the better search ordering, we can get estimates to generate aspiration windows from iterative deepening. Given that a k-ply search leads to a minimax value v, we can use a small window around v as a hypothetical range for the minimax value of the $(k+1)$-ply search, such as $(v-\varepsilon, v+\varepsilon)$. As discussed earlier in Section 2.3.1, reducing the size of the initial window reduces the number of bottom positions that are explored by the $\alpha\beta$ algorithm. If we are confident that the evaluation function does not fluctuate wildly as we evaluate trees of greater depth, using aspiration windows in conjunction with iterative deepening will save many bottom positions from being evaluated.

2.3.3 Move Ordering Techniques

The first type of move ordering that was suggested for ordering the successors in a game tree was a *fixed ordering* [86]. A fixed ordering is based on sorting the moves based on the evaluation of each successor node. The move leading to the best minimax value is placed at the front of the list, while the move leading to the worst minimax value is moved to the end of the list. Thus, the ordering may vary as the search tree is traversed, but is fixed every time a particular node is visited. This method has worked well in sorting moves for Othello endgame searches, but does not work well in chess. One problem is that the ordering is not dynamic: it cannot use information gathered from searching other parts of the tree. As well, the evaluations are wasted in the case of a cutoff occurring at that node in the tree.

Specific information about a search can be saved in a *transposition table*, first mentioned by Greenblatt, Eastlake and Crocker [38]. In the $\alpha\beta$ procedure given in Section 2.2, all of the information about a node can be accumulated including the best score, the best move from that position, the depth it was searched to, and whether the score is exact, a fail low or a fail high. All of this information is commonly stored into one transposition table entry. Transposition tables are normally constructed as closed hash tables, with hashing functions that are easy to update (such as a number of XOR operations [97]) as one traverses the tree. The transposition table information can be used in two main ways: *duplicate detection* and move ordering.

Why would we need to detect duplicates in a game tree? In reality, the game tree is a graph; some of the positions appear in multiple places within the tree. Thus, it makes sense that each position should only be explored once if the information obtained is sufficient to terminate the search. The transposition table assists in finding and eliminating these duplicated positions.

The same position in the game will always hash to the same location in the transposition table. What if the information stored in the table is the same position as the current node, and the stored result of a search of that position is at least as deep as the search we are attempting to execute? Depending on the type of information that has been saved in the transposition table, one of two things may occur. If we have an exact minimax value in the hash table for a search that is at least as deep as the one to be executed, we can use the result from the hash table and prune the entire search. In the majority of cases where we can use search results from the transposition table, we will only have a bound stored in the hash table. In this case, the bound can be used to reduce the size of the search window. If this reduction completely closes the search window (i.e. $\alpha \geq \beta$), the search can also be pruned without exploring any nodes.

Most of the time, the duplicate detection will fail to completely eliminate the search, and we can exploit the transposition table to improve our move ordering. In the games we are studying, the best move from a previous search depth is likely to be the best move at the current search depth. Thus, we can obtain the previous best move from the transposition table, and search the previous best move before all others. In general, the move ordering benefits of combining iterative deepening and the transposition table are at least as important to the NBP count as the duplicate detection property, depending on the application chosen.

The *killer table* [87] is another method of storing moves which might be the best move in a position. However, the killer table does not store moves for specific positions. Each entry in the table corresponds to a given depth in the game tree. When a move causes a cutoff in the search, the *killer move* is changed to the move that causes the cutoff.

The rationale is that the same move played at the same point in the move sequence might cause cutoffs in other branches of the tree. For example, assume that we have a position where Min has an obvious winning move, but Max has some defensive moves that can be played to prevent the winning move. Each time Max searches a move that is not involved in this defence, Min will find the winning move and cause a cutoff at that level. By searching this killer move before other move choices for Min, we can determine which of Max's moves will fail.

The killer and transposition table only offer move ordering information about a couple of moves in the move list. The *history heuristic* [82] is a useful technique for sorting all other moves. In the game of chess, a 64 by 64 matrix is used to store statistics. Each time a move from a square `startsq` to a square `endsq` is chosen as a best move during the search, a bonus is stored in the matrix at the location [`startsq,endsq`]. The size of this bonus depends on the depth at which the move was successful at. A bonus that varies exponentially based on the depth of the subtree under that position has been found to work well in practice. Moves with higher history values are more likely to be best moves at other points in the tree; thus, moves are sorted based on their *current* history values. This makes a dynamic ordering for all possible legal moves in cases where no ordering information exists.

In the programs that the author is aware of, the three latter move ordering techniques are used. The transposition table move is always used first, since it yields specific information about that node from a previous search. Once the transposition table move has been searched, the next move to be attempted is the killer move (if it is legal in the position). Once both of these heuristics have been used, the remaining moves are sorted by the history heuristic.

2.3.4 Null Windows

A further improvement on the standard fail-soft $\alpha\beta$ algorithm can be derived from searching with *null windows*. Earlier, it was noted that a smaller initial search window reduces the size of the tree searched by the $\alpha\beta$ algorithm. With the use of the ordering techniques described earlier, it is likely that the first move examined will have the best value. As a consequence, all of the other moves are likely to be inferior. Let us assume that the first move returns a minimax value `gamma`. Instead of searching the other moves with the window (`gamma`,`beta`), we can search them with a null window (`gamma`,`gamma`+1).

If the move is indeed inferior, the value returned to that node will be less than or equal to `gamma` and no further work is required. However, if the value returned is greater than `gamma`, then the move just searched was superior to the first move, and we must determine the correct minimax value by calling the routine with a larger window.

Null window searching is not worth implementing in a domain where the move ordering is poor, due to the number of re-searches required. In domains which are assisted by strong move ordering, null windows provide a saving in search effort. For example, Feldmann reports that the Zugzwang chess program visits 1.0% to 8.3% more nodes when using $\alpha\beta$ as opposed to using a search routine with null windows [29].

Principal Variation Search (also known as PVS) [60] and NegaScout [78] are two methods which implement null windows in a negamax formulation. The NegaScout routine is based on an improvement of Pearl's Scout algorithm [73], which uses null windows and a proof procedure to determine whether the move being tested is better than the one already evaluated.

The code for the NegaScout algorithm is given in Figure 2.9. The `plytogo` argument to this function decreases as the tree is descended, and eventually reaches zero. The `plytogo` passed in at the root is the limit to how far ahead NegaScout should search. The code that checks the transposition table, the killer table and the history

```
int NegaScout(position p, int alpha, int beta, int plytogo) {

   int numOfSuccessors;                         /* total moves */
   int gamma;                                /* current maximum */
   int i;                                       /* move counter */
   int sc;                          /* score returned by search */
   int under;                   /* alpha for move to be searched */
   int over;                     /* beta for move to be searched */

   if (plytogo == 0) { return(Evaluate(p)); }

   numOfSuccessors = GenerateSuccessors(p);
   if (numOfSuccessors == 0) { return(Evaluate(p)); }

   gamma = -∞;
   /* set window for first child */
   under = alpha;
   over = beta;

   for(i=1; i <= numOfSuccessors; i++) {
      sc = -NegaScout(p.succ[i],-over,-under,plytogo-1);
      /* Is a research necessary?  */
      if (sc > under && i > 1 &&
          sc < beta && plytogo > 2) {
         sc = -NegaScout(p.succ[i],-beta,-sc,plytogo-1);
      }

      gamma = max(gamma, sc);
      if (gamma >= beta) { return(gamma); }

      /* set window for next child */
      under = max(gamma, alpha);
      over = under + 1;
   }
   return(gamma);

} /* NegaScout */
```

Figure 2.9: The NegaScout Algorithm

heuristic is contained in the `GenerateSuccessors` function, which will generate a sorted move list based on those heuristics.

The first move is searched with the full window (`alpha`,`beta`), while all of the other moves are searched with the minimal window (`under`,`under`+1). There are a number of conditions on when we should re-search, other than the returned value `sc` is greater than our current maximum score `gamma`. We need not re-search the first move, since it was searched with a full window. Similarly, if `sc` $\geq$ `beta`, this results in a cutoff and a re-search would not be of any assistance.

The last condition notes NegaScout returns the correct minimax value when searching the last two ply, irrespective of the search window. When `plytogo > 2`, we need to re-search the game tree to determine a better bound on the minimax value. This condition is only used in game-playing programs where the search depth is fixed. Chess and checkers programs usually cannot use this enhancement due to the implementation of search extensions (described in Section 2.4).

If null window searches are good within the tree, why do we not use them at the root of the game tree? Recent research has shown that the common best-first search strategies can be reformulated as depth-first search strategies by using null windows at the root of the game tree and a sufficiently large transposition table. One such algorithm, a variant of Memory-enhanced Test called MTD(f), has been shown to be better than NegaScout in many of the games we are interested in [75].

Instead of aspiration searching, MTD(f) uses a heuristic guess about the minimax value and a succession of null window searches at the root of the game tree to rapidly generate upper and lower bounds on the minimax value. Once the upper and lower bounds have converged, the search is completed. An example of the driver code for MTD(f) is given in Figure 2.10. MTD(f) calls the `AlphaBeta` function, enhanced with a transposition table and the other move ordering techniques given in Section 2.3.3. MTD(f) does not use `NegaScout` since null windows are used throughout the game tree.

```
int MTD(position p, int guess) {

   int lowbd;       /* lower bound on minimax */
   int uppbd;       /* upper bound on minimax */
   int gamma;        /* current search window */
   int sc;       /* score returned by search */

   sc = guess;
   lowbd = -∞; uppbd = +∞;

   do {
      if (sc == lowbd) { gamma = sc+1; }
      else { gamma = sc; }
      sc = AlphaBeta(p, gamma-1, gamma);
      if (sc < gamma) { uppbd = sc; }
      else { lowbd = sc; }
   } while (uppbd != lowbd);
   return(gamma);

} /* MTD */
```

Figure 2.10: The MTD(f) Algorithm

2.3.5 The Minimal Graph

We stated at the end of Section 2.2 that the critical tree is not necessarily minimal. NegaScout and MTD(f), with the improvements listed previously in this section such as aspiration searching and transposition tables, can generate trees that are much smaller than the $\alpha\beta$ critical tree while generating the correct minimax value.

The *left-first minimal graph* has been used by many authors to illustrate the size of the smallest tree that we can search. The left-first minimal graph can be grown by searching the tree once to determine the minimax value, and then searching the tree again using the transposition table move ordering information to generate a perfectly-ordered game-tree [28].

However, we can do significantly better than the left-first minimal graph. The left-first minimal graph does not attempt to maximize the number of duplicates within the game tree, allowing the transposition table to cut off more branches. Furthermore, the left-first minimal graph does not always take the cheapest possible cutoff when attempting to determine the minimal graph. The cutoff found during the original search is used, but there may be other moves which prune the same node with less effort. Recent research has shown that the *real minimal graph*, the smallest possible tree that any algorithm can search to determine the minimax value, is significantly smaller than the left-first minimal graph [74]. Thus, game-tree search algorithms are not as efficient as portrayed in the literature.

However, determining the size of the real minimal graph for a game tree is very difficult. Not only is minimax search required, but numerous re-searches are required to find the optimal move ordering with respect to node count at every interior node within the tree. Finding the real minimal graph for game trees is a computationally intractable problem [74].

As a final note, researchers have been somewhat sloppy with the term "minimal tree". It has one of many definitions depending on the context in which it is mentioned, including the critical tree, the left-first minimal graph or the real minimal graph. Thus, the term has been avoided in this document.

2.4 From Fixed-Depth to Variable-Depth

There are numerous search extension and reduction heuristics that can be used in $\alpha\beta$-based algorithms. Some of these heuristics are used to stabilize the search algorithm, preventing wild fluctuations as the search depth is incremented. Other enhancements help reduce the search effort when a bad move is played, since fixed-depth game-tree search algorithms spend a lot of time proving that poor moves are inferior to the principal variation. We will discuss search extension heuristics first, followed by strategies to reduce the search depth.

2.4.1 Search Extensions

In some positions, searching one ply further ahead can cause a dramatic shift in the evaluation function. For example, consider the game of chess: if we consider a node that has a queen that can be captured, the evaluation may not take the capture into consideration. Thus, when the position is searched one ply deeper, we see the queen being captured, and the minimax value for the position changes dramatically.

The inability of a program to assess these tactical features in an evaluation function is called the *horizon effect* [10]. We do not want to evaluate a position at the bottom of the fixed-depth tree in games where such problems have been identified. We use *quiescence search* to extend the search to a *quiet* position.

For chess, a quiet position is often defined as a position without captures or checks on either king. Most modern chess programs extend the search of positions with checks and captures on the board until the checks and captures have been completely exhausted. This allows chess programs to see deep tactical combinations using a low nominal search depth.

Not all search extension heuristics are used to stabilize the search. In some situations within the game tree, one has cases where there are very few moves generated. In chess, an example of this occurs when a player is in check. Since the player must move out of check, there are a limited number of options available. This reduces the number of positions searched in the subtree underneath the node. Other branches of the tree may not have these move-limiting properties and we search more bottom nodes than the former subtree. Thus, one would like to extend the search depth when we see a position with a limited number of options so that we can search a similar number of bottom positions in each subtree.

There are other situations within the game tree where there is a single move which is clearly better than all other alternatives. The search can be extended for this move so that the search effort can be focused on the singular move. The application of this idea, *singular extensions*, can be found in the chess program DEEP THOUGHT [4] and its successor DEEP BLUE.

2.4.2 Search Reductions

In most games, we have a small number of good moves at every node within the tree and a large number of mediocre and poor moves. The goal of search reduction techniques is to quickly determine which moves are bad, avoiding a full fixed-depth search of a bad move.

The best known application of search reduction in the game of Othello is ProbCut [20]. Before a d-ply search is attempted, ProbCut executes a shallow-depth ($d' < d$) search with a wider search window. If the d'-ply search fails low or fails high, then the d-ply search will behave the same way with high probability, and the appropriate window bound (α or β) is returned back up the tree.

Although this idea has been attempted in other domains, such as in CHINOOK, ProbCut is the first attempt to use statistically gathered information to generate the window for the shallow-depth search. ProbCut requires that the mean and standard deviation of the difference between a d-ply search and a d'-ply search be computed off-line for positions in the test domain. The mean and standard deviation are used in the search to control the amount of certainty one wants in the pruning decision. In general, the more certainty one wants, the wider the search window for the shallow-depth search, and the smaller the number of searches that will be pruned.

In the game of chess, *null moves* have become the common method of reducing search effort. A null move is simply a pass in a position; the opponent is allowed to have two moves in a row. In games such as chess, simply passing is not legal, but for the purposes of testing the threats that exist in a given position, it can be considered as a legal move. If we have a position where a player does nothing and the search fails high, it is very likely that any move in this position will also fail high, and we can terminate the search immediately without exploring any of the real moves. To reduce the effort of searching the null move, the search depth is reduced by an additional 1 or 2 ply. This makes the null move an inexpensive heuristic for determining whether the branch is worth exploring. Null moves are discussed in numerous places throughout the literature [1, 9, 27, 37, 72].

To use null moves in a game-playing program, the implicit assumption is that passing is worse than doing something. This is not true in chess where a player is in check or in *zugzwang*. When a player is in check, the player must move to save the king from being captured. Zugzwang is a state in which it is better to pass than to make any move. In the game of chess, passing is not an option and a player in zugzwang must make a move and weaken their position. Null moves work well for chess (aside from the aforementioned exceptions) but not for Othello and checkers, since it is often preferable to pass instead of playing a move.

In the game of Othello, playing a move generally reduces your mobility and increases your opponent's mobility. Thus, null moves for your opponent tend to yield a pessimistic value, rather than an optimistic value. However, this suggests that null-move-enhanced searches (that is, adding null moves to the list of possible moves) gives an optimistic minimax value for the player.

2.5 Other Game-Tree Search Methods

Although the research on $\alpha\beta$-based game-tree search is our focus in this chapter, there are other search methods for game trees that are important contributions to the literature.

2.5.1 SSS*

Stockman [89] introduced the SSS* algorithm, a variant to the depth-first $\alpha\beta$ search algorithms for determining the minimax value. 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. The original algorithm, unfortunately, did not have the desired property. An improvement by Campbell [21] makes the dominance proof correct.

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.

Although versions of SSS* eventually managed to become faster than $\alpha\beta$ for game trees [79], it has been recently discovered that SSS* and other best-first search strategies (such as DUAL* [1] [51, 65]) can be implemented as a series of null-window $\alpha\beta$ calls, using a transposition table instead of an OPEN list [75]. The research showed that the perceived drawbacks of SSS* are not true. However, it is also important to note that the benefits also disappear: SSS* is not necessarily better than $\alpha\beta$ when dynamic move reordering is considered. Furthermore, when all of the typical $\alpha\beta$ enhancements are used, SSS* can be outperformed by NegaScout and MTD(f).

2.5.2 Algorithms for Probabilistic Evaluation Functions

Many approaches have been given in the literature that attempt to retrieve more information from a node than a simple minimax value. The information is used to determine the best move with less "brute force" than the current $\alpha\beta$-based approaches.

Berliner [11] introduced B* search, which attempts to prove that one move is better than all others at the root of the game tree. This can be accomplished through the use of heuristically-defined optimistic and pessimistic bounds on the minimax value of a node; the pessimistic bound of the best move at the root of the game tree should be better than the optimistic values of all other moves at the root. B* attempts in a best-first manner to focus the tree search towards the nodes that can give the maximum benefit towards completing the proof. Once this proof is complete, the search can be terminated. This allows the algorithm to make "obvious" moves quickly, a feature that the $\alpha\beta$-based algorithms lack.

[1] DUAL* is an algorithm that attempts to mirror the SSS* search strategy. SSS* attempts to determine the minimax value by starting at $+\infty$ and approaching the minimax value from above. DUAL* starts at $-\infty$ and approaches the minimax value from below.

Palay [72] introduced a probabilistic version of B*, where the optimistic and pessimistic bounds were determined by shallow null-move searches instead of heuristic evaluation functions. Furthermore, Palay introduced rules for backing up the probability distribution determined at each node further up the tree. Berliner and McConnell [12] have further improved methods for finding optimistic and pessimistic estimates to guide the search. The probability distributions from Palay's thesis were also improved to simplify the structures maintained within the search tree.

Baum and Smith [7, 8] have done similar work to probability-based B* with the BPIP (Best Play for Imperfect Players) algorithm. The basic idea behind BPIP is to return a probability distribution instead of a simple value when searching, and choose the move at the root that has the highest mean probability distribution. Experiments show that the approach works better than $\alpha\beta$ on both Othello and warri.

2.5.3 Conspiracy Numbers

McAllester introduced conspiracy numbers [66] as an alternative to finding a minimax value. The basic idea is to determine how many leaves within a game tree must change their value for the minimax value of a position to change to that value. One would like to show the root of the game tree needs a large number of nodes to conspire to change its minimax value. The tree can be grown one node at a time in a manner that attempts to maximize the conspiracy numbers at the root of the game tree. The idea seems reasonable in theory and works well in tactical positions in the game of chess. However, it is often very difficult to prove that the root of the game tree relies on 2 or more leaves in non-tactical positions [83].

Lorenz and Rottmann developed the Controlled Conspiracy Number Search algorithm [56], which attempts to partition the work in proving a conspiracy number at a node v amongst the successors of v. This approach allows the search space to be subdivided and examined in the traditional depth-first manner that $\alpha\beta$-based programmers are accustomed to. Furthermore, information about what the node is attempting to prove allows the program to use tighter $\alpha\beta$ search windows to determine scores at leaf nodes than the original Conspiracy Number Search algorithm, which relied on exact values at every leaf. UlyssesCCN played at the 1994 Paderborn Chess tournament using the CCNS algorithm, and achieved a score of 3.5 in 7 games, which is an admirable result for a relatively new algorithm.

2.6 Summary

In theory, $\alpha\beta$ is a very simple algorithm. The `AlphaBeta` algorithm given in Figure 2.5 has less than 20 lines. However, real implementations of $\alpha\beta$ include transposition tables, iterative deepening, the history heuristic, killer tables, *et cetera.* All of these enhancements yield an algorithm that can be more than 20 pages of code. As a result, we have very small trees and near-perfect move ordering. We are approaching the limit of what $\alpha\beta$-based game-tree search algorithms can accomplish on a single processor.

There are two possible methods of taking game-tree search to the next level of performance. One alternative is the examination of algorithms not based on $\alpha\beta$-based search. B* and BPIP are interesting ideas on paper, but are very complicated to get working in practice. Independent tests should be completed to verify the experimental claims of B* and BPIP; both algorithms seem worthy. The work on conspiracy numbers shows promise, although it is not better than $\alpha\beta$-based variants employed by game-playing programs at the present time.

A second alternative is the use of parallelism to speed up $\alpha\beta$-based algorithms. A discussion of how parallelism has been utilized in tree search algorithms is given in the subsequent chapter.

Chapter 3

Parallel Search

3.1 Introduction

The field of parallel search algorithms is blessed with a veritable cornucopia of game-tree search and single-agent search algorithms. As well, attempts have been made recently at writing general libraries for branch-and-bound search problems. The solutions revolve around a major characteristic of the sequential search algorithm employed: depth-first search or best-first search. As we shall see in this chapter, the depth-first search algorithms are generally more successful than their best-first counterparts.

Section 3.2 introduces some of the terms that will be necessary to understand the brief descriptions given in this chapter. Section 3.3 covers parallel $\alpha\beta$ game-tree algorithms in depth, and Section 3.4 gives a summary of other parallel game-tree search algorithms. Section 3.5 introduces the closely-related field of single-agent search and illustrates approaches to parallelizing single-agent search. Section 3.6 discusses the work on portable parallel branch-and-bound libraries.

3.2 Parallel Search Terminology

Before we start to discuss the various parallel tree-searching algorithms that are available, it is important to define some of the common terms that are used in the field of parallel processing.

Most work on parallel processing uses the variable n to represent the number of processors being used. This definition will be used throughout the document.

3.2.1 Speedup and Efficiency

One way to measure the performance of a parallel algorithm is the *speedup*, which measures how much faster the parallel algorithm arrives at the same solution than the best sequential algorithm:

$$\text{speedup} = \frac{\text{time taken by the best sequential algorithm}}{\text{time taken by a parallel algorithm}}.$$

The speedup is not normalized to the number of processors used. The *efficiency* of a parallel algorithm measures how well the entire system is utilized:

$$\text{efficiency} = \frac{\text{speedup}}{n} = \frac{\text{time taken by the best sequential algorithm}}{\text{time taken by a parallel algorithm} \times n}.$$

In general, one would like to achieve high efficiency while using many processors. An algorithm with a *linear speedup* is an algorithm with a constant efficiency as you increase the number of processors.

In most tree-searching algorithms, the best sequential algorithm for the goal in question (determining the minimax value, finding the optimal path from the start to the goal node) is unknown. In general, authors use their sequential algorithm as the baseline when measuring speedups. This is defined as the *observed speedup*:

$$\text{observed speedup} = \frac{\text{time taken by a sequential algorithm}}{\text{time taken by a parallel algorithm}}.$$

We can also define the *observed efficiency* in an analogous manner. Most of the speedups and efficiencies that will be discussed in this document will be observed.

However, using the observed speedup causes a host of problems that cannot be dismissed. There is a tendency to compare the observed speedups or observed efficiencies to determine something about a pair of algorithms, such as Algorithm A is better than Algorithm B. There are a number of reasons why this can be very misleading, especially if the two sequential programs used are different.

(1) Simulated observed speedups are often misleading. The simulation model is often over-simplified, and it is difficult to determine how a parallel algorithm will behave on a real application with a large number of processors.

(2) Observed speedups with artificial game trees rarely reflect the unique properties of searching a real game tree, and might be geared towards illustrating the strength of the parallel algorithm [75].

(3) We cannot take real trees at face value either, since the sequential algorithm used to generate the tree may not be an efficient searcher. The sequential algorithm used could be reorganized in a more efficient way for use on a single processor. Inefficiencies could also be caused by leaving out key move ordering techniques, such as a sufficiently large transposition table, iterative deepening or killer moves. A poor sequential searcher will yield more opportunities for parallelism, and may increase the observed speedup achieved by the algorithm.

(4) The varied branching factor of the game trees in different test domains has a profound effect on the observed speedup. 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 [57]. The breadth of checkers trees yield observed speedups that are much smaller in magnitude than the same algorithm implemented to search a chess tree.

(5) The speed of the processor versus the speed of the network also affects how the algorithm performs on the target hardware. 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 impossible to objectively compare observed speedups or efficiencies for two parallel search implementations without understanding the experimental methodology.

3.2.2 Overheads

This would not be an active field of research if we could achieve near perfect efficiency for parallel tree search algorithms. We must understand where the inefficiencies of the parallel search are coming from. Thus, one often analyzes the overheads associated with a parallel algorithm. The *total overhead* is the amount of additional processing time that is required to achieve the same result:

$$\text{total overhead} = \frac{\text{parallel time } \times n - \text{ sequential time}}{\text{sequential time}}.$$

The parallel algorithm often searches many more nodes than the sequential algorithm does. The *search overhead* is defined as:

$$\text{search overhead} = \frac{\text{number of nodes searched by parallel algorithm}}{\text{number of nodes searched by sequential algorithm}}.$$

The search overhead may occur for any number of reasons. For example, information from other parts of the search tree may be unavailable to the processor, causing it to search more nodes. Another possible reason is that unnecessary speculative work could be initiated by the parallel algorithm and subsequently discarded.

We will be focusing on the difference between synchronous and asynchronous parallel search algorithms. In a parallel search algorithm, the different parallel tasks are often dependent upon one another to efficiently solve a specific task. A *synchronization point* is a point in the algorithm where all processors must reach consensus on their work before any processor is allowed to continue past that point. An example of such a synchronization point in game-tree search would be that all processors must complete their work at k-ply before proceeding to the $(k+1)$-ply search.

In general, a *synchronous algorithm* has many of these synchronization points. The amount of time that processors sit idle at synchronization points is called the *synchronization overhead*, and can be expressed as:

$$\text{synchronization overhead} = \frac{\text{time spent idle waiting at synchronization points}}{\text{sequential time}}.$$

Conversely, an *asynchronous algorithm* has no synchronization points during the search (aside from the end of the search). Each process is allowed to execute its own work without regard for the global state of the search. At no time does a process in an asynchronous search algorithm sit idle while waiting for other processes. Thus, it does not make sense to express synchronization overheads for asynchronous algorithms.

Other overheads may be significant, depending on the parallel algorithm used, such as the *communication overhead* or the *parallelization overhead*. In general, the communication overhead measures the amount of time that the parallel algorithm spends sending and receiving results from other processors. Since a parallel algorithm does many things in addition to running the sequential algorithm, we expect the parallel algorithm to run slower. The parallelization overhead measures the difference in terms of number of bottom positions (or nodes) examined per second in the sequential and parallel algorithms.

In an efficient parallel algorithm, the communication overhead should be very similar to the parallelization overhead. However, in some cases, the parallel algorithm necessitates using different data structures to keep track of the parallel work. These data structures may take a long time to update and are generally not counted in the sequential algorithm. Depending on the algorithm, it may make sense to analyze both the communication and parallelization overhead.

3.3 Parallel $\alpha\beta$-based Game-Tree Search

In the last twenty years, a number of articles and theses have been written that contain innovative solutions to parallel $\alpha\beta$-based game-tree search. The authors of the parallel algorithms have shown how their work is unique and interesting. Some authors have attempted to classify the game-tree search algorithms by listing implementation details [5, 23].

Upon deeper analysis, we notice that many algorithms are simply minor variations of other algorithms. Some algorithms that seem very different on the surface are using the same underlying algorithm; the implementation serves to obfuscate the nature of the algorithm being used. To our knowledge, no attempt has been made to classify the algorithms based solely on the algorithmic properties. A taxonomy would make it easy to ascertain what has and has not been accomplished in parallel $\alpha\beta$-based game-tree search. In particular, we will show that asynchronous search algorithms have been ignored by the majority of researchers.

The taxonomy in Tables 3.1 and 3.2 isolates the differences between the various $\alpha\beta$-based algorithms and their implementations. We will first describe the categories within the taxonomy, followed by a brief summary of each of the algorithms mentioned in the taxonomy.

3.3.1 Comparison of the $\alpha\beta$ Algorithms

Table 3.1 summarizes and classifies the various $\alpha\beta$-based algorithms.

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 algorithm has been described in many papers, but all the details are given in Feldmann's thesis [29].

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

Algorithm (Reference)	Date First Described	Processor Hierarchy/ Control Distribution	Parallelism Possible At These Nodes	Synchronization Done At These Nodes
Parallel Aspiration Search [6]	1978	Static/ Centralized	Root ($\alpha\beta$ window)	Root
Mandatory Work First [2]	1979	Static/ Centralized	Type-1+3+Left-most child of 3	Root + Bad Type-2
Tree Splitting [34]	1980	Static/ Centralized	Top k-ply	Root
PV-Split [61]	1981	Static/ Centralized	Type-1	Type-1
Key Node [55]	1983	Static/ Centralized	Type-1+3+Left-most child of 3	Root + Bad Type-2
UIDPABS [69]	1986	Static/ Centralized	Root	None
DPVS [81]	01/1987	Dynamic/ Centralized	Type-1+3+ Bad Type-2	Type-1+3+ Bad Type-2
EPVS [45]	06/1987	Dynamic/ Centralized	Type-1+3	Type-1+3
Waycool [31]	1987	Dynamic/ Distributed	All, except Type-2 with no TT entry	Nodes with TT & no cutoff
Young Brothers Wait [29]	10/1987	Dynamic/ Distributed	Type-1+3+ Bad Type-2	Type-1+Bad Type-2
Dynamic Tree Splitting [44]	1988	Dynamic/ Distributed	Type-1+3+ Bad Type-2	Root+Bad Type-2
Bound-and-Branch [32]	08/1988	Dynamic/ Distributed	Type-1+3+ Bad Type-2	Type-1+Bad Type-2
Delayed Branch Tree Expansion [41]	1990	Static/ Centralized	Type-1+3	Root + Bad Type-2
Frontier Splitting [57]	1993	Dynamic/ Distributed	All	Root
$\alpha\beta$* [25]	1993	Dynamic/ Distributed	Type-1+3+ Bad Type-2	Type-1+3+ Bad Type-2
CABP [24]	1994	Static/ Centralized	Type-1+3	Root + Bad Type-2
Jamboree [53]	1994	Dynamic/ Distributed	Type-1+3+ Bad Type-2	Type-1+Bad Type-2
ABDADA [95]	1995	Dynamic/ Distributed	Type-1+3+ Bad Type-2	Type-1+Bad Type-2
Dynamic Multiple PV-Split [62]	1995	Dynamic/ Distributed	Nodes within PV set	Nodes within PV set

Table 3.1: Comparison of Parallel $\alpha\beta$-based Game-Tree Search Algorithms

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 in the game tree where parallelism could occur. The critical game tree in Figure 2.8 can be used to define where the parallelism can occur. However, the critical tree is a perfect tree. In some cases, a type-2 node may not necessarily have a move which causes a cutoff as its first branch. Thus, the type-2 nodes have been separated 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 nodes, while the Young Brothers Wait algorithms allow for parallelism at type-1, type-3 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, the root of the game tree and 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.

It is important to note that these are not necessarily global synchronization points as mentioned in Section 3.2. For the algorithms given here, synchronization at the root or at type-1 nodes are global synchronization points.

3.3.2 Comparison of the $\alpha\beta$ Implementations

Table 3.2 summarizes an implementation of each algorithm given in Table 3.1.

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.

Algorithm (Reference)	Hardware Used	Test Domain	Sequential Algorithm	Trans-Position Table	Speedup Obtained
Parallel Aspiration Search [6]	Simulation	Artificial Trees	$\alpha\beta$	none	≤ 6 for large n (simulated)
Mandatory Work First [2]	Simulation	Artificial Trees	$\alpha\beta$	"score table"	≤ 6 for large n (simulated)
Tree Splitting [34]	LSI-11 & Simulation	Checkers	$\alpha\beta$	none	2.34 (n=3) 5.12 (n=27,sim)
PV-Split [63]	Sun 3 Network	Chess	PVS	local	3.75 (n=5)
Key Node [55]	Simulation	Artificial Trees	$\alpha\beta$	none	12.57 (n=20)
UIDPABS [69]	Data General (mixed procs.)	Chess	$\alpha\beta$	local	3.94 (n=8)
DPVS [81]	Sun 3 Network	Chess	NegaScout	local + TT Manager	7.64 (n=19)
EPVS [45]	Sequent Balance	Chess	$\alpha\beta$	shared memory	5.93 (n=16)
Waycool [31]	Hypercube	Chess	$\alpha\beta$	distributed messages	101 (n=256)
Young Brothers Wait [29]	Transputers	Chess	NegaScout	distributed messages	142 (n=256) 344 (n=1024)
Dynamic Tree Splitting [44]	Cray C916	Chess	$\alpha\beta$	shared memory	11.1 (n=16)
Bound-and-Branch [32]	Hypercube	Othello	$\alpha\beta$	distributed messages	12 (n=32)
Delayed Branch Tree Expansion [41]	Simulation	Chess	$\alpha\beta$	none	350 (n=1000, simulated)
Frontier Splitting [57]	BBN TC2000	Checkers	NegaScout	shared memory	3.32 (n=16)
$\alpha\beta*$ [25]	Transputers	Chess	NegaScout	distributed messages	6.5 (n=8+8TT)
CABP [24]	Sequent Balance	Artificial Trees	$\alpha\beta$	shared memory	4.6 (n=9)
Jamboree [53]	CM-5	Chess	NegaScout	distributed messages	$\approx$50 (n=512)
ABDADA [96]	CM-5	Chess (& Othello)	NegaScout	distributed messages	15.85 (n=32)
Dynamic Multiple PV-Split [62]	AP-1000	Artificial Trees	PVS	none	$\approx$32 (n=64)

Table 3.2: Comparison of Parallel $\alpha\beta$-based Game-Tree Search Implementations

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 were not generated by game-playing programs, they are considered to be *artificial trees*. As discussed earlier, 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: $\alpha\beta$, PVS or NegaScout. 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 use a shared-memory transposition table, but it is generally faster than distributed transposition tables based on message passing. *Local* transposition tables are maintained separately on each processor, and no transposition table information is shared between the processors.

The final column gives the observed speedup of the implementation on a large number of processors. As warned in Section 3.2.1, it is nearly impossible to compare speedups or efficiencies of tree-searching algorithms.

3.3.3 Summary of $\alpha\beta$ Algorithms

Baudet's thesis described a method of doing *parallel aspiration search* [6]. As described in Section 2.3.1, aspiration search reduces the size 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 thesis, the initial $\alpha\beta$ window is subdivided into n disjoint windows. 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 were the first to propose and simulate a *mandatory work first* algorithm [2]. The idea of the algorithm is to explore in parallel those leaves that would be examined if the game tree was perfectly ordered. There are two categories of nodes that correspond to the type-2 and type-3 nodes from the critical tree. Left-hand nodes are similar to type-3 nodes, and all of their successors are evaluated at different processors in parallel. Right-hand nodes are similar to type-2 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 introduced the concept of *tree splitting* [34]. 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 [21, 61] is a natural extension of tree splitting, based on the regular structure of the critical game tree as we travel along the principal variation (PV). The first stage of the algorithm involves a recursive call to itself as PV-Split 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 completely 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 [61], 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 trees. There are several published experiments with the PV-Split algorithm [63, 64, 68]. The best reported efficiency in these implementations was a speedup of 3.75 on 5 processors [63]. 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.

The *Key Node* method [55] 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 related 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 algorithm, *Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search* [69], 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 [81] is an enhancement in the PV-Split framework that allows for dynamic processor trees. 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 type-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 is balanced by 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 transposition table requests near the root of the game tree, along with a mechanism for rebroadcasting history heuristic information, the speedup for the chess program 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 [43, 45] is a different type of dynamic allocation to the PV-Split algorithm. In the Enhanced PV-Split (EPVS) algorithm, when a processor becomes idle, all of the other processors are stopped and a new split node is created two ply further down the tree of one of the busy processors. All processors then start to work on the smaller subtree. The transposition table ensures that the smaller subtree has 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.

Felten and Otto [31] 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 were hierarchically organized into a tree structure at the start of the search, but this processor tree could 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 [29, 30, 94]. 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 a type-3 node if we evaluate the leftmost branch and 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 3.1 and 3.2 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 [29].

YBWC* used a request work message sent to a random processor to achieve good load balancing on a network of Transputers. As well, the slow node-processing speed of ZUGZWANG allowed the Transputers to be used effectively as a giant shared-memory hash table, further increasing the quality of the move ordering.

Hyatt introduced *Dynamic Tree Splitting* (DTS) in his Ph.D. thesis [43]. 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 type-3 node that is closest to the leaf. 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 a branch 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. The scheme effectively removes the synchronization points from most type-1 nodes other than the root of the game tree.

In the implementation, split points were placed in shared memory so that other processors had an opportunity to take branches without disturbing other processors. Using a Cray C916/1024, DTS generated an average speedup of 11.1 on 16 processors when searching a series of positions from a chess game [44].

Bound-and-Branch [32] 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 assisted by iterative deepening and a distributed "game-tree representation", similar to a transposition table.

Hsu described a *queued processor array model* for implementing a parallel $\alpha\beta$ algorithm within the second version of DEEP THOUGHT [41]. The host workstation traverses the tree according to the algorithm until the parallelization horizon is reached. The subproblems (nodes on the parallelization horizon) are then placed on 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 into the host processor. The results are then added to the tree representation in the host computer.

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

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 reported that a speedup of 350 with 1000 processors is possible, once the machine is completely constructed.

Lu [57] 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, *frontier splitting* was proposed and tested. Frontier splitting creates new split nodes closer to the root of the variation being explored by the Controller process, as 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 type-1 and type-3 nodes, and only at type-2 nodes when there is no other parallelism left. 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 because the PV-Split algorithm will not generate a speedup larger than the branching factor.

David's $\alpha\beta*$ [25] is a new type of architecture for game-tree search 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 table 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 [95]. 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 [24], similar to the Deep Thought design presented by Hsu. 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. Unlike

Hsu's work, the critical nodes are evaluated first, followed by the non-critical children once the first queue has been finished. 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 [53] 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 [95]. Weill suggested and tested a decision method based on the Young Brothers Wait Concept, 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 is not 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 heuristic 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 [96], the combined method was called *Alpha-Bêta Distributé avec Droit d'Aînesse*, or ABDADA. Weill 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 [62]. To understand the algorithm, it is necessary to define the *PV set*. The root is a member of the PV set. At subsequent depths in the tree, 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 based on the depth of node in the game tree, and is not necessarily a fixed number. Thus, the PV set is a right-pruned version of the game tree. An appropriate function 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 critical tree, the last two columns in Table 3.2 reflect this by referring to the PV set, and not Knuth and Moore's classification of critical 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.

3.4 Other Parallel Game-Tree Search Approaches

Although $\alpha\beta$-based methods have been emphasized, there are other search strategies that can be used to generate the minimax value or a move decision on multiple processors. The four subsections cover the original formulation of SSS*, conspiracy numbers, the ER method, and theoretical models for game-tree search.

3.4.1 SSS*

A parallel SSS* algorithm was proposed by Campbell [21, 22], 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 [54] proposed the HYBRID algorithm, based on the problem heap from SSS*. However, the paper contains no details pertaining to an implementation. Vornberger and Monien presented the results of a parallel SSS* algorithm [94], 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.

Shinghal and Shved [85] propose the MDSSS algorithm, which implements a PV-Split constraint at the root of the game tree before executing DUAL* on the children of the root. The simulations in the paper show that MDSSS, for simulated game trees, searches less nodes in parallel than the other algorithms, including PV-Split and Mandatory Work First. However, no attempt was made to compare the total time required to run the SSS* algorithm in practice to implementations of $\alpha\beta$. Diderich described an implementation of Synchronized Distributed State Space Search (SDSSS*) in an attempt to balance the workload on many distributed processors [26]. SDSSS* 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 Usui *et al.* [92], and Kraas [50] which 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, the overhead of dealing with the problem heap is not necessary. It has been recently discovered that $SSS*$ can be implemented as a series of null-window $\alpha\beta$

calls, using a transposition table instead of an OPEN list [74]. Until the overhead of distributing the ordered problem heap can be brought below that of the depth-first search algorithms presented earlier in the thesis, the author does not believe that these implementations are practical choices for game-tree search.

3.4.2 Conspiracy Numbers

Section 2.5.3 dealt with the sequential version of the Controlled Conspiracy Number Search (CCNS) algorithm. Since CCNS is a depth-first search instead of a best-first search, a parallel CCNS algorithm can borrow ideas from other parallel depth-first search algorithms. Lorenz and Rottmann introduced Parallel CCNS [56], an algorithm that is very similar to the Young Brothers Wait algorithm. Processors with no work use work stealing to ask a processor for work. A node that has not been evaluated by a processor can be transferred to a processor for evaluation, establishing a master/slave relationship. When the slave processor determines whether it can or cannot satisfy the proof required, the result is returned to the master processor. Parallel CCNS effectively speeds up the search by a factor of 36 on 63 processors, when compared to a sequential CCNS algorithm. Although it is hard to compare CCNS searches to $\alpha\beta$-based searches, the observed speedup is impressive.

3.4.3 ER

Steinberg and Solomon [88] presented the ER method of searching game trees. ER stands for Evaluate-Refute, and the method 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. When compared to sequential ER search, the parallel ER algorithm achieved a speedup of 10 with 16 processors, and a speedup of 14.7 with 27 processors.

3.4.4 Theoretical Methods

There are a number of theoretical algorithms that, to the author's knowledge, have not been programmed. Karp and Zhang [46] proposed the Parallel α-β algorithm which yields a linear speedup on trees of height $O(n)$. Althöfer [3] proposed an algorithm which yields a linear speedup on average if the tree height is $O(n \log n)$. Broder *et al.* [17] have shown that for any parallel tree searching algorithm, there exists a tree instance that does not run in polylogarithmic parallel run-time[1] on a large number of processors. Broder *et al.* also describe the ParHope algorithm and prove a bound on its performance over sequential prefix-driven algorithms such as $\alpha\beta$.

3.5 Parallelism in Single-Agent Search

The goal in a single-agent search problem is to find a path from an initial state S to a goal state G with the minimum possible cost. Although this problem seems different from the determination of the minimax value of a game-tree, the two fields are closely related.

An example of a problem that can be solved with single-agent search is the sliding-tile puzzle. Figure 3.1 gives an example of a typical start and goal state for the sliding-tile puzzle. The only move available is moving a single tile horizontally or vertically into the blank space. The goal is to move from the start state to the goal

[1]Polylogarithmic parallel run-time implies that there exists constants k, l such that on n^k processors, every instance runs in $O((\log n)^l)$ time.

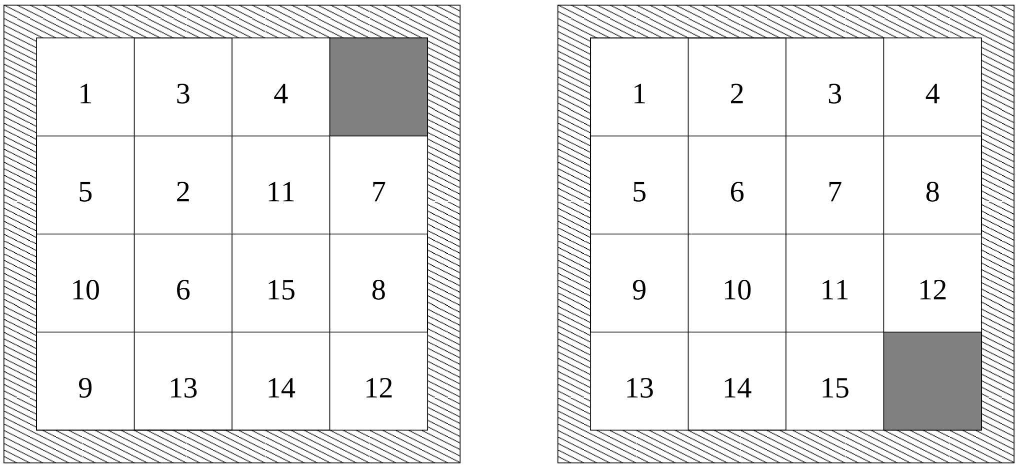


Figure 3.1: A Start and Goal Position for the Sliding Tile Puzzle

state in the minimum number of moves.

3.5.1 A*

The A* search algorithm [40, 71] is often used to solve single-agent search problems. A* is a best-first algorithm, similar to SSS* in the domain of game-tree search.

A* search starts with the initial state in a main data structure known as the *OPEN list*. The *CLOSED list* represents the positions that we have already examined, and is initially empty. For each node within the OPEN and CLOSED lists, A* maintains two heuristic values: $g(n)$, the best-known minimum cost, and $h(n)$, the estimate of the cost to a goal state. Thus, the best node to examine at any point in the algorithm has the lowest estimated total cost: $f(n) = g(n) + h(n)$.

The A* algorithm is an iterative process. In each step, A* takes the best state s from the OPEN list and moves it to the CLOSED list. The successors of the best state, s_i, are generated and are examined in turn. If a successor s_i does not appear in either the OPEN or CLOSED list, then s_i is added to the OPEN list. However, if s_i already appears in either list, we must check to see if the minimum cost $g(n)$ has decreased. If $g(n)$ decreases, the node s_i must be deleted from its current location and reinserted into the OPEN list.

The heuristic $h(n)$ is critical for the performance of the A* algorithm. $h(n)$ is said to be *admissible* if the heuristic never overestimates the cost of travelling to the goal state. If $h(n)$ is admissible, A* is guaranteed to generate the least cost or optimal solution the first time the goal node is generated. In the sliding-tile puzzle, the Manhattan distance [2] is an admissible and effective heuristic for use in A* search.

As in our discussion of $SSS*$ searches, the emphasis of parallelizing A* searches generally relies on how to distribute the OPEN and CLOSED lists amongst processors in an effective manner. In a centralized scheme, contention for the OPEN and CLOSED lists becomes a serious bottleneck. However, the search overhead can be limited to a small amount. Even in distributed schemes which share information through a blackboard structure [52], or by message passing at numerous synchronization points [42], the memory requirements of the best-first A* algorithm rapidly becomes excessive on non-trivial problems.

3.5.2 IDA*

One does not necessarily need to use a best-first algorithm to perform single-agent search. Korf discovered that iterative deepening works as well in single-agent search as it does in game-tree search [49]. Instead of a memory-intensive OPEN list, IDA* search is based on depth-first searches, and iterates on successively larger lower bounds for the total cost $f(n)$. IDA* also returns an optimal solution if used with an admissible heuristic $h(n)$. In practice, IDA* is preferred over A* for the smaller memory requirements.

The parallel algorithms that have been proposed for IDA* search are similar to $\alpha\beta$-based parallel search algorithms. This should not be surprising since the underlying applications are tree searches with natural synchronization points between iterations of iterative deepening.

Rao, Kumar and Ramesh [77] illustrated the first synchronized search of the state-space using IDA*. Their algorithm, PIDA*, partitioned the space amongst all of the

[2] The sum of the vertical and horizontal displacements of each tile from its current square to its goal square.

processors and let them search independently of one another. As processors finish their work for an iteration, they ask neighbouring processors if they need assistance finishing their iteration. Eventually, all processors run out of work and the next iteration is started. This continues until one of the processors discovers the goal node, and the search is subsequently terminated. PIDA* achieves an average speedup of 28 on 30 processors of a Sequent Balance 21000.

IDPS by Mahanti and Daniels [58] and SIDA* by Powley, Ferguson and Korf [76] are similar algorithms for handling parallel IDA* search on SIMD machines. Both algorithms partition a set of frontier nodes to each processing element. Each processing element uses depth-first search independently. When a number of processors go idle, a load balancing algorithm is used to repartition the work over all the available processors. Eventually, all processors finish their work for a given iteration, and the search then continues to the next iteration. As in PIDA*, one of the processors will eventually discover the goal node, and the search is immediately terminated. A 16K processor CM-2 yields efficiencies of 57% for SIDA* and 76% for IDPS.

Reinefeld and Schnecke took the ideas from these algorithms and implemented an asynchronous algorithm for handling IDA* search [80]. Asynchronous IDA*, or AIDA*, works in a similar manner to all IDA* algorithms when distributing the work. However, only weak synchronization is used to inquire about pieces of work for load balancing purposes. Until new work arrives for the current iteration, the processor is allowed to continue to the next iteration. This scheme keeps the processors working on approximately the same iteration.

AIDA* yields a efficiency of 79% for the 25 problems generating the largest search trees from Korf's data set [49] on a 1024-node Transputer system. Although the efficiency may look small, Korf's sample positions take too little time to test AIDA* adequately. AIDA* takes 24.2 minutes to generate the results for all 100 of Korf's test cases, 5.7 times faster than SIDA* on 32,768 nodes of a CM-2.

3.5.3 Comparing Parallel IDA* and Parallel $\alpha\beta$

It is often wondered why $\alpha\beta$-based game-tree search is so hard to parallelize, while IDA* search is relatively straightforward. Even with the small branching factor of the 15-puzzle, all of the IDA* algorithms described have remarkable efficiencies on massively parallel systems. There are a number of important differences between IDA* search and $\alpha\beta$-based game-tree search.

IDA* has almost no pruning whatsoever. Thus, the search is similar to minimax search, and not to $\alpha\beta$ search. It is relatively easy to get a high efficiency when searching without any pruning techniques. When an improved $\alpha\beta$-based algorithm such as MTD(f) is used, the tree is pruned aggressively and severe load imbalances can occur.

Furthermore, when using $\alpha\beta$ and the exact value of the first child is not known, many additional nodes must be searched. Although we can generate a guessed minimax value, a small measure of uncertainty is often sufficient to cause a large increase in the number of nodes searched. Thus, the work is usually synchronized on determining the minimax value of the first child before allowing full parallelism at any node within the game tree.

Positions in a search tree are generally independent of one another in IDA*; although IDA* can benefit from using a shared transposition table, it does not seem to be a large benefit for Korf's 15-puzzle benchmarks. The equivalent is not true for some game-tree search domains; chess game trees can suffer a large increase in search effort if local transposition table information is not shared.

Finally, IDA* searches until it finds a solution; it is not time constrained. Game-tree search must continually deal with real-time constraints: a move decision must be made within t seconds. Thus, the game tree generated is not as deep as an IDA* tree, and does not yield as many natural opportunities for parallelism.

In the author's opinion, these reasons are what make $\alpha\beta$-based game-tree search more challenging to parallelize than IDA* search.

3.6 Parallel Search Libraries

There are a few parallel search libraries in the literature. However, most of the libraries, like PPBB-Lib [91], concentrate on implementing parallel branch-and-bound solutions in an efficient manner. The interface has specific information for defining subproblems, new bounds and parallel I/O support, as well as support for many different load-balancing schemes. The system uses PVM [35] as its underlying method of communication.

The ZRAM parallel search bench [19] takes a more ambitious approach to developing a general tool to be used by researchers. Many search engines are available within ZRAM, including branch-and-bound, reverse search and backtracking. A virtual machine layer allows for dynamic load balancing, as well as checkpointing and termination detection. The ZRAM method uses the point-to-point communication of MPI [39] to send messages between processes. The ZRAM parallel search bench has been used to show that the 15-puzzle takes at most 80 moves to solve any position, as well as prove conjectures made in other fields such as materials science. However, ZRAM does not yet have a parallel game-tree search engine.

3.7 Conclusions

The taxonomy given in Section 3.3 shows that there are a number of implementations which are using the same underlying algorithm. A large number of $\alpha\beta$-based algorithms are employing minor variations of the same technique. 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.

There are other parallel depth-first search algorithms that are not based on $\alpha\beta$, such as IDA*. The study of parallel IDA* search algorithms poses an interesting question: are synchronous algorithms better than asynchronous algorithms for implementing depth-first search strategies? We will investigate this issue in game-tree search in the subsequent chapters.

Chapter 4

Theoretical Comparison

4.1 Introduction

In this chapter, we will compare an asynchronous game-tree search algorithm to the typical synchronous game-tree search algorithm using a theoretical model. In Section 4.2, we will describe the computational model used to analyze the sequential and parallel performance, as well as the method of generating a theoretical speedup for a d-ply game-tree search using iterative deepening. Section 4.3 deals with modeling game trees, and how to make a more realistic model through the use of empirical evidence. Section 4.4 describes and analyzes the typical synchronous parallel game-tree search algorithm used today. Section 4.5 describes and analyzes an asynchronous game-tree search algorithm. Theoretical parallel speedups are shown for both algorithms using two sequential tree models. Finally, Section 4.6 attempts to compare and contrast the two algorithms, and shows that asynchronous game-tree search has the potential for greater speedups than synchronous game-tree search in realistic game trees.

The caveats mentioned in Chapter 3 regarding the use of theoretical or simulated results when comparing observed speedups should be noted. The theoretical results given here are *not* intended to indicate what can be achieved in practice but, rather, are a conservative estimate of the best results we can hope to obtain. Many of the real overheads that exist in practical parallel processing are missing from the computational models used. However, the results are intended to give the reader an idea of how synchronous and asynchronous game-tree search algorithms compare in a theoretical framework.

4.2 Experimental Setup

4.2.1 Computational Model

The model of computation used in this chapter is the *leaf-evaluation model* [17, 46]. In this model, each node of a game tree that is evaluated takes one unit of time, and all other costs are considered to be negligible. The number of leaves that a sequential algorithm evaluates determines its running time. A parallel algorithm can have up to n processors each evaluating a separate leaf within a single unit of time. The number of steps required to evaluate all of the leaves with n processors yields the parallel time.

One of the assumptions of the leaf-evaluation model is that the parallel and sequential algorithms evaluate the same number of nodes per second. In practice, implementing a parallel algorithm may reduce the average number of leaf evaluations per second by a small amount (up to 10%). This effect will be ignored for the results in this chapter.

4.2.2 Methodology

Once we define a parallel algorithm, how can we derive a theoretical speedup? In general, a program that executes a d-ply game-tree search uses iterative deepening (Section 2.3.2) in steps of 1 ply; the k-ply search tree is completed before searching the $(k+1)$-ply game tree. Thus, a search of a d-ply search tree involves d searches of the game tree to successively greater depths. If we have a model for the number of leaves we must evaluate in the sequential case for a k-ply tree, along with the formula for the expected speedup, we can compute the time required to search the k-ply tree in parallel. By summing each of the iterations of iterative deepening from 1 to d ply, we can generate a total sequential time, a total parallel time and a theoretical speedup for the complete d-ply iteratively-deepened game-tree search.

The expected theoretical speedup for a d-ply iteratively-deepened search can be determined mathematically. However, the summation of the d iterations to generate an overall speedup yields a very complex formula. Instead of attempting to give this formula exactly, we can use a calculator or other process to compute the sum of the parallel and sequential times and, thus, the expected theoretical speedup. For this purpose, we have written a small program to complete the summation for us. It is important to note that this program does not actually simulate or execute a real game-tree search.

As we shall see in the subsequent sections, the parameters required to determine the expected theoretical speedup are the branching factor of the tree, the depth that the tree must be searched to, and the number of processors involved in the parallel search. The type of tree to be searched and the parallel algorithm to be used are implicitly defined by the parallel speedup formula.

We wish to simulate different games that we are interested in (such as checkers, Othello and chess). For these games, there are well-established average branching factors: 38 for chess [36], 10 for Othello [74], and 3 for checkers [57]. These three branching factors will be examined throughout this chapter. The two processor configurations that will be examined are $n = 16$ and $n = 256$, which illustrate typical small and large parallel configurations.

To compare speedups generated by different processor configurations, graphs will be plotted that use the parallel efficiency as the vertical axis. We use the number of nodes searched sequentially on the horizontal axis, so that we can compare trees that have different branching factors. In this way, we can compare the searches based on all of the implicit and explicit parameters of the software model.

4.3 Modeling Game Trees

There are many possible theoretical models for game trees. The probabilistic strongly-ordered tree model [59] is the most frequently used model when simulating game trees. In this model, a strongly-ordered game-tree can be generated by ensuring that for every node in the tree, there is a fixed percentage chance that the best move (with respect to the minimax value) will be the first child examined. With this model, we can vary the strength of the move ordering, and simulate many different types of game trees by varying the percentage, the depth and the branching factor of the tree. However, from a theoretical point of view, this would be complicated to analyze without a simulation program, both in the sequential and the parallel case. Thus, we opt for models of game trees that are easier to analyze without resorting to simulated search.

One such model of the game tree, the perfect critical tree (as illustrated in Figure 2.8), is ideal for its relative simplicity and wealth of published results. There are some implicit assumptions in the perfect critical tree: (1) the game tree has a uniform branching factor b, (2) we always search the best move first, (3) there are no duplicate nodes within the tree, and (4) the search window at the root of the tree contains the minimax value.

However, the perfect critical tree is a poor example of what we can expect in practice, and drawing conclusions based on this unrealistic model are somewhat suspect. For the same nominal depth of search and branching factor, game trees generated by $\alpha\beta$ can differ in size by a factor of 100 or more. Let us say that we are interested in investigating subtrees of d' ply. We cannot know, *a priori*, how many nodes $\alpha\beta$ must search for a given d'-ply subtree. So, the question is: what is the typical probability distribution of the size of subtrees searched by $\alpha\beta$?

In theory, the branching factor estimates given earlier can vary significantly from the estimate, and are correlated to the game state. If we ignore the correlation to the game state, the probability distribution does not affect the expected size of these $\alpha\beta$ trees. Irrespective of the probability distribution, the expected size of a d'-ply

$\alpha\beta$ tree will be the expected size of a $(d'-1)$-ply $\alpha\beta$ tree, plus $(b-1)$ times the expected size of a $(d'-1)$-ply $\alpha\beta$ refutation (where b is the average branching factor). Although the expected size is the same, there is a greater probability of generating a smaller tree than a larger tree, because there are fewer random choices at interior nodes required to generate a small tree. This yields a distribution of tree sizes with a positive skew.

How does the theory vary in comparison to practice? We can obtain empirical evidence on the size of d'-ply game trees through an experiment. If we take a game-tree searching program, and instrument it to capture the size (in terms of number of bottom positions) of each d'-ply subtree, we can generate a histogram of the distribution of the size. In this experiment we used KEYANO, CHINOOK, and THETURK as representative samples of Othello, checkers and chess programs. We measured the size of $d' = 4$ ply trees in KEYANO and THETURK, and $d' = 6$ ply trees in CHINOOK[1]. Each program searched all of the positions in the appropriate test set from Appendix A to 8 ply in THETURK, 9 ply in KEYANO, and 13 ply in CHINOOK. These depths are roughly double the ply of the subtrees being examined for each application. This allows us to generate a sufficiently large number of d'-ply subtrees within the search of each full game tree.

Tree sizes were organized into buckets covering a range of 10 tree sizes for the purposes of drawing a histogram. For example, trees with 30 to 39 leaf nodes evaluated are represented by a single point on the graph. Examining the histograms in Figures 4.1 show that there is a positive skew to each of the histograms. The histogram has been normalized for the number of samples being used (approximately 30,000 samples for CHINOOK and KEYANO, and over 80,000 for THETURK) to generate a sample probability distribution function. The positive skew and visual inspection indicates that a single probability density function that can accurately model each histogram may be either a Poisson or gamma probability distribution function.

[1] $d' = 4$ ply game-trees in checkers are too small to be analyzed with the same methodology used for the chess and Othello trees.

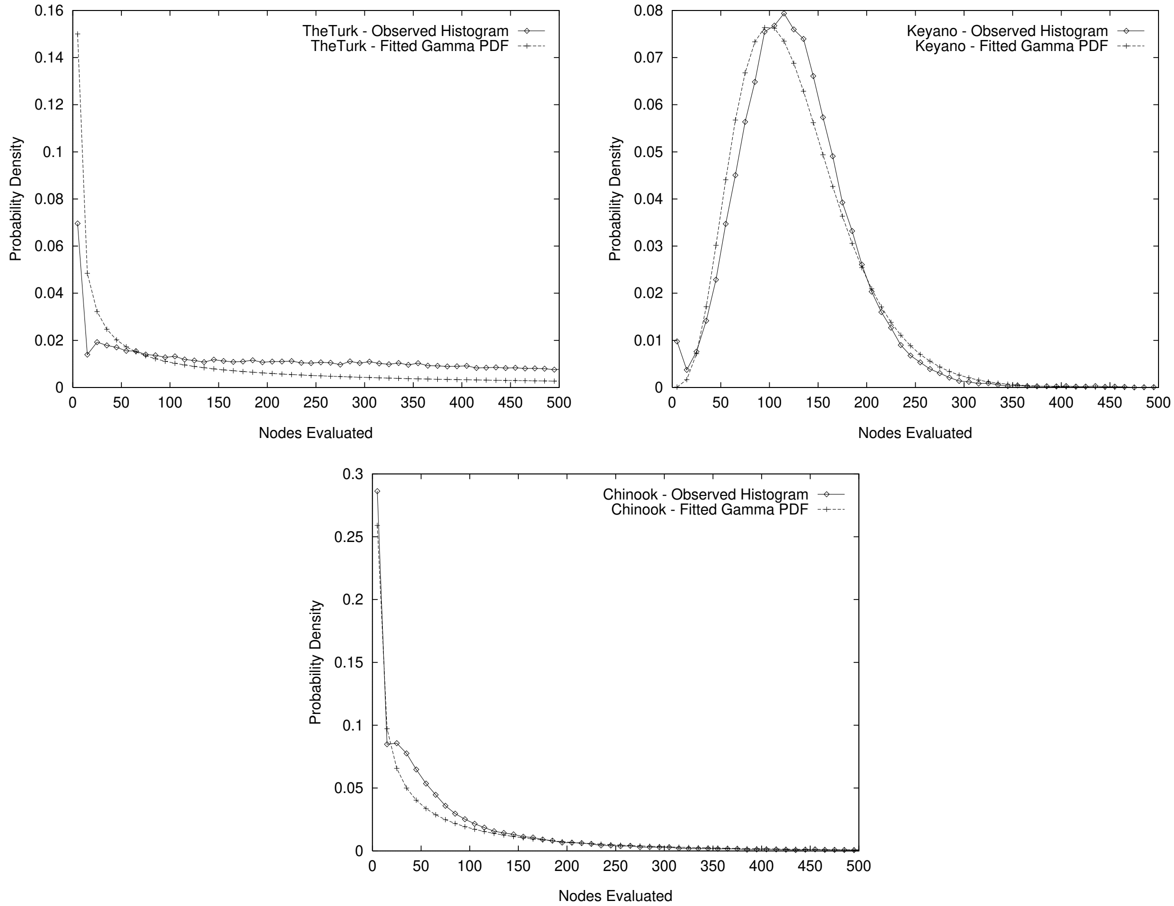


Figure 4.1: Sample Histograms and Predicted Probability Density Function

It should be noted that the histogram for KEYANO looks different than the histogram for CHINOOK and THETURK, and that the vertical scales are different for each program. The reason behind this is that the full version of the program with transposition tables enabled was used during the test. Chess and checkers programs greatly benefit from the duplicate detection property of the transposition table, in comparison to Othello programs[2]. This accounts for the large spike at the leftmost data point for chess and checkers, and the missing spike in the Othello program's histogram. When the spike is removed from the data, and the data is scaled so that each expected tree size is roughly the same, all three graphs look similar to the graph

[2]Moves in chess and checkers have only a limited effect on the board position, making it likely that the same moves, if played in a different order, can yield the same position. In Othello, a given move can flip a large number of discs, dramatically changing the board position. Thus, it is less likely in Othello for a specific position to reoccur by transposing moves.

for Keyano.

We can use the sample moments to determine probability density function parameters for each histogram. The second line on each graph in Figure 4.1 represents the fitted gamma probability distribution function based on the calculation of the sample moments.

Varying the level d' does not significantly alter the histogram in each program tested, once the different scale of the expected tree size is taken into effect. Thus, we can use the fitted gamma probability density function to determine a similar distribution of the size of k-ply subtrees for each program. The fitted gamma probability density function can be scaled so that the expected average size of k-ply subtrees equals the size of k-ply subtrees in the critical tree.

To generate a *realistic critical tree* from the perfect critical tree, we will allow k-ply subtrees at the bottom of the game tree to vary according to the scaled gamma probability density function. Note that the top $(d - k)$ levels of the tree are exactly the same as the perfect critical tree. We will be using the fitted gamma probability density function that we determined from TheTurk, Keyano and Chinook when the branching factor equals 38, 10, and 3, respectively.

From a sequential viewpoint, the realistic critical tree and the perfect critical tree evaluate exactly the same number of leaves. Hence, the two trees take the same time to evaluate sequentially. In parallel search, this size distribution of k-ply trees can give us a realistic look at the amount of time spent at a global synchronization point. We shall see how this delay can be accounted for in the subsequent analysis of synchronous and asynchronous game-tree search.

4.4 Analysis of Synchronous Game-Tree Search

The synchronous $\alpha\beta$-based parallel game-tree search algorithm that we will examine has been implemented independently by many authors. In Young Brothers Wait [29], Dynamic Tree Splitting [44], Bound-and-Branch [32], Delayed Tree Branch Expansion

[41], Jamboree [53], and ABDADA [96], the basic ideas on where to create parallelism and where to synchronize the search are based on the critical tree (Figure 2.8). The first child must be completely evaluated at a type-1 node so that an alpha-beta bound can be discovered for the subsequent children. Once that is completed, all of the remaining children of a type-1 node can be evaluated in parallel. Probable type-3 nodes have all of their children evaluated in parallel. Probable type-2 nodes have their children evaluated (in general) without parallelism.

For the purposes of this analysis, we will assume that the algorithm completely synchronizes at each node along the principal variation. Some of the algorithms mentioned earlier have attempted to remove the global synchronization points along the principal variation. That is, the algorithm only needs to synchronize at the root of the game tree. However, the algorithm that has reported the best results on a massively parallel system, Young Brothers Wait, uses synchronization at every node along the principal variation. Thus, we have decided to use that type of algorithm in our analysis.

It is also important to note that the parallel algorithms listed have different schemes to determine when a probable type-2 node should be handled as a type-3 node. These schemes are not relevant to the analysis, since there are no type-2 or type-3 nodes in unanticipated places for the game tree models being used.

4.4.1 Perfect Critical Tree Model

In this analysis of parallelism in the perfect critical tree, we will review the total number of nodes, and the size and number of work granules underneath a given type-1 node. We will then determine formulas for the theoretical parallel speedup in two cases, depending on whether the number of processors or the number of work granules is larger. This section will end with an illustration of theoretical parallel efficiencies based on the speedup formulas determined.

The number of nodes of each type within the perfect critical tree can be found in the literature [47]. At ply d of a critical tree with uniform branching factor b, there is exactly one node of type-1, $b^{\lceil d/2 \rceil} - 1$ nodes of type-2, and $b^{\lfloor d/2 \rfloor} - 1$ nodes of type-3.

We can also determine the number of bottom positions for each type of node in the critical tree. Given a critical tree with uniform branching factor b and leaves that are d ply away from the node, the number of bottom positions underneath the node, depending on its type, are:

$$\begin{aligned} \text{NBP}_{\text{type-1}} &= b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1, \\ \text{NBP}_{\text{type-2}} &= b^{\lfloor d/2 \rfloor}, \\ \text{NBP}_{\text{type-3}} &= b^{\lceil d/2 \rceil}. \end{aligned}$$

Synchronous parallel game-tree search algorithms do not spawn work below a given size, since the size of the work piece may be too small to be worth examining in parallel. This *minimum granularity* is dependent on the hardware and properties of the application used. Let us assume that the minimum granularity is d'-ply; no work is given to another processor with d' or less depth. The minimum granularity can also be referred to as the *parallelization horizon*; work can be subdivided and studied in parallel above the horizon, but not below it.

Having reviewed the number of bottom positions and defined the parallelization horizon, we can discuss how the $(d-d')$ synchronization points affect the d-ply search. In between each global synchronization point, we must determine how many pieces of work are available and the size of each piece of work. Table 4.1 illustrates the available parallelism and the size of each piece of work available over varying distances to the minimum granularity.

The first row represents when we evaluate the type-1 node at the horizon: only one processor can execute the d'-ply search of the PV node at $(d - d')$ ply. Thus, we must use $\text{NBP}_{\text{type-1}}$ to compute the size of a d'-ply search of a type-1 node.

Once that initial search is finished, the search proceeds towards the root of the game-tree by evaluating the next PV node. Observe that if k is the number of levels within the game tree we are away from the minimum granularity, we have $b^{\lceil k/2 \rceil} - b^{\lceil k/2 \rceil - 1}$ nodes at $(d - d')$ ply that we can evaluate in parallel on separate

Ply To Horizon	Available Parallelism At Horizon	Size Of Work Granule At Horizon
0	1	$b^{\lceil d'/2 \rceil} + b^{\lfloor d'/2 \rfloor} - 1$
1	$b-1$	$b^{\lfloor d'/2 \rfloor}$
2	$b-1$	$b^{\lceil d'/2 \rceil}$
3	b^2-b	$b^{\lfloor d'/2 \rfloor}$
4	b^2-b	$b^{\lceil d'/2 \rceil}$
5	b^3-b^2	$b^{\lfloor d'/2 \rfloor}$
6	b^3-b^2	$b^{\lceil d'/2 \rceil}$

Table 4.1: Available Parallelism and Size of Work Granules for Type-1 Nodes in the Perfect Critical Tree

processors. This can be determined by considering the type-1 node as a type-3 node with only $b-1$ children. The first move is not counted at a type-1 node for determining the parallelism because it must be evaluated before parallelism can start at a type-1 node.

The type of nodes at the parallelization horizon alternates as we increase the depth away from the horizon. As illustrated in Table 4.1, the work granules are rooted at type-2 nodes at odd ply to the horizon, and at type-3 nodes at even ply to the horizon. Each of these granules of work at $(d - d')$ ply has either $b^{\lfloor d'/2 \rfloor}$ or $b^{\lceil d'/2 \rceil}$ bottom positions underneath them, depending on whether the nodes at $(d - d')$ ply are type-2 or type-3 nodes, respectively.

We are now ready to determine the theoretical speedup for a single d-ply fixed-depth critical tree with uniform branching factor. The analysis is complicated by whether there are more processors than pieces of work. For a depth d search with a d'-ply parallelism horizon, there are $b^{\lceil (d-d')/2 \rceil} - b^{\lceil ((d-d')/2)-1 \rceil}$ pieces of work. The first case to be analyzed is when there are more processors than pieces of work, or $n > b^{\lceil (d-d')/2 \rceil} - b^{\lceil ((d-d')/2)-1 \rceil}$. The second case is the converse: there are at least as many pieces of work as processors in the system. The first case is required to solve the second case, since the search commences with only one processor executing the search and eventually grows until all processors are involved.

Case 1: More processors than pieces of work.

We will define $\mathrm{SEQ}_{\mathrm{case1}}$ to be the amount of time required to search the d-ply critical tree. We are using the leaf-evaluation computational model, where each bottom position is equal to one unit of time. Thus,

$$\mathrm{SEQ}_{\mathrm{case1}} = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1.$$

In parallel, we have more processors than pieces of work. Thus, each processor gets at most one d'-ply granule from the young brothers of a given PV node. This piece of work takes all processors exactly the same amount of time to compute, before they all proceed in tandem to the next PV node. We define the search of the d-ply tree in parallel as $\mathrm{PAR}_{\mathrm{case1}}$ units of time. This is the same as the sum of the size of the work granules as we ascend from the principal variation from $(d - d')$-ply to the root:

$$\begin{aligned}
\mathrm{PAR}_{\mathrm{case1}} &= \sum_{k=d'}^{d} (\text{size of work granule from PV node at depth } (d-k)) \\
&= b^{\lceil d'/2 \rceil} + b^{\lfloor d'/2 \rfloor} - 1 + \sum_{k=d'+1}^{d} (\text{work granule at depth } (d-k)) \\
&\simeq b^{\lceil d'/2 \rceil} + b^{\lfloor d'/2 \rfloor} - 1 + ((d - d' - 1)/2)(b^{\lfloor d'/2 \rfloor} + b^{\lceil d'/2 \rceil}) \\
&\simeq (((d - d' + 1)/2))(b^{\lceil d'/2 \rceil} + b^{\lfloor d'/2 \rfloor} - 1).
\end{aligned}$$

Now, to calculate the speedup, $\mathrm{SPD}_{\mathrm{case1}}$, we divide the sequential time by the parallel time:

$$\begin{aligned}
\mathrm{SPD}_{\mathrm{case1}} &= \frac{\mathrm{SEQ}_{\mathrm{case1}}}{\mathrm{PAR}_{\mathrm{case1}}} \\
&\simeq \frac{b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1}{((d - d' + 1)/2)(b^{\lceil d'/2 \rceil} + b^{\lfloor d'/2 \rfloor} - 1)} \\
&\simeq \frac{b^{\lceil (d-d')/2 \rceil}}{((d - d' + 1)/2)}.
\end{aligned}$$

Now that we have calculated the approximate speedup when there are idle processors, we can calculate the speedup when there is work for every processor to execute.

Case 2: At least one piece of work per processor.

If we assume that $n > 1$, we know that there exists a point in the search where the available parallelism of a d''-ply sub-search is insufficient to keep all processors busy while the $(d''+1)$-ply search is sufficient to keep all processors busy. This point can be found where $d'' \simeq d' + 2\ \log n / \log b$.

The parallel speedup $\mathrm{SPD}_{\mathrm{case2}}$ can be computed in two parts. The speedup up to and including the d''-ply search, $\mathrm{SPD}_{\mathrm{case2a}}$, can be computed directly from $\mathrm{SPD}_{\mathrm{case1}}$, and the definition of d'' from the previous paragraph:

$$\begin{aligned}
\mathrm{SPD}_{\mathrm{case2a}} &= \sum_{k=d'}^{d''} (\text{size of work granule from PV node at depth } (d-k)) \\
&= \frac{b^{\lceil (d''-d')/2 \rceil}}{((d''-d'+1)/2)} \\
&\simeq \frac{b^{\lceil (d'+2(\log n/\log b)-d')/2 \rceil}}{((d'+2(\log n/\log b)-d'+1)/2)} \\
&\simeq \frac{b^{\lceil (\log n/\log b) \rceil}}{((\log n/\log b)+1/2)} \\
&\simeq \frac{n}{((\log n/\log b)+1/2)}.
\end{aligned}$$

The second part of the speedup (from $(d''+1)$ ply to d ply) has every processor investigating at least one work granule in between each global synchronization point. If we assume that we can achieve a perfect speedup when there are more pieces of work than processors, we can greatly simplify the calculation of the speedup. However, there may be an uneven distribution of work granules amongst the processors. If there are w pieces of work to be divided amongst n processors, the speedup will be equal to $\mathrm{SPD}_{\mathrm{case2b}} = w/\lceil w/n \rceil$.

Since $w > n$, we can determine two constants c and k such that $w = cn + k$, with $c > 1$ and $0 \le k \le n-1$. If k=0, $\mathrm{SPD}_{\mathrm{case2b}} = (c \times n)/\lceil (c \times n)/n \rceil = n$. This is not surprising, since each processor gets an even number of pieces of work to be analyzed.

Thus, we will examine the case where we have an uneven workload on each processor, or $k \neq 0$.

When $k \neq 0$, we can compute a lower bound on the speedup:

$$\begin{aligned} \mathrm{SPD}_{\mathrm{case2b}} &= (c \times n + k)/\lceil (c \times n + k)/n \rceil \\ &= (c \times n + k)/(c+1) \\ &\geq (c \times n + 1)/(c+1), \text{ since } k \geq 1. \end{aligned}$$

Since $c \geq 1$, this gives us a lower bound on the speedup of $(n+1)/2$. As the depth of search increases, both w and c increase by a factor of b every 2 ply. Thus, as the search depth increases, the lower bound on $\mathrm{SPD}_{\mathrm{case2b}}$ rapidly approaches n. Even as the lower bound approaches n, it is a bad approximation of the real speedup. For the majority of values of c and k, $\mathrm{SPD}_{\mathrm{case2b}} \approx n$. Furthermore, since the leaf-evaluation model neglects a number of overheads, the theoretical speedup is really an upper bound on the achievable speedup. For these reasons, it makes sense to approximate $\mathrm{SPD}_{\mathrm{case2b}}$ for each search from $(d''+1)$ ply to d ply with the upper bound, or $\mathrm{SPD}_{\mathrm{case2b}} = n$.

Using the speedups for the two parts of Case 2, we can generate parallel times for each part of the search, and add the results together to generate a total parallel time and speedup for a single d-ply search.

Now that we have determined the speedup for the two cases, we can combine a series of fixed-depth searches (from 1-ply to d-ply) to generate the total sequential time and total parallel time required for an iteratively-deepened d-ply search. In turn, this determines the theoretical parallel speedup for the iteratively-deepened d-ply search.

Before we analyze the formulas for different branching factors and processor configurations, we must first define the minimum granularity. In the author's experience, sending a piece of work to another processor that involves less than $1000 = 10^3$ node evaluations is not worthwhile because the communication costs are too high. We will define the minimum granularity d' for each simulation such that we do not send out

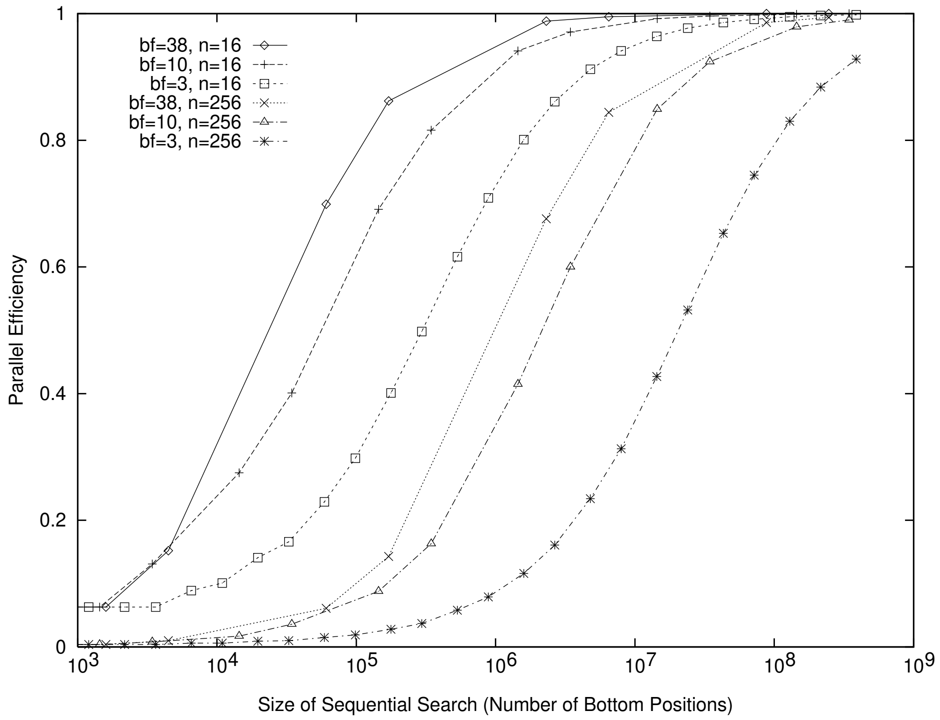


Figure 4.2: Synchronous Model, Efficiency on Perfect Critical Tree

work that is expected to be smaller than 10^3 node evaluations. It is important to note that the number of node evaluations for a piece of work to be considered worthwhile is not an absolute measure. A slower network or faster CPUs will cause the granularity to increase, while a faster network or slower CPUs will cause the granularity to decrease.

The graph in Figure 4.2 shows six lines. Each line represents one processor configuration with a particular branching factor. Each point on a particular line represents the theoretical expected efficiency for a parallel iteratively-deepened k-ply search. Since the graphs are normalized on the basis of sequential search size rather than depth of tree searched, we have many more data points for the curves where the branching factor is 3 (searched to a maximum depth of 33) than when the branching factor is 38 (searched to a maximum depth of 10).

If we varied the minimum granularity to be larger than 10^3 nodes, each of the lines on Figure 4.2 would retain its shape, but shift to the right. For example, if we increased the minimum granularity by a factor of 100, the size of the sequential search required to generate a given parallel efficiency would also increase by a factor of 100.

We can determine the expected speedup by multiplying the parallel efficiency by the number of processors used. For example, for a tree with 10^6 nodes evaluated, the model predicts an approximate speedup of $0.505 \times 256 = 129$ when the branching factor is 38, $0.350 \times 256 = 89$ when the branching factor is 10 and $0.086 \times 256 = 22.2$ when the branching factor is 3. If we allow the sequential search size to grow to 10^7 nodes, the model predicts speedups of 222, 200 and 91 for branching factors of 38, 10 and 3, respectively. It is important to note that this is a theoretical parallel speedup, and is not an accurate reflection of what can be achieved in practice. Although many chess programs can search 10^7 nodes during a regular search, none have regularly exhibited 222-fold speedups on 256 processors.

The graph in Figure 4.2 confirms two observed phenomena in synchronous game-tree search experiments. The first is that increasing the search depth increases the parallel efficiency of the search. The graph also confirms the observation that there is a wide disparity between the speedup of a typical chess tree search (branching factor of 38) versus the speedup of a checkers tree search (branching factor of 3).

4.4.2 Realistic Critical Tree Model

Perfect trees only exist in a perfect world. If all the trees were like the perfect critical tree, we would be achieving a near-perfect parallel efficiency when we have more work than processors. Of course, this is not the case in reality.

We have seen the available parallelism and relative frequency of each of the synchronization points in a perfect critical tree. For the realistic critical tree, the size of the work granules is not uniform, and follows a random distribution. At each synchronization point, one must wait for *all* processors to finish their work. Thus, at each synchronization point, we must compute when each processor finishes its last piece of work. To simplify the computation, we will assume that each processor gets its last piece of work at exactly the same time.

If we have m processors working on different k-ply subtrees, how can we determine the maximum size of the subtrees? A related problem is determining the maximum of m random numbers drawn from a uniform distribution of real numbers from 0 to 1. Fortunately, the latter problem is solved easily by computing the median of the probability distribution.

Let us call the median of the probability distribution x. Thus, the largest random number chosen will be less than the real number x 50% of the time. For the maximal random number to be less than x, all of the random numbers chosen must be less than x. If m independent random numbers between 0 and 1 must be less than x, this implies that $x^m = 0.5$. Solving for x, we get $x = e^{ln(0.5)/m}$.

This result can not only be used to determine a likely estimate of the maximum in the uniform random number case, but for any probability density function. We can determine the point x in the cumulative density function where $\mathrm{cdf}(x) = e^{ln(0.5)/m}$ to determine the median of the maximum of m random variables.

Using the fitted gamma probability density functions from Section 4.3, we can determine the median of the largest of m pieces of work, and use that to uniformly add a penalty at each global synchronization point. For example, if 256 processors attempt to finish a 4-ply search in a bf=38 search, we are looking for the point where $\mathrm{cdf}(x) = 0.99729$. Using the fitted gamma distribution with a mean of 2887 for chess ($\lambda = 7.5718 \times 10^{-5}$, and $r = 0.2186$), this yields a median of 44295 nodes for the largest subtree. Thus, we add a penalty of 44295-2887 = 41408 to the parallel time at each synchronization point where 256 processors are employed. Figure 4.3 shows the parallel efficiencies, once the penalties for randomness in the size of work granules have been taken into account.

Note that the theoretical parallel efficiency is dramatically reduced by using the realistic critical tree instead of the perfect critical tree. For example, if we look at 256 processors and a 10^7 node sequential search, Figure 4.3 shows speedups of 71, 67, and 22 for branching factors of 38, 10 and 3, respectively. The equivalent theoretical speedups from the perfect critical tree are 222, 200 and 91. This leads us to suspect that there are large potential gains from removing or reducing the effect of global

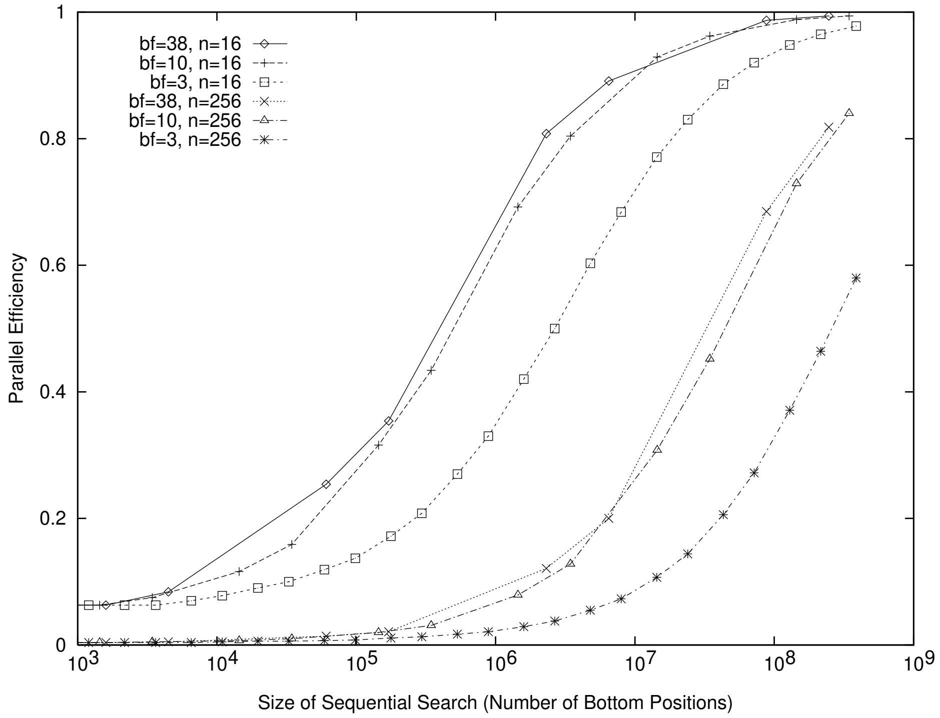


Figure 4.3: Synchronous Model, Efficiency on Realistic Critical Tree

synchronization points in the search.

4.5 Analysis of Asynchronous Game-Tree Search

The asynchronous search algorithm that we will explore will be based on a generalization of Newborn's UIDPABS search [69]. The algorithm will partition the game tree amongst the available processors. The processors can then continually search their own work to greater and greater depths, independently of all other processors.

The algorithm will use a fixed fanout of size m to recursively add more processes to the system until we have one process for each of the n processors in the system. The first process searches the root of the game tree until it has enough sufficiently large pieces of work to hand to m processes. Then, the game tree is subdivided into m independent processes on m separate processors. These m processes can search their work independently of one another to successively greater depths. Each of these m processes can, hierarchically, partition their work to create another group of m

processors. This continues until all n processors have a process allocated to them.

Once a time limit is reached, all processes stop, combine their results and make a decision regarding the best move in the position. The exact mechanism of how the decision is reached is not germane to the analysis; the only important point is that all of the searches have been executed to a common search depth before the time limit.

4.5.1 Perfect Critical Tree Model

In the perfect critical tree, we have many stages before all of the processors are completely busy. At first, only one processor searches the game tree, attempting to generate enough work to keep m processors busy. Let us define this minimum depth as an x-ply search. Clearly, x is dependent on the number of processors that need work, the number of pieces of work each processor should get and the size of each piece of work before it is handed off.

After this point, up to x' ply within the tree, m processors are attempting to generate enough work to keep m^2 processors busy. This continues, recursively, until all of the processors are completely utilized. Let us define the search depth where all processors are working as a v-ply search.

Thus, the time required to search the tree in the asynchronous case will take $\mathrm{PAR}_{\mathrm{asynch}}$ units of time, where:

$$\begin{aligned}
\mathrm{PAR}_{\mathrm{asynch}} &= (\sum_{k=1}^{x}(\text{size of } k\text{-ply search}))/1 \\
&+ (\sum_{k=x+1}^{x'}(\text{size of } k\text{-ply search}))/m \\
&+ (\sum_{k=x'+1}^{x''}(\text{size of } k\text{-ply search}))/m^2 \\
&+ \cdots + (\sum_{k=v+1}^{d}(\text{size of } k\text{-ply search}))/n.
\end{aligned}$$

Using the formula for sequential tree size from 4.4.1, we can sum up each stage of the iteratively-deepened search, to determine $\mathrm{SEQ}_{\mathrm{asynch}}$, which is exactly the same as

$\mathrm{SEQ}_{\mathrm{case1}}$. Once this is determined, we can get the speedup $\mathrm{SPD}_{\mathrm{asynch}}$.

However, there are a number of parameters that must be defined before we can compute the sequential time, the parallel time and the speedup. The first parameter is the size of the fanout m. We want the fanout to be large enough so that the work can be distributed quickly to all processors. However, if the fanout is too large, the single process generating the pieces of work may take a prohibitive amount of time. For the analysis presented here, we will use a fanout of $m = 16$ because of the number of processors chosen.

We must also define how many pieces of work are to be created, and how big each piece of work must be. Although each processor only needs one piece of work, we will wait until each processor can be allocated 16 pieces of work. If the algorithm has a load balancing system, 16 pieces of work will be sufficient to allow the workload to be evenly distributed amongst the processors. More pieces of work per processor could be sent out, but the key to achieving good parallelism is to minimize the number of pieces of work we must wait for. This allows the parallelism to start up quickly.

Each of these pieces of work should have at least 20 nodes underneath the horizon before the piece of work can be given to another processor. This ensures that there is some stability in the minimax value for the piece of work. In the author's experience, 20 nodes is sufficient to guarantee some stability in the minimax value.

Figure 4.4 shows the parallel efficiency of the proposed asynchronous algorithm for the same branching factors and processor configurations used in the analysis of synchronous game-tree search. Using a 10^7 leaf node sequential search and 256 processors, we see that we can achieve approximate speedups of 94, 161 and 133 when the branching factors are 38, 10 and 3, respectively.

We note that there is a much smaller gap between the best and worst performance as we vary the branching factor. This is to be expected, since there is no implicit reliance on the branching factor in the definition of $\mathrm{PAR}_{\mathrm{asynch}}$. In the definition of the theoretical parallel speedup for synchronous search, the branching factor is an important variable in the formula.

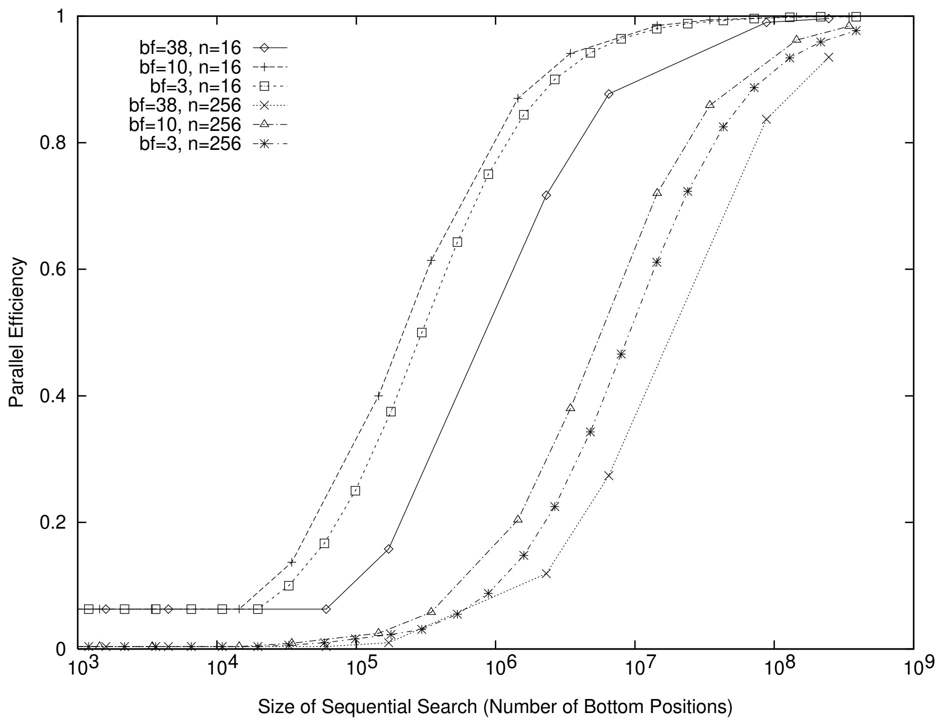


Figure 4.4: Asynchronous Model, Efficiency on Perfect Critical Tree

Another interesting point is that the speedup when the branching factor is 38 is worse than the speedup when the branching factor is 10 or 3. The main reason for this is the size of the steps taken in iterative deepening. By examining Figure 4.4 closely, we note that the parallelism seems to start at a larger sequential search size when the branching factor is 38 than when the branching factor is 10 or 3. It should be noted that a smaller branching factor allows the algorithm to react quickly to the availability of many pieces of work, and thereby achieve greater parallel efficiency.

4.5.2 Realistic Critical Tree Model

For the synchronous search algorithm in Section 4.4.2, we altered the model to introduce a factor to represent the maximum tree size when all processors are working together. We will now proceed to add this factor to the theoretical parallel speedup for the asynchronous algorithm.

In terms of synchronization at the end of the search, we are in a worse situation with asynchronous algorithms. This is because there is no opportunity for the processors to synchronize with one another during the search. Under the asynchronous algorithm, they will always have something to do, but it may not be relevant to a particular search. If a processor starts working on an iteration while other processors lag behind, the search may be wasted. Without load balancing, some processors could be waiting for others on the same iteration for a long time. Thus, we will assume that there is some form of load balancing in place for the asynchronous algorithm. This load-balancing scheme was implicit in the synchronous model given earlier since we stated that the distribution of work was perfect except for the final piece of work.

If the parallelization horizon is at v-ply, we can assume that we must complete a $(d - v)$-ply search underneath that node. The same penalties that we imposed on the synchronous model can be imposed here. However, with an efficient load-balancing scheme in place, all of the processors will be doing useful work until the final iteration; the penalty should not be assigned until the last level. Thus, we will assign the penalty only at the final level of a k-ply iteratively-deepened search. Unlike the case of the synchronous algorithm analyzed earlier, penalties from a completed k-ply search will not affect an iteratively-deepened $(k + 1)$-ply search.

Figure 4.5 shows the parallel efficiency of the asynchronous algorithm with the penalty for synchronization inserted. There is little difference between the perfect and realistic critical trees for the asynchronous algorithm. This is expected since there is only one synchronization point (at the end of the search) to worry about. For a 10^7 node search using 256 processors, we note that the parallel speedup drops from 94 to 92 when the branching factor equals 38. The speedups also drop from 161 to 145 when the branching factor is 10, and from 133 to 121 when the branching factor is 3.

It is important to observe that the curves taper off before reaching a perfect efficiency. The synchronization penalty takes place on a $(d - v)$-ply tree for the asynchronous algorithm. As the search depth d increases, the size of the tree that we apply the penalty to increases at the same relative rate. Thus, the penalty becomes a fixed constant of the parallel efficiency. Note that in the analysis of the synchronous

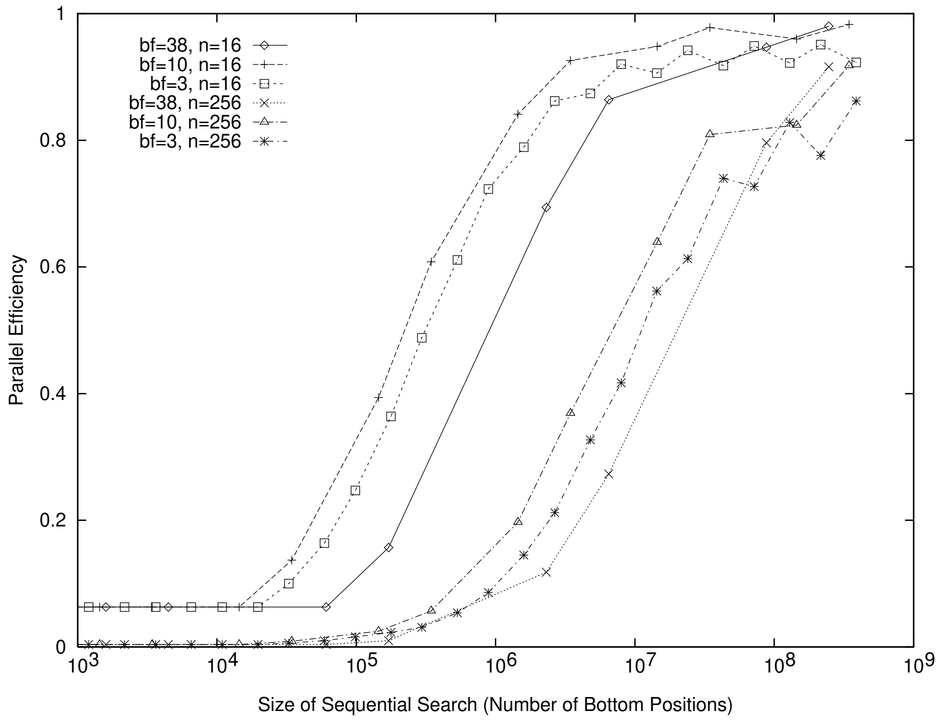


Figure 4.5: Asynchronous Model, Efficiency on Realistic Critical Tree

algorithm, the work granule size was fixed. Thus, the penalty for a k-ply work granule, where k does not increase with d, eventually becomes negligible in comparison to the overall tree size.

The curves also have a sawtooth pattern as they approach the limit of their efficiency in the asynchronous algorithm. This is because the number of leaves examined in the last piece of work only increases every two ply, while the tree size increases at every ply. Thus, the relative penalty decreases and increases, depending on the parity of the depth of search.

There are other overheads that should be examined in the asynchronous algorithm. For example, the implicit assumption is that we know *a priori* the window that we must search to determine the minimax value at any given node. In practice, we do not know the search window. Furthermore, in cases where we guess incorrectly, the work must be re-searched with the correct window (just in case it is the final iteration). It seems that we must use a process to keep track of the poor window choices and to alert the workers of their mistakes.

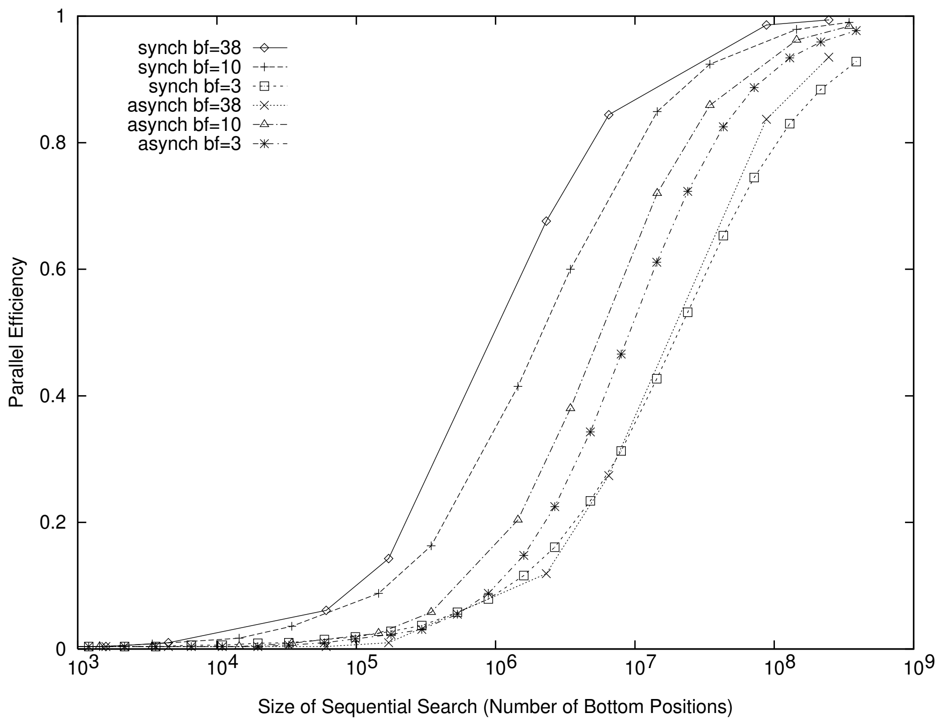


Figure 4.6: Comparing Models on Perfect Critical Tree, n=256

This means that the processor must receive score information from the searchers on a timely basis. Furthermore, the system is now centralized, causing potential communication bottlenecks. The theoretical model does not consider these things but they are important in practice, as we shall see in subsequent chapters.

4.6 Summary

Let us now compare the synchronous model to the asynchronous model on both the perfect and realistic critical trees. The synchronous model outperforms the asynchronous model on the perfect critical tree for the simulated chess and Othello game trees examined (Figure 4.6). The simulation of checkers search trees show a slight improvement for asynchronous search. However, the benefits of asynchronous game-tree search are relatively clear once the realistic critical tree model is examined (Figure 4.7), especially for trees similar to those found in the game of checkers. This result is not surprising; global synchronization can impose a huge cost on game-tree search,

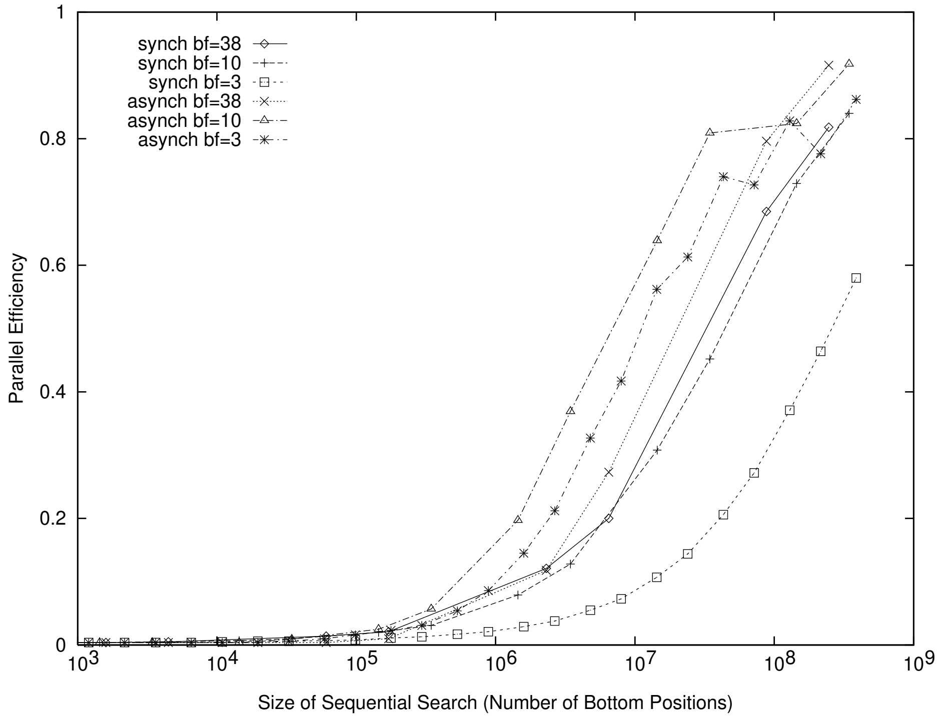


Figure 4.7: Comparing Models on Realistic Critical Tree, n=256

and should be minimized if possible.

However, the synchronous game-tree search algorithm will eventually catch up and overtake the proposed asynchronous algorithm on the realistic critical tree, given a sufficiently long search. The global synchronization is relatively large at the beginning of the synchronous game-tree search algorithm, but decreases as the search depth increases. Whether the synchronous or asynchronous search algorithm performs better will depend on the time limit. For the relatively short time constraints under which moves must be selected in chess, Othello and checkers, the asynchronous search algorithm seems to be favoured.

It is very difficult to determine whether synchronous or asynchronous game-tree search is better suited for a given real-world application. Important considerations have not been addressed by this theoretical framework that impact on one or both of the parallel search implementations. The requirements for sharing transposition table entries, centralized versus distributed control, message latency and bandwidth, the ability to guess the correct $\alpha\beta$ search window and the time limit for making the move decision are all important factors in determining the actual performance of the algorithms.

These benefits and drawbacks should not only be examined in a theoretical manner, but also on real game trees using real game-playing programs. Before we can study asynchronous game-tree search on real game-playing programs, we must develop a working implementation that can be tested. The implementation of APHID, a parallel game-tree search library using the asynchronous algorithm described in Section 4.5, will be described in the next chapter.

Chapter 5

The APHID Algorithm

5.1 Introduction

Young Brothers Wait and other synchronous parallel search algorithms suffer from four serious problems. First, the numerous global synchronization points along the principal variation result in idle time. This suggests that a new algorithm must strive to reduce or eliminate global synchronization points altogether. Second, the chaotic nature of a work-stealing scheduler requires algorithms such as YBWC* and Jamboree to use a shared transposition table and/or recursive iterative deepening to achieve a good move ordering and reasonable performance. 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 cut-off, 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. Fourth, parallel game-tree search usually requires a significant software engineering effort to install a parallel algorithm into an existing application. We must worry about getting the parallel algorithm correct, as well as sorting out all of the foreseen and unforeseen interactions between the algorithm and the application.

This chapter introduces the Asynchronous Parallel Hierarchical Iterative Deepening (APHID[1]) game-tree search algorithm. APHID represents a departure from the plethora of synchronous algorithms described in Chapter 3 and has been designed to address the aforementioned problems. First, the algorithm is asynchronous in nature; it removes all global synchronization points from the $\alpha\beta$ search and from iterative deepening. Second, the algorithm does not require a shared transposition table for move ordering information, although one can be used if duplicate detection is important in the underlying application. Third, parallelism is only applied at nodes that have a high probability of needing parallelism, and this decision is based on the best information available at a given point in time. Finally, APHID is designed to easily fit into an existing sequential $\alpha\beta$-based search algorithm. APHID has been implemented as a game-independent library of routines. This library, combined with application-dependent routines that the programmer supplies, allows 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 and debug, the game-independent library allows programmers to integrate parallelism into their $\alpha\beta$-based application with only a few hours of work.

APHID subdivides the tree into many distinct pieces which each process can search independently of all other processes. Figure 5.1 gives an example of how the game tree would be divided into distinct portions by the APHID algorithm. The figure shows us a single-level master/slave hierarchy. A single master process controls the top d' ply of the game tree, including the root. The leaves of the master's d'-ply tree are the pieces of work that each slave process examines. Each of the k slave processes get a portion of the pieces of work on the horizon. By only analyzing the pieces of work allocated to them by the master, the k processes subdivide the remainder of the game tree, as illustrated in the figure.

In APHID, the master process makes repeated searches or *passes* over its part of the game tree. The top part of the tree is small and is quickly searched by the master

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

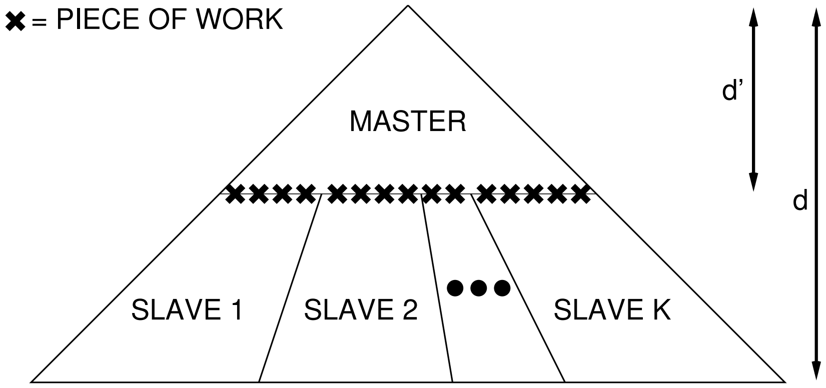


Figure 5.1: APHID Partitioning Game Tree Amongst Processes

process. The master tells the slaves about new pieces of work that it uncovers, and can delete pieces of work that are no longer relevant. The master may move pieces of work from one slave to another for load-balancing purposes and subdivide a piece of work into many smaller pieces of work.

At the horizon between the master and the slave processes in APHID (as illustrated in Figure 5.1), each x along the horizon represents a piece of work that the master has given to a specific slave. Each slave is given a number of pieces of work. The master informs each slave of the relative importance of each piece of work.

Each slave is responsible for searching all of its allocated pieces of work to the same depth, plus any extensions that the master requires. Instead of waiting for the master to inform the slave to what depth each piece of work should be searched, the slave uses iterative deepening to search each piece of work deeper and deeper until the master tells the slave to stop. Thus, the slave is always searching one of its own pieces of work, independently of the other processes. The slave informs the master of the minimax value of the searches as soon as they are determined, and periodically listens for important update messages from the master.

By partitioning the game tree in this manner, APHID's performance does not rely on the implementation of a global shared memory or a fast interconnection network between the processes. This makes the APHID algorithm suitable for loosely-coupled architectures (such as a network of workstations), as well as tightly-coupled architectures.

The APHID algorithm has been implemented as an application-independent library of routines that can be easily ported to any $\alpha\beta$-based application. The library has been designed to provide minimal interference with an existing sequential search algorithm. APHID uses call-back functions and application-dependent variables provided by the programmer to access vital information from the sequential search algorithm.

The APHID library is also portable across various parallel and distributed architectures. APHID is coded in the C programming language and uses PVM [35], a portable message-passing interface, to allow a large selection of parallel hardware to use the APHID library without the user having to modify the source code.

In this chapter, we will discuss the mechanics of the APHID algorithm and the application-level interface. Section 5.2 discusses how the APHID algorithm performs the asynchronous search. Section 5.3 gives an in-depth example of how APHID operates during a parallel search. Section 5.4 illustrates the external interface of APHID, and how the algorithm integrates with an existing sequential $\alpha\beta$-based algorithm.

5.2 Internal Mechanics of the APHID Algorithm

As mentioned in the introduction, APHID can be viewed as an asynchronous master/slave program. A master process is responsible for organizing the search on a number of slave processes, but never globally synchronizes the slaves during the search itself. For a depth d search, the master is responsible for the top d' ply of the tree. The remaining $(d - d')$-ply are searched by the slaves.

Figure 5.2 illustrates graphically where work is allocated over the course of a typical APHID and YBWC* search. Each location marked with an `x` shows where the parallelism typically takes place. Although more parallelism could be generated in YBWC*, we should note that each `x` along the left side of the YBWC* tree represents a global synchronization point.

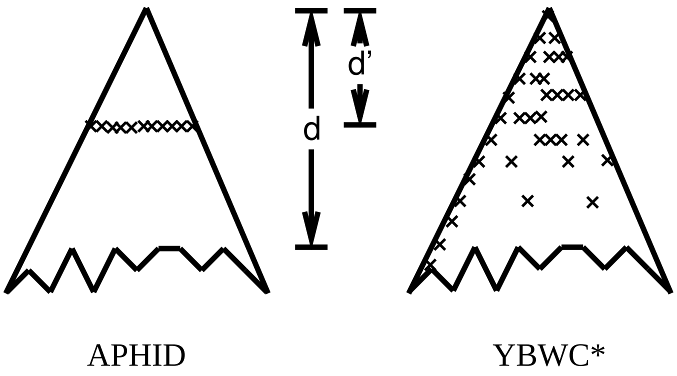


Figure 5.2: Location of Parallelism in Typical APHID and YBWC* Search

We will start by describing the processes and illustrating the interactions between a master and its slave processes (Sections 5.2.1 to 5.2.3). Further sections describe how the hierarchy of processes is constructed (Section 5.2.4) and how the work load is balanced amongst the processes (Section 5.2.5). Finally, we discuss the implementation of a novel distributed transposition table mechanism that has been added to the APHID library of routines (Section 5.2.6).

5.2.1 Operation of the Master in APHID

In general, when a master process is informed that it must execute a d-ply search of the game tree, it continually repeats the following five steps:

1. Execute a quick search (or *pass*) of the d'-ply tree using $\alpha\beta$. The search uses exact or guessed evaluations of the remaining $(d - d')$-ply of the tree at the leaves of the d'-ply tree.

2. If we have determined an accurate d-ply minimax value during the last pass, exit the loop.

3. Compute any required changes to the work lists.

4. Inform slaves of changes to their work lists.

5. Wait for new information from a slave process before going back to the first step.

The first question that is raised is why does the master continually search the d'-ply game tree? The master continually obtains new information on the leaves of the d'-ply tree from the slaves. There are two alternatives for updating the d'-ply tree to determine a minimax value. The first is to maintain the top part of the game tree in memory and update the tree in place as results are returned. This method was suggested for Hsu's queued processor array model [41]. The second is to repeatedly traverse the d'-ply tree until an accurate minimax value has been determined on one of the searches of the tree. Hsu's choice was based on the relatively large tree that must be maintained within the host workstation. The parallelization horizon d' is usually small in APHID, and we cannot assume that a game tree representation is given to us by the search algorithm. Thus, the second approach is taken in APHID.

Now that we understand why we search the tree repeatedly, we will deal with the details of how the master conducts a search of the d'-ply game tree. When the master reaches a leaf node at the artificial horizon of the d'-ply tree, the master must decide whether to give this work to a slave or search the node itself. We define the variable g to represent the minimum number of ply required for a slave process to be allowed to search the node. This is often called the *minimum granularity* for a piece of work. Thus, a leaf node that needs to be searched to g ply or greater is transmitted to a slave process. Leaf nodes of the d'-ply tree that require less than g ply of search are carried out by the master, since they are deemed to be too small to be handed to another process.

Assuming that the leaf has been given to a slave in a previous or the current pass of the tree, we must determine a minimax value for the leaf so that the master may complete the $\alpha\beta$ search. If the slave has already given the master a $(d-d')$-ply search result, that value is used[2]. If the $(d-d')$-ply result is unavailable because the slave has

[2] In the implementations presented here, we do not use deeper ply values even if they are available. This will be discussed in subsequent chapters.

not reached that depth of search, or the $(d-d')$-ply result has returned a bound that yields insufficient information on the minimax value with respect to the $\alpha\beta$ search window, then the algorithm guesses at the minimax value. Any node where we are forced to guess at the minimax value is marked as an *uncertain* node.

As minimax values get backed up the tree during the search, the master maintains a count of how many uncertain nodes have been visited in a pass of the d'-ply tree. As long as the score at any of the leaves that are currently within the master's d'-ply tree are uncertain, we have not determined the minimax value at the root with certainty, and the master must execute another pass of the d'-ply tree. Once the master has a reliable value for all the leaves in its d'-ply tree, the minimax value of the complete d-ply tree is known, and we exit the loop. Generally, the application would proceed to the next iteration by incrementing d and asking the master to search the tree again.

Earlier, we avoided defining exactly how the algorithm generates a guessed minimax value. The reason for this is that generating a guessed minimax value at a leaf node is a very complex issue, and will be covered in-depth in the following paragraphs.

The guessed score algorithm attempts to find a result which completely determines the minimax value with respect to the search window. For example, if the search window is (0,2) and the guessed value is ≤ 5, the guessed value does not tell us anything relevant to the $\alpha\beta$ algorithm. The search window (0,2) tells us that we want to know whether the minimax value is ≤ 0, equal to 1 or ≥ 2. The algorithm checks successively shallower depths of search returned by the slave until we get a result that is $\leq \alpha$, $\geq \beta$, or an exact minimax value. This is guaranteed to happen, since the master stores an exact evaluation of a leaf node when the leaf is first generated and given to a slave.

The first definition of the best available result used in the APHID algorithm was the largest search result that had the same parity of depth. This definition was chosen since some applications exhibit an *odd/even effect.* [3] Thus, APHID only used

[3] A program that exhibits an odd/even effect will have oscillating minimax values as we use iterative deepening to search the tree to deeper levels. However, when we look at only odd search depths or even search depths, the minimax values are stable.

even search depths for guessing an even search depth, and odd search depths for guessing the value of an odd search depth. This definition was originally believed to be sufficient to cover the majority of applications, and was present in earlier tests of the APHID library [15, 16].

However, as numerous applications were tested, some deficiencies in this algorithm became clear. The first is that some programs do not exhibit an odd/even effect in tactical positions; using the result of the same parity can dramatically alter the master's d'-ply tree. For example, if a tactical combination is determined at $(d-1)$ ply that changes the principal variation, using guessed values from $(d-2)$ ply will cause the tree to revert to the old principal variation and the program must re-discover the combination.

Another serious deficiency occurs when the minimax value of the principal variation drops, but will still be the best move for a depth d search. If the $(d-2)$-ply minimax value is 5, and the d-ply value for the principal variation drops to 3, the only information that we have for the other variations is that the minimax value is ≤ 5 at $(d-2)$ ply. The algorithm believes that the best move is worse than every other move choice. The APHID algorithm attempts to prove that each of the alternative move choices is better, and fails on each move. Every change in the best move choice at the root of the game tree causes the master's tree to change dramatically, and this yields severe performance degradation.

In light of these concerns, a new definition of best available result was required. The current definition can use any ply value, not just the deepest available result of the same parity of depth. However, we should not use the result returned by the slave directly. We must scale the slave's minimax value, based on the difference between the minimax value of the full search that it was generated for and the hypothetical minimax value of the current search. For example, if the current search has a hypothetical minimax value of 5, and we wish to use a result of ≤ 6 from a search that had a minimax value of 7 at the root of the game tree, we would add the difference of the minimax values (5-7) to the result. Thus, the value changes from ≤ 6 for the

Ply Of Slave Result	0	1	2	3	4	5	6
Result Returned By Slave	$= 20$	≤ -8	≥ 20	= -6	≥ 2	≤ 15	?
Minimax Value of Full Search	3	5	7	-1	-3	-7	?
Scaled Result Returned By Slave	$= 15$	≤ -15	≥ 11	= -7	≥ 3	≤ 20	?

Table 5.1: Example of Information Used to Determine Guessed Minimax Value

earlier search to ≤ 4 for use as a guessed minimax value in the current search.

This scaling algorithm is necessary to guarantee that a node affects the search tree in the same way as it previously affected the search tree. If the old search result (≤ 6) was sufficient to cause the tree to be pruned in the earlier search, the guessed value (≤ 4) will be sufficient to prune the search tree in the current search. Thus, the master's search tree does not change until new information is given to the master.

Let us follow a complete example of how the guessed score algorithm would behave at a sample leaf node of the master's d'-ply search tree. Assume that the master has reached a leaf node with the search window (-10, 8), and we want the minimax value of a 6-ply search. From the search window and the definition of $\alpha\beta$, we want to determine a minimax value that is $\leq -10, \geq 8$, or an exact value in between those two bounds. For the purposes of the algorithm, assume that the hypothetical minimax value is -2. In Table 5.1, the first row of numbers represent the ply that the leaf node has been searched to on the slave. The second row represents the values returned by the slave for each search depth. The third row represents the minimax value of the complete d-ply search that this search depth is associated with. If we do not use search extensions or reductions, d is equal to d' plus the search depth given in the first row. The fourth and final row represents the values returned by the slave, scaled by the difference between the hypothetical minimax value, -2, and the minimax value of the earlier search (as given in the third row). Unknown values that have not been sent to the master are represented as a question mark (?).

The algorithm starts by looking at the scaled 6-ply result, and determines that the value is not available. Thus, the algorithm proceeds to the 5-ply scaled result. This result of ≤ 20 is not of interest, since it doesn't tell us where the minimax value lies with relation to the search window (-10, 8). We proceed to the 4-ply result, where we have a scaled value of ≥ 3. Although this result is interesting, it does not tell us whether the minimax value is ≥ 8 or, if the value is in between 3 and 7, what the exact minimax value is. Thus, this value is also insufficient to terminate the search. The algorithms proceeds to the 3-ply result, with a scaled minimax value of -7. Since this result is exact, this scaled value is used by the master as the evaluation of the leaf node, the node is marked as uncertain, and the master's search is allowed to continue.

It is important to note that the 1-ply and 2-ply result would also be sufficient to terminate the search, since the scaled minimax results are ≤ -10 and ≥ 8, respectively. Also note that the 0-ply result is sufficient to terminate the search since the scaled result is always an exact value and not a bound on the minimax value.

When we described the definition of uncertain nodes, we alluded to the fact that the $(d - d')$-ply search result may be insufficient for the master to determine the minimax value. We have seen examples where a scaled minimax value from an earlier search may not be relevant to the search window. This can also happen with the $(d - d')$-ply search result. If we do not remedy the situation by determining a useful $(d-d')$-ply search result, the node will remain uncertain and the d-ply search will never terminate. Thus, the slave responsible for searching the node must be informed that it needs to execute a "bad bound" search. When a bad bound search is generated, the depth of search and the search window that must be searched are communicated to the slave process. Clearly, any nodes where we are waiting for bad bound information to be updated by the slave are considered as uncertain by the master, since we must use the guessed score algorithm to determine a likely minimax value. In a future pass of the game tree, the slave will return updated minimax value information that is consistent with both the original information and the search window requested[4].

The master obtains information as the search of the d'-ply tree is executed. The moves that are played to reach the leaf node are obtained as we descend the search

[4]It may happen that the original search and the "bad bound" search are inconsistent with one another, through the use of search extensions that may or may not be triggered based on the search window used. In this case, the search explicitly requested by the master overrides the information that had been previously stored.

tree. In some positions, we may not want to search a given piece of work to $(d - d')$ ply as we did in the sequential algorithm. Search extensions and reductions may be applied to the node in the master's d'-ply tree. The adjustment to the required search depth is determined before we attempt to evaluate the leaf of the d'-ply search tree, so that it may be transmitted to the appropriate slave process.

Additional information is also acquired as we back up the master's search tree. We can determine upper and lower bounds on every node within the master, including the root of the game tree. Furthermore, we have a hypothetical principal variation from the last pass of the master's tree that can be used at any time during the search.

The number of uncertain and reliable leaf evaluations is also available to the application. We can use this additional information for a sophisticated termination algorithm. For example, if there is only one uncertain node left in an iteration, and the application has reached the time limit, the time limit might be extended for a few seconds to allow the search of the last piece of work to complete.

Before deciding to start searching in parallel, APHID also obtains information on the frequency and location at which nodes are generated during the search of the game tree. These statistics are stored in a table that is organized by ply number. APHID uses advice from the user along with these statistics to determine when to start parallelism.

To be precise, we attempt to generate hypothetical values for d' and g, based on their definitions. The user, as we shall see in Section 5.4, defines the minimum number of pieces of work per slave process, and the minimum average size of each piece of work. According to the APHID algorithm, the definition of the parallelization horizon d' is the smallest ply within the game tree where we routinely search k nodes or more. k is the user-set minimum number of pieces of work per slave process multiplied by the number of slave processes. Similarly, the minimum granularity g is defined as the difference between d' and the smallest ply within the game tree where we routinely search l nodes or more. l is the user-defined minimum size of each piece of work multiplied by k, as defined earlier.

If hypothetical values of d' and g can be satisfactorily determined by the statistics, the search starts in parallel with those parameters. Otherwise, the full d-ply search is done sequentially by the master. It is important to note that the size of the tree searched in each iteration increases. Thus, the search will eventually grow large enough to define values for d' and g.

As a final note, APHID solves one of the problems that synchronous algorithms have with respect to initializing parallelism incorrectly at a potential type-2 node. By using the guessed scores when accurate information is not available, the APHID algorithm automatically determines if a subsequent child is likely to generate a cut-off at a failed type-2 node. If it seems likely that a child will generate a cut-off based on guessed values, the children of the failed type-2 node are evaluated sequentially. If it seems unlikely that the node will be pruned due to low minimax values, the search would continue for a promising node at that branch in parallel. This is all handled automatically by the $\alpha\beta$ routine combined with the estimated and/or accurate minimax values. The handling of a hypothesized type-2 node is stronger than the equivalent scenario in the YBWC algorithm, which ignores previous score information available for some branches of the failed type-2 node. In the full version of YBWC*, application-dependent information is used to do what APHID handles automatically with the $\alpha\beta$ algorithm.

5.2.2 The APHID Table

If a leaf node is visited by the master for the first time, it is allocated to a slave process. This information is recorded in a table, the *APHID table*, that is shared by all processes. Figure 5.3 shows an example of how the APHID table would be organized at a given point in time.

The table is replicated on the master and slave processes. However, each slave process only knows of the entries relevant to it within the table. For example, in Figure 5.3, the first slave only knows about the entries for 1, 4 and 7, and does not know of the existence of the other entries. The master, which is responsible for

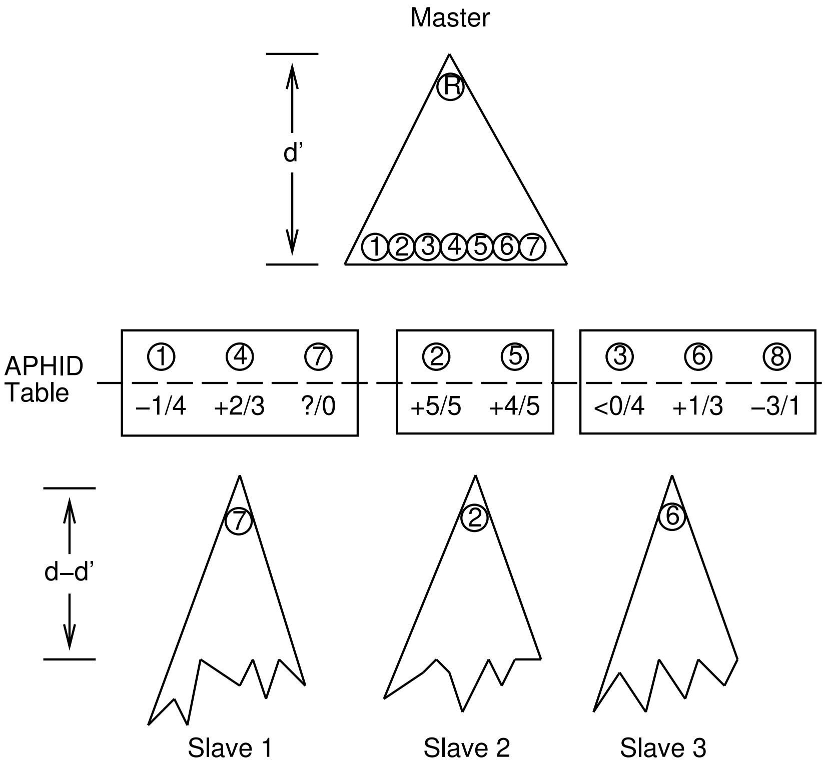


Figure 5.3: A Snapshot of APHID Search in Operation

distributing the work to all of its slaves, has copies of every table entry. Thus, there are two copies of a given entry in all of the processes' APHID tables.

The master and slave only read their local copies of the information; there are no explicit messages sent between them asking for information. The entries in the APHID table are 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 master process to update its replicated copy of the table entry.

The master's half of the table is illustrated above the dashed line in Figure 5.3. For each leaf that has been visited by the master, there is an entry in the APHID table. Information maintained on the leaves includes the moves required to generate the leaf positions from the root `R`, the approximate location of the leaf in the tree (which is used by the slave to prioritize work), whether this leaf was visited 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 roughly the same number of leaves have been allocated to each slave. Note that there is an additional leaf, 8, that is not represented in the master's d'-ply search tree. This leaf node has been visited on a previous pass of the d'-ply search tree, and was not visited on the latest pass. However, the information that the slave has generated may be needed in a later pass of the tree and is not deleted by the master. Leaves are initially allocated to the slaves in a round-robin manner, and may move due to load balancing (as described in Section 5.2.5). Although there may be better methods of allocating leaves, it has been found that this is a reasonable method of initially balancing the load on a small number of processes.

The slave's part of the table, illustrated by the area below the dashed line, contains information on the minimax value at various depths of search. The best information (with respect to search depth) and the ply to which the leaf was examined is given underneath each leaf node. 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 both an upper bound and a lower bound on the minimax value for each ply of search depth. Clearly, the value is known to be exact when the upper and lower bounds are the same. In Figure 5.3, we note that only a single bound is shown to make the figure less complicated.

Early implementations allocated space for a large number of possible entries in the APHID table before the beginning of each search. These implementations had difficulties because of the size of the data structure; the APHID table took about half of the 8 MB of RAM available on some computers. Thus, the current version of the APHID algorithm features an APHID table that is grown dynamically by the master and slaves. As more entries are required on a process, more locations to store the entries are dynamically created. This allows the applications being parallelized with APHID to use the majority of the memory available on the system, if needed.

5.2.3 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 three steps until the master tells it that the search is complete:

1. Look in its portion of its local copy of the APHID table, and find the best node to search.

2. Execute the search.

3. Report the result back to the master (fetching any update to its APHID table in return).

The first work selection criterion is based on the depth to which the slave has already searched a node. Nodes with shallower search depths are preferred over those with deeper depths, because they represent more work to be done. As we can see for Slave 1 in Figure 5.3, leaves 1, 4 and 7 have been searched to 4, 3 and 0 ply, respectively. Thus, Slave 1 is attempting to search leaf 7 to 1 ply, and will continue to search leaf 7 up to 3 ply using iterative deepening, if no new work arrives from the master.

Although we have not shown the information in Figure 5.3, each of these pieces of work need not necessarily be searched to the same depth. If the master has told the slave that there is a depth adjustment of 2 for a given piece of work (for example, a search extension of 2 ply), the piece of work must be searched to depth $(k+2)$ while the other pieces of work are searched to depth k. Thus, the depth adjustment must be subtracted from the depth already searched in the algorithm given above. This allows work that must be searched to varying depths to have the same priority.

The second criterion is the location of the node within the master's game-tree. This criterion is necessary since it is usually beneficial to generate the results in a left-to-right order for the master. Children of nodes are usually considered in a best-to-worst ordering, implying that the left-most branches at a node have a higher probability of being useful than the right-most ones. For Slave 2 in Figure 5.3, leaves 2 and 5 have both been searched to 5 ply, but leaf 2 is being searched in the slave to 6 ply since it is further left in the tree than leaf 5.

Unfortunately, maintaining a complete ordering of each leaf in the d'-ply tree can be expensive. Thus, APHID uses a priority scheme to give an approximation of this second criterion. For each type-1 node that is traversed as a move path is generated, 4 is added to the priority. Since the root is a type-1 node, all nodes that are visited on a pass have a minimum priority of 4. If the node is judged to be part of the critical tree, two is added to the priority. This favours critical work over speculative work in cases where the guessed minimax values are incorrect. The final adjustment is an addition of 1 if we are evaluating one of the first few uncertain nodes. This final adjustment ensures that the search proceeds in a roughly left-to-right manner when the other priority determinants are equivalent. If a node is not touched on a pass of the master's tree, it is given a priority of zero.

A node that has a priority of zero will not be selected for further search by a slave. For Slave 3, we notice that Leaf 8 would be searched if it had been visited by the master on the latest pass. Leaf 8 is ignored by the scheduling algorithm because it is not currently part of the master's tree.

Each process may have many pieces of work to examine each time it decides to choose a node. In earlier versions of APHID, the algorithm would search through every entry for the best node to evaluate. This proved to be costly, since 99% of the entries were empty in a typical parallel search.

APHID now maintains a doubly-linked list of priority buckets for each depth of required search. A bucket is designed to hold a fixed number of locations within the APHID table that correspond to the same priority. Each bucket contains a pointer to the previous and next bucket in the list, the priority and the adjusted depth that each entry in the bucket has been solved to. For each depth of search, the buckets are kept in order of decreasing priority, making it easy for the algorithm to find a bucket and, thus, an entry with the highest priority and/or the lowest search depth.

The slave can determine to what level each node must be searched. If there are nodes that must be searched for the current iteration of the master, the node with the highest priority is always scheduled until it has been searched to the requisite depth. When all nodes at a slave have been searched to the required depth, the nodes at the lowest search depth have their search extended, with priority values as a secondary consideration.

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 priority value, and the hypothetical value of the root of the master's tree. Although the width of the search window is application-dependent, we normally want to center the window around this hypothesized root value. If we are certain of the minimax value of the leftmost child at a type-1 node, we would typically use a null window to search the alternative branches. However, if we are uncertain of the minimax value, the window should be marginally larger to reflect this uncertainty. This uncertainty should also be taken into account when we start speculative searches for future iterations, since we have no information on the minimax value at the next ply. In each of the programs tested in Chapter 6, the window selection algorithm described above has been used. The algorithm has been customized for each application, based on the scale and variance of the evaluation function.

When reporting the search result back to the master, a count of the number of nodes is returned to the master along with the minimax value. These node counts are used in the load-balancing algorithm, as we shall see in Section 5.2.5.

The cost of creating and sending a message to the master for each piece of work may be prohibitive if the interconnection network is slow and the slave process generates results quickly. Furthermore, at the beginning of the parallel search, there are a large number of results to be sent to the master. If each result is sent in a separate message, the master will become congested with message traffic. For these reasons, the score updates sent from the slave to the master are buffered before the results are sent. APHID does not allow score updates to be sent unless the slave has searched at least m nodes since the last score update was sent. m is based on the number of processors in the system, and the number of nodes per second that the application typically searches.

There are three types of update messages that a slave receives from the master: a new piece of work has been given to a slave process, the location of a leaf node within the master's tree has changed (changing the secondary work scheduling criterion), and notification of a "bad bound" on a node. The bad bound message alerts the slave that a position's search information is insufficient from the master's point of view. In this case, the slave must re-search the node with the ply requested and the $\alpha\beta$ search window selected by the master.

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, the slave starts re-searching its work speculatively in anticipation of the next iteration (depth $d + 1$). The slave routinely checks the communication channel for messages from the master as it searches a piece of work, since the APHID algorithm must be able to respond quickly to "terminate search" messages from the master. If the slave receives a new piece of work to do at $(d - d')$ ply or less, the speculative search is immediately aborted and control is returned to the slave's scheduling algorithm. The scheduling algorithm will force the required work to be executed before the speculative work can continue.

Depending on how the score updates are buffered, there may be a significant delay in the master receiving the final score update for the current search depth. Thus, we force the master to recognize that the slave process has finished its work for the current search by flushing the score update buffer as the last piece of work for a d-ply search is completed.

At some points during the search, a slave process will have no work to examine. This occurs routinely at the start of the search when the master is searching the tree sequentially. If a slave process has no work to execute, it iteratively searches the root of the game tree while waiting for a piece of work to examine. This root-node search is treated as a speculative search, and is useful for seeding the move-ordering heuristics with information before the search is started in parallel. However, this root-node search may not always be beneficial if information is shared amongst processes, as we shall see in Chapter 6.

5.2.4 Hierarchies

Using a single master and many slaves will eventually cause a communication bottleneck at the master. Although we have limited our previous discussion to a single master/multiple slave relationship, APHID allows the implementation of a hierarchical structure within the processes. A mid-level process can behave as a slave for its parent, and a master for the processes underneath it. This will spread out the communication congestion at the pinnacle of the process hierarchy.

In the current implementation of APHID, this hierarchy is determined by the user before the program is started. This allows APHID to be responsible for creating the processes on the requested computers, and starting up the interprocess communication package. Over the course of a run of the application, the hierarchy is static and cannot be changed. There are many schemes in the literature for making dynamic process hierarchies, as we have seen in Chapter 3. However, the static hierarchies are sufficient to alleviate the communication bottlenecks, assuming the resources being used remain constant.

5.2.5 Load Balancing

Although the master attempts to give an equal amount of work to each slave in APHID, neither the master nor the slave can predict the amount of effort required to complete a $(d - d')$-ply search for a given piece of work. Thus, load imbalances can occur based on the allocation of work to slaves.

As part of a pass of the d'-ply tree, the master computes how many uncertain nodes it is waiting for from each slave. The master can move leaves of the d'-ply tree from an *overworked slave* (a slave with a large number of uncertain nodes) to an *underworked slave* (a slave with no uncertain nodes). This yields a tradeoff between faster convergence for a given ply search of the tree and additional search overhead, since the previous searches for the piece of work to be moved must be re-searched on another processor.

The load-balancing algorithm always attempts to strip pieces of work away from the most overworked slave at the current point in time. The algorithm prefers to take pieces of work that are small, since they lead to smaller search overheads. To prevent the piece of work that the slave is working on from being taken by the master, the first uncertain node encountered on each slave during a pass cannot be considered for load-balancing purposes. Another stipulation is that the same piece of work cannot be moved twice in a row; this prevents a very small piece of work from being passed from process to process.

Another cause of a load imbalance is a piece of work that is much larger than the other pieces of work. For example, the search tree for the node along the principal variation is generally much larger than the last subtrees examined during a sequential search. When we have such a large piece of work, we would like multiple processes to participate in generating the minimax value. Thus, we need a mechanism that breaks a large piece of work into a number of smaller work pieces that can be distributed (via the load-balancing algorithm) to other processes.

One method of accomplishing this is moving the master's parallelization horizon deeper within the tree for a large piece of work. This allows the master to subdivide a single piece of work into many smaller pieces of work. It could be said that the large piece of work is *exempted* from the parallelization horizon at d'-ply. In Figure 5.4, we see an example of how the horizon can change when exemptions are used. Note that new pieces of work created from exemptions can also be exempted.

The master within APHID is responsible for determining the pieces of work to be exempted, since the master can calculate how much effort has been devoted to each piece of work. In between passes of the tree or in between iterations, the master determines the largest pieces of work that have been explored for the last few itera-

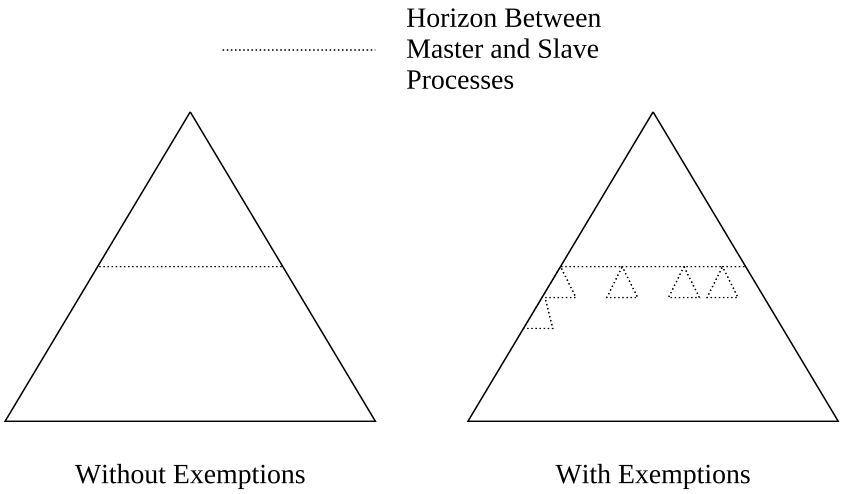


Figure 5.4: An Example of APHID's Horizon, With and Without Exemptions

tions, along with the average size of each piece of work. If the size of the largest piece of work is v times the size of the average piece of work, the largest piece of work is exempted in future searches. In the current implementation, v is a parameter that can be found in the APHID algorithm, and modified to suit the application.

The nature of the $\alpha\beta$ algorithm does not guarantee that many more work granules will be created if we only extend the horizon by a single ply. For example, if the node to be exempted is a type-2 node, the search will likely generate a single type-3 node. When APHID exempts a large piece of work, APHID always extends the horizon by 2 ply to guarantee that the work will be split into multiple pieces.

5.2.6 Distributed Transposition Tables

Insofar as possible, a master in APHID endeavours to keep a piece of work on the slave process that it has been assigned to. This allows all of the move ordering information to be stored locally. In some domains, such as checkers and chess, the duplicate detection feature of the transposition table is as important as the move ordering information stored in the transposition table. For these domains, sharing transposition table information is vital to achieving high parallel efficiency. Therefore, a distributed transposition table has been integrated into the APHID library as a supplement to the asynchronous algorithm.

There are many distributed transposition table algorithms in the literature. As in other parts of APHID, the goal is to achieve the sharing of transposition table entries with minimal use of communication and without requiring the use of shared memory. Thus, a general-purpose scheme cannot generate a message per node examined; this would clog the interconnection network between the machines participating in the search. A depth-limited scheme that only inquires about nodes a fixed number of ply away from the root shows more promise, but the majority of the benefits are already encapsulated within the master's d'-ply search tree. Thus, a new algorithm is needed for sharing transposition table entries over an interconnection network.

In the APHID model, we have only described communication between a master process and its slaves; slave processes cannot communicate directly with one another. However, slave processes that have the same master are peer processes. For the purposes of sharing transposition table results, they should be allowed to communicate with one another directly. We define any pair of slaves that have the same master as *peer slave processes*.

When APHID creates all of the processes, we keep track of the next peer slave process that is spawned, and allow one-way communication from a slave process to the subsequent peer slave process. We also allow the last peer slave process in the process list to communicate with the first, creating a complete cycle.

Instead of transmitting requests for specific nodes to another process, we accumulate information within a small transposition table which is called the *shadow transposition table*. The existence of APHID's shadow transposition table is hidden from the application. Over the course of the search, APHID attempts to store the transposition table entries that are most likely to be useful in the shadow transposition table. In general, entries that are closer to the root of the game tree are more useful. Thus, the shadow transposition table entries are selected based on search depth.

As a slave communicates search results back to its master, the distributed transposition table algorithm is permitted to send its own shadow transposition table to the next peer slave process. After the message is sent, the shadow transposition table is cleared and the slave begins to fill up the shadow transposition table again with information from the next work granule.

When a slave receives a shadow transposition table, the slave inserts the shadow table entries into the slave's local transposition table. APHID then immediately forwards the message to the next peer slave process. Once the message goes around to all of the peers and returns to the original sender, the original sender is allowed to send out new shadow transposition table information. A given slave is not permitted to send out another shadow transposition table until the message completes the entire cycle of peer slave processes and returns to the original sender. Thus, each slave can have at most one outstanding shadow transposition table being sent amongst its peer slave processes.

It may seem wasteful to immediately forward peer slave update messages, especially if we receive the message in the middle of a search. Unlike other messages in the APHID system, the peer slave updates are time-critical. If a peer slave update takes too long to reach a process, the search results within the message may have already been replicated by the process. Thus, the utility of the message decreases as the message transmission time increases. Therefore, it is imperative to get the table entries to all peer processes as quickly as possible.

The size of the shadow transposition table is another important factor to consider. As the number of shadow table entries increases, it takes more time to process the message at each slave process. We would prefer a small and fast shadow table over a large and slow shadow table, because the messages are time-critical. However, if the messages are too small, they are less likely to assist the search. In our tests in Section 6.4, having a shadow transposition table with 128 entries was sufficient for distributing useful and timely transposition table information between 15 slaves.

There are better schemes for broadcasting information, depending on the network architecture being used. However, all communication might be serialized through a small number of processes. Improved schemes which use hypercube-like links to parallelize message sending are not useful under these situations. In some cases, sending out more messages in parallel may hurt the overall performance of the APHID algorithm by slowing down important master/slave messages. Thus, the simplistic broadcast mechanism described earlier is used in the current version of APHID.

One drawback of the shadow transposition table algorithm is that the process has no idea whether the information that is being broadcast is useful for another process. Thus, a second hash table is used to retain the hash table locations that have been broadcast by other peer slave processes. The general concept is that if a process generates the same 32-bit hash key as a hash key broadcast by another process, it is likely that the two processes are searching the same position. In that case, the 32-bit hash key is flagged as important, and any update to the information with that hash key will be stored in the shadow transposition table, irrespective of search depth. This second hash table is much larger in size than the shadow transposition table. For the tests run in Section 6.4, the second hash table has 64K entries.

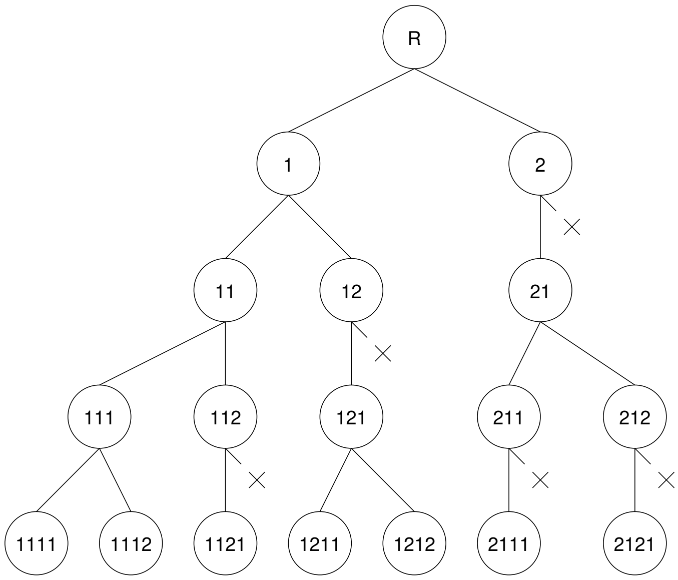
5.3 APHID in Operation: An Example

This section attempts to illustrate some of the algorithms and decisions during the operation of a parallel search with the APHID algorithm. The example will illustrate the slave scheduling algorithms, show examples of combining guessed and real minimax values, show how APHID balances the load on many slaves, and illustrate how pieces of work are created, moved, and removed from scheduling.

We commence at a point in time where Figure 5.5 represents the current state of the master's tree and of each of the 3 slaves assisting the master.

The master's tree is represented on the left-hand side of Figure 5.5. The master has set the parallelization horizon for the search at $d' = 4$ ply, and the application is currently trying to search a 10-ply search tree.

For each position, there are only two move alternatives in the game tree to be



SLAVE 1	1111			
1111	23	5	=14	
1211	11	5	>=14	
2121	7	5	<=12	
2222	0	2		

SLAVE 2	1121			
1121	15	5	<=14	
1212	11	5	>=15	
2122	0	2		
2212	0	3		

SLAVE 3	2111			
1112	19	6	=16	
2111	7	5	<=13	
2211	0	3		
2221	0	2		

Figure 5.5: APHID Example, Part 1: Snapshot of Master Tree and Information from APHID Table for each Slave Process

examined. An X appears in the diagram to indicate a path that is not explored. We have labelled each position in the tree depending on the moves required to reach the node. Starting at the root node (R), by investigating the first move (1) for the player and the second move (2) for the opponent, we reach position 12 in the game tree.

The information on the right-hand side of Figure 5.5 represents some of the important information for each slave process. The upper left field in each rectangle gives the identifier for the slave. The number in the upper right field represents the leaf node that the slave is currently searching. For example, Slave 2 is searching position 1121, while Slave 3 is searching position 2111.

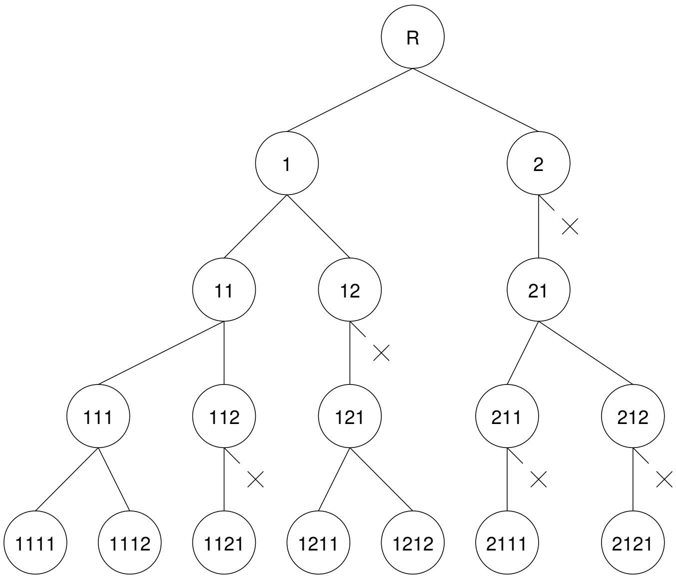
The columns in the bottom part of each rectangle represent the work that each slave has attributed to them, along with the important statistics. The first column gives the leaf node identification. The second column gives us the priority of each piece of work. For example, leaf 1211 (Slave 1) has a priority of 11, while leaf 2221 (Slave 3) has a priority of 0. The third column represents the depth to which the leaf node in the first column has been searched by the slave. For example, leaf 1211 has been searched to 5 ply, but leaf 2221 has only been searched to 2 ply. The fourth column gives us either the accurate minimax value returned for the correct depth of search or the scaled guessed minimax value, depending on the depth of search. For example, leaf 1112 has a minimax value of 16 which is accurate, since we are using 6-ply results from the slave to generate values for the full search depth of 10 ply. Leaf 2121 is using a guessed minimax value of ≤ 12 since it has only been searched to 5 ply on a previous iteration. It is important to note that the 9-ply minimax value for the root node is 14, and that the best information for the current 10-ply search is the same as the 9-ply value. Hence, there is no difference between the 9-ply minimax value and the guess at 10 ply. The fifth column, which is blank for all nodes in Figure 5.5, will represent the "bad bound" search information (the search window and the depth of search).

Now that we have described the information provided for each slave process, we can note that each slave in Figure 5.5 is searching the highest priority node that has not been searched to 6-ply. Slave 3 has already searched the highest-priority node, 1112, to 6-ply, and must search the only other active node, 2111.

Until one of the three slaves returns some score information to the master, the master sits idle. Only when a score is returned is another pass of the tree executed. In our example, we will simulate Slave 3 finishing the search of node 2111 to 6-ply, and returning a bound of ≤ 14. We can see the changes in the master's next pass reflected in Figure 5.6.

Information that changes in between the figures in this example are emphasized with bold-faced text. Note that the depth of search has changed from 5 ply to 6 ply for leaf 2111. The minimax value shown for leaf 2111 (≤ 14) is an exact minimax value, instead of a guessed minimax value. Also note that since Slave 3 has now searched all of its non-zero priority nodes to 6 ply, it starts working on extending all of its non-zero priority nodes to 7 ply, starting with the highest priority node, 1112.

When the master gets the 6-ply result from Slave 3, it now recognizes that Slave 1 is the most overworked process, while Slave 3 has finished all of its work. Thus, the master moves the node 2121 from Slave 1 to Slave 3, resulting in Figure 5.7.

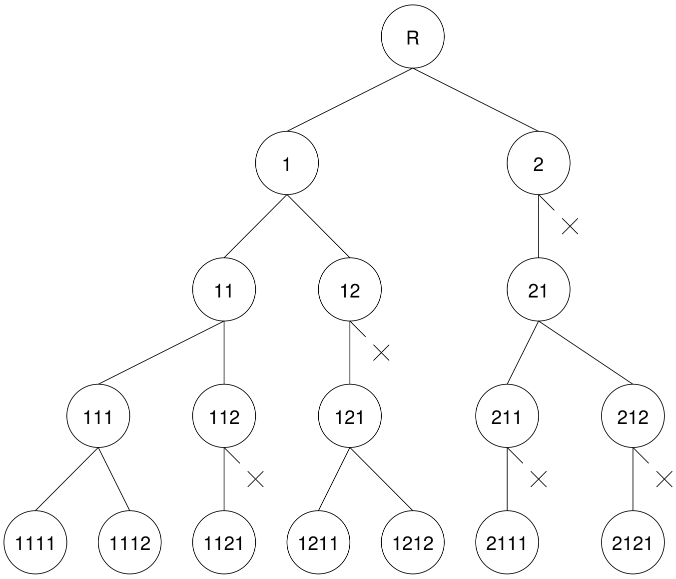


SLAVE 1	1111			
1111	23	5	=14	
1211	11	5	>=14	
2121	7	5	<=12	
2222	0	2		

SLAVE 2	1121			
1121	15	5	<=14	
1212	11	5	>=15	
2122	0	2		
2212	0	3		

SLAVE 3	**1112 (SPECULATIVE)**			
1112	19	6	=16	
2111	7	**6**	**<=14**	
2211	0	3		
2221	0	2		

Figure 5.6: APHID Example, Part 2: Slave 3 is Searching Speculatively

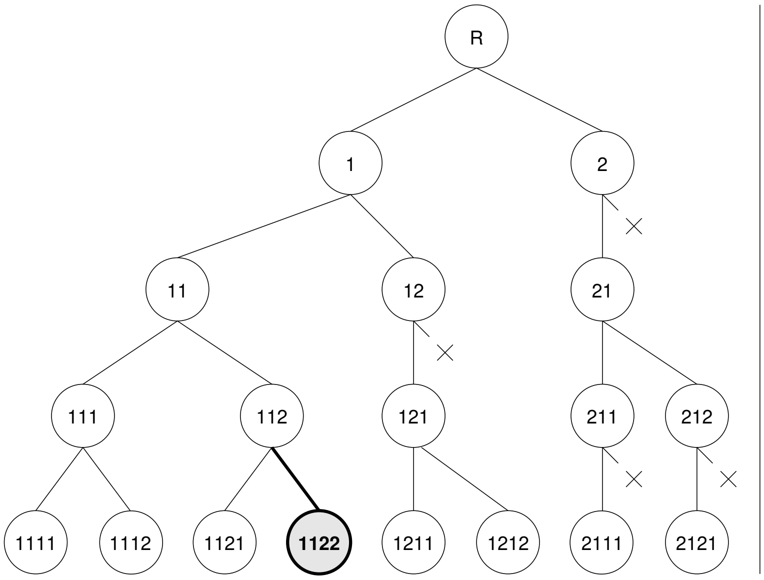


SLAVE 1	1111			
1111	23	5	=14	
1211	11	5	>=14	
2222	0	2		

SLAVE 2	1121			
1121	15	5	<=14	
1212	11	5	>=15	
2122	0	2		
2212	0	3		

SLAVE 3	**2121**			
1112	19	6	=16	
2111	7	6	<=14	
2121	**7**	**0**	**=5**	
2211	0	3		
2221	0	2		

Figure 5.7: APHID Example, Part 3: Load Balancing Brings Work to Slave 3



SLAVE 1	1111			
1111	23	5	=14	
1122	**13**	**0**	**=-200**	
1211	11	5	>=14	
2222	0	2		

SLAVE 2	**1212**			
1121	15	**6**	**>=15**	
1212	11	5	>=15	
2122	0	2		
2212	0	3		

SLAVE 3	2121			
1112	19	6	=16	
2111	7	6	<=14	
2121	7	0	=5	
2211	0	3		
2221	0	2		

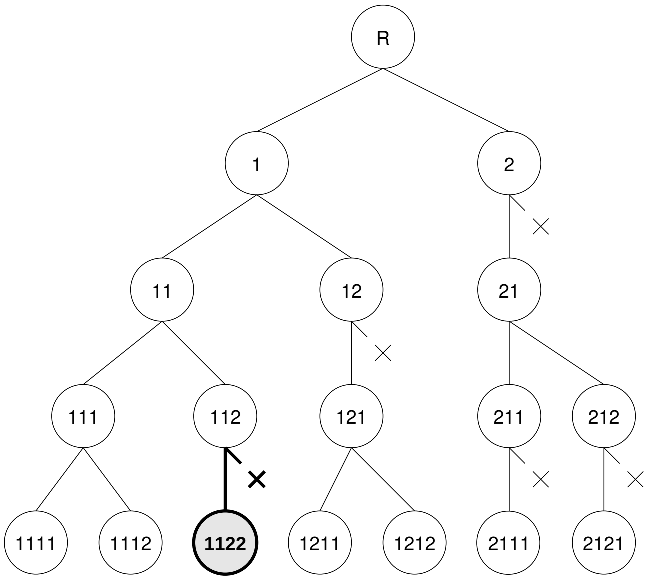
Figure 5.8: APHID Example, Part 4: Creation of a New Leaf Node (1122)

Note that as the work is moved, the previous score information is deleted, since Slave 3 has not generated those results. This results in using the evaluation of the leaf node in the guessed score algorithm. In this case, we have generated a guessed minimax score of 5. Also note that when leaf 2121 arrives at Slave 3, the 7-ply speculative search on 1112 is stopped immediately, and the search on 2121 begins.

We have seen how the search tree is not modified when the returned result falls in line with the guessed minimax value. What happens if the result does not fall in line with expectations? Before the next pass of the master in our example, Slave 2 returns a score of ≥ 15 for the 6-ply search of node 1121. We examine the changes made to the information on the next pass of the master in Figure 5.8.

During the $\alpha\beta$ search, node 1121 no longer yields a score that is sufficient to cut-off the search at node 112. Thus, the master must examine position 1122. Position 1122 is a very bad position for the first player, resulting in a guessed minimax value of -200. Thus, a re-search from node 11 is not necessary, since $\alpha\beta$ will return a score of -200 to node 11.

This is the first time that 1122 has been touched during the search, so it must be allocated to a slave process. Since none of the processes are doing speculative



SLAVE 1	1122			
1111	23	5	=17	
1122	15	0	=-198	
1211	11	5	>=16	
2222	0	2		

SLAVE 2	1212			
1121	0	6		
1212	11	5	>=17	
2122	0	2		
2212	0	3		

SLAVE 3	2121			
1112	19	6	=16	
2111	7	6	<=16	(16,17)/D:6
2121	7	0	=7	
2211	0	3		
2221	0	2		

Figure 5.9: APHID Example, Part 5: Change of PV and Bad Bound Search

work, the work is handed out to the next process in a round-robin fashion. For our purposes, let us assume that the round-robin allocation gave the last new node to Slave 3. Thus, we give this new piece of work to Slave 1. Adding node 1122 on Slave 1 does not interrupt the search of node 1111, since the search of node 1111 is not speculative.

As a final note on Figure 5.8, we should observe that instead of exploring 1121 on the next pass, we will ignore 1121 and explore only node 1122 and attempt to show that it is ≤ 14 for a 6-ply search. Thus, 1121 remains in Slave 2 but will be set to a priority of zero on the next pass of the master's tree. The priority of node 1122 will increase from 13 to 15, since the node will be explored first and become part of the critical tree.

We will show another example of how the master's tree and slave information can change for the fifth part of this example. In this final part, slave 1 returns the 6-ply result of =17 for node 1111, which is different from the predicted value of =14. This will change the PV on the next pass of the tree from the path leading to node 1111 to the path leading to 1112. Furthermore, the change will also change the hypothetical minimax value. These changes are reflected in Figure 5.9.

Note that the exact minimax values returned for 6-ply searches do not change. The hypothetical minimax value of the entire tree is 16 after finishing the search under node 111, instead of 14. Thus, all guessed minimax values are increased by 2 to reflect this change in the hypothetical minimax value.

Node 2111 (on slave 3) poses a problem because it is searched with the search window (16, 17) in this pass of the master's tree. The 6-ply result returned by the slave after the first pass in our example (≤ 14) does not yield any useful information for this search window. Thus, we must use a guessed minimax value for the node (represented by <= 16 in the fourth column), and instruct the slave to do a "bad bound" search to a depth of 6 ply with the search window (16, 17). This information is encoded in the fifth column for Slave 3 in Figure 5.9.

As a final note on the example, we should observe that on the subsequent pass of the tree, the location of nodes 1111 and 1112 will be reversed if we have reasonable ordering techniques in the $\alpha\beta$ routine, since 1112 lies on the principal variation.

We hope this illustration of the master's tree and slave information evolving over time has left the reader with a greater understanding of how the procedures described in Section 5.2 fit together.

5.4 External Interface of the APHID Algorithm

The APHID algorithm has been written as an application-independent library of C routines. The library was written to provide minimal intervention into a working version of sequential $\alpha\beta$ or its common variants: NegaScout, PVS, or MTD-(f). Since the library is application-independent, a potential user must write a few application-dependent routines (such as move format, how to make/unmake moves, position format, setting a window for a slave's search, *etc.*).

In this section, we will start by describing how APHID modifies existing code, and then describe the application-dependent routines (or call-back functions) that the programmer is required to write. Finally, we will discuss the configuration file and how the APHID library instantiates the processes.

Before we start the descriptions, we should note that Appendix B contains specific information on what each `aphid_` function accomplishes, the types of each parameter, and the required tasks to be carried out by each of the call-back functions.

5.4.1 Modifications of Existing Code Required for APHID

To parallelize a sequential $\alpha\beta$ program, the user modifies his or her search routine as shown in Figure 5.10. The changes required by APHID are marked by shading, and easily fit into the standard $\alpha\beta$ framework. This one piece of code functions as the search algorithm for both the master and the slave processes.

The code is very similar to the code for NegaScout presented earlier (Figure 2.9), but we have shown more of the code here to illustrate some of the changes that are required. The main difference between Figures 2.9 and 5.10 is that the transposition table code was encapsulated in `GenerateSuccessors` in Figure 2.9. Although the code given here shows a transposition table implementation, APHID does not require a transposition table to work properly. The first highlighted change (marked `(1)` on Figure 5.10) is not required if a transposition table implementation is not present. The final change (marked `(8)`) is only required if we wish to use APHID's distributed transposition table algorithm.

Let us examine each of the highlighted changes from Figure 5.10 in detail. The first shaded code example illustrates that any master process should not use the transposition table score information to immediately terminate the search or curtail the search window. The main reason for not using the transposition table score information is that the master may be required to make multiple passes over the top of the game tree. If we use the score information, information from the first pass of the tree (which will be based on numerous guessed minimax values) will immediately terminate the search during the second pass. Thus, we must prevent a master from using the score information. On the other hand, using the best move advice from the transposition table is critical and should not be forbidden.

```
int NegaScout(position p, int alpha, int beta, int depth, int plytogo) {

   move movelist[MAX_LEGAL_MOVES];       /* ordered list of all moves */
   int numOfSuccessors;        /* total number of moves in movelist[] */
   int gamma;                           /* current best minimax value */
   int i;                                           /* move counter */
   int sc;                                /* score returned by search */
   int under;                        /* alpha for move to be searched */
   int over;                          /* beta for move to be searched */
   move move_opt;                                /* current best move */
   char *p_entry;         /* pointer to physical location of TT entry */
   char *p_hash;                              /* pointer to hash value */
   char *p_key;                         /* pointer to hash table lock */
   int h_length;               /* ply position previously searched to */
   int h_score;                      /* score at h_length ply from TT */
   int h_flag;             /* type of score (VALID, LBOUND or UBOUND) */
   move h_move;                           /* recommended move from TT */

   /* Generate hash value and key for this position */
   generate_hash(p, p_hash, p_key);
   /* Fetch information from transposition table  */
   retrieve(p_hash, p_key, p_entry, h_length, h_score, h_flag, h_move);

   /* If we have searched position deep enough, use score info */
   if (aphid_master() == FALSE && h_length >= plytogo) {                 (1)
      if (flag == VALID)  { return(h_score); }
      if (flag == LBOUND) { alpha = max(alpha,h_score); }
      if (flag == UBOUND) { beta = min(beta,h_score); }
      if (alpha >= beta)  { return(h_score); }
   }

   /* Evaluate position if at bottom of tree */
   if (plytogo == 0) { return(Evaluate(p)); }
   if (aphid_horizon(depth, plytogo, p_hash, p_key) == TRUE) {           (2)
      return(aphid_eval_leaf(alpha, beta, depth, plytogo, p_hash, p_key));
   }

   /* Generate move list, evaluate position if no moves */
   numOfSuccessors = GenerateSuccessors(p);
   if (numOfSuccessors == 0) { return(Evaluate(p)); }
   if (aphid_checkalarm(FALSE) != FALSE) {                               (3)
      terminate_search = TRUE;
      return(0);  /* Should exit Negascout quickly when alarm on */
   }

   gamma = -INFINITY;
   under = alpha;   over = beta;
   aphid_intnode_start(depth, p_hash, p_key);                            (4)

   for(i=1; (i <= numOfSuccessors && score <= beta); i++) {
      aphid_intnode_move(depth, &(movelist[i]));                         (5)
      make_move(p, movelist[i]);

      sc = -NegaScout(p,-over,-under,plytogo-1);
      /* Is a research necessary? */
      if (sc > under && i > 1 && sc < beta && plytogo > 2) {
         sc = -NegaScout(p,-beta,-sc,plytogo-1);
      }

      unmake_move(p, movelist[i]);
      aphid_intnode_update(depth, value);                                (6)
      if (sc > gamma) {
         gamma = sc;
         move_opt = movelist[i];
      }

      /* set window for next child */
      under = max(gamma, alpha);  over = under + 1;
   }

   aphid_intnode_end(depth, score, beta);                                (7)

   /* Write information into transposition table */
   h_flag = VALID;
   if (score <= alpha) { h_flag = LBOUND; }
   if (score >= beta) { h_flag = UBOUND; }
   if (h_length <= plytogo) {
      p_entry = store(p_hash, p_key, plytogo, score, h_flag, move_opt);
      aphid_distt_insertentry((int) *p_hash, p_entry, plytogo);          (8)
   }

   return(gamma);

} /* NegaScout */
```

Figure 5.10: Code Example: APHID within the NegaScout Algorithm

The second highlighted code sample in Figure 5.10 illustrates the artificial search horizon created by the APHID library. The `aphid_horizon()` call returns `TRUE` when we have hit the parallelization horizon for the master. `aphid_eval_leaf()` determines the correct or guessed minimax value, as described in Section 5.2.1. The other parameters to the `aphid_horizon()` call are necessary because the function is responsible for detecting and extending the horizon for exempted pieces of work.

The third highlighted code sample shows the `aphid_checkalarm()` function. This function computes how many nodes have been searched, and calls the communication routine after a fixed number of calls to determine if there are any messages pending for the process. The code encapsulated here may receive, process and forward peer slave requests as well as obtain new information from the master. The most important piece of information to be obtained from the master is whether or not the search has been terminated. If `aphid_checkalarm()` returns a non-zero number, the current search must be immediately aborted. Note that there are two possible reasons for terminating the search: the master has asked all slaves to stop processing, or the slave is working on a speculative piece of work and a new piece of work for the current search depth has arrived. `terminate_search` is an example of a global variable that is set to terminate the search when a real-time constraint is reached. This application-dependent variable, or some other routine that accomplishes a shutdown of a search in progress, is normally present in game-playing programs.

The highlighted examples marked `(4)` through `(7)` in Figure 5.10 represent the calls that the application must execute if the process is a master in the hierarchy. The `aphid_intnode_` family of routines is designed to gather and obtain information from a pass of the master's tree.

`aphid_intnode_start()` initializes the processing of a master's node. If the function is called, `aphid_intnode_end()` must also be called to terminate the gathering of information.

`aphid_intnode_move()` intercepts the move path information that we are examining so that it can be transmitted to the slaves. `aphid_intnode_update()` receives

```
aphid_initsearch(MAXDEPTH);                                                    (1)
for(plytogo = 1; plytogo <= MAXDEPTH && done == FALSE); plytogo++) {
   /* Set up search */
   /* Search at root around value (guess) with small error (eps) */
   /* Call to aphid_rootsearch replaces call to NegaScout */                   (2)
   score = aphid_rootsearch(0, plytogo, guess-eps, guess+eps);
   /* Print out results of search */
}
aphid_endsearch();                                                             (3)
```

Figure 5.11: Code Example: APHID within the Iterative Deepening Loop

the score information so that the code can accumulate information on the bounds of the minimax value at a specific node within the master's tree.

The final highlighted change (`(8)` in Figure 5.10) illustrates how the transposition table information is absorbed into the APHID library for use by the distributed transposition table implementation. The `p_entry` pointer gives the location where the transposition table information was written to. As stated earlier, this addition to the code is not required if the distributed transposition table implementation in APHID is not used.

There are only a relatively small number of changes to be made outside of the $\alpha\beta$ algorithm, but they are critical to APHID's performance. The first piece of code that we should examine is the iterative deepening loop. A sample of this loop is given in Figure 5.11.

There are three highlighted changes in Figure 5.11. The first is the call to `aphid_initsearch()`. This call tells all of the slaves that we will be starting a new search. With the help of some `aphid_stub_` functions described in Appendix B, the function also transmits the current state of the game to each slave so that the slaves can start examining the root position, while waiting for the master to give it work to do.

The second highlighted change in the iterative deepening example is the call to `aphid_rootsearch()`. If running in parallel, this routine is the main loop that keeps the master cycling over the d' ply tree until we have no uncertain nodes. If running sequentially, this routine simply calls the regular search function, and returns the minimax value back to the application.

The final highlighted change is a call to `aphid_endsearch()`. This routine tells

```
aphid_initsearch(MAXDEPTH);                                                (1)
for(plytogo = 1; plytogo <= MAXDEPTH && done == FALSE); plytogo++) {
   /* Set up search */
   /* Search at root around value (guess) with small error (eps) */
   score = Root_NegaScout(root_pos, guess-eps, guess+eps, 0, plytogo);
   /* Print out results of search */
}
aphid_endsearch();                                                         (2)

Root_NegaScout(root_pos, alpha, beta, depth, plytogo)
...
{
   ...
   /* Search PV Move */
   lower = alpha;

search_best_move:
   aphid_intnode_premove(depth, &bestmove);                                (3)
   make_move(p, bestmove);
   /* aphid_rootsearch replaces call to NegaScout */                       (4)
   oldscore = -aphid_rootsearch(depth+1, plytogo-1, -beta, -lower);
   unmake_move(p, bestmove);

   /* Search other moves at root, and only swith if move beats */
   /* PV score (oldscore) by a small margin (delta)            */

   newscore = oldscore;
   for (i=2; (i<=numOfSuccessors && newscore<=oldscore+delta); i++) {
      aphid_intnode_premove(depth, &(move[i]));                            (5)
      make_move(p, move[i]);
      /* Check if move beats PV move by more than delta */
      newscore = -aphid_rootsearch(depth+1, plytogo-1,                     (6)
                        -oldscore-delta-1, -oldscore-delta);
      unmake_move(p, move[i]);
   }

   if (newscore > oldscore + delta) {
      /* set new best move/score and research (if necessary) */
      bestmove = move[i];
      lower = newscore;
      if (newscore < beta) { goto search_best_move; }
   }
   ...
} /* Root_NegaScout */
```

Figure 5.12: Code Example: APHID in Code to Handle the Root of the Game Tree

all slaves to stop searching the current position. The slave processes will sit idle until the search is started again with another `aphid_initsearch()` call.

Initially, it was anticipated that all users would want to search in parallel from the root of the game tree. However, there are some applications that wish to handle the root of the game tree in a different way than the other leaves of the search tree. For example, we could add calls to the time-control mechanism or modify the search window at the root. APHID has been generalized to integrate with this style of searching the game tree; Figure 5.12 illustrates the changes necessary. The `aphid_intnode_premove()` calls are the only significant changes from Figure 5.11. This allows the master process to determine the prefix of all of the move paths that should be handed to the slaves.

The prefix also allows APHID to remove only those nodes that should not be examined by the slaves. Recall that if a node is not touched within the pass of a tree,

```
int main(argc, argv)
int argc;
char *argv[];
{
   /* Initialization required by any process in system */

   aphid_startup(argv);                                          (1)
   /* Only the absolute master process gets here */

   /* Initialization required only by the absolute master process */
   /* Play game */

   aphid_exit();                                                 (2)
   exit(0);
} /* main */
```

Figure 5.13: Code Example: APHID within the Main Program

its priority is set to zero. However, if the prefix on the untouched piece of work is not the same as the prefix that we are currently searching, we should not stop scheduling the piece of work because we did not expect to see the piece of work within the current search. Instead, APHID lowers its priority and speculatively works on the pieces with different prefixes if a slave has nothing better to do.

There are two calls that are inserted into the main program, which are illustrated in Figure 5.13. `aphid_startup()` is the routine which starts up PVM (if necessary), spawns slave processes, and allows the process to determine where it belongs in the hierarchy. `aphid_exit()` should be called before any process terminates in the system.

5.4.2 Application-Dependent Knowledge in APHID

Since APHID is designed as a general-purpose system for any algorithm that uses $\alpha\beta$ as the core search mechanism, the programmer must define a number of application-dependent constants. For example, some of the constants that APHID needs to know are how big the hash type and key are, how big a transposition table entry is and what the minimum and maximum values for the evaluation are. These constants are covered in greater detail in Appendix B.2.

Since the code does not rely on direct access to the application's data structures, the APHID library also needs the programmer to provide a series of application-dependent call-back routines to perform some tasks. These include encoding a board position, decoding the encoded board position, determining the next step of iterative deepening and how to call the evaluation function and search routines. Specific information on the stubs can be found in in Appendix B.3.

5.4.3 Configuration File

Although a number of configuration parameters are compiled directly into the code, a number of important parameters can be modified without recompiling the code. All of these parameters are available in the configuration file entitled `aphid.config`, which should be available and readable by all processes in the hierarchy. A sample configuration file is illustrated in Figure 5.14.

The numbers on the first line represent the minimum size of each piece of work (in terms of non-terminal nodes), and the minimum number of pieces of work that each slave process should get, respectively. We use non-terminal nodes to count the size of work in APHID, since the count is maintained through calls to `aphid_checkalarm()`. This function is usually not called at leaf nodes where we return an evaluation. For example, in Figure 5.10, we see the call to `Evaluate()` before the second highlighted change, and the call to `aphid_checkalarm()` is in the third highlighted change. Because APHID does not include leaf nodes in the count, the minimum size of each piece of work is usually very small. For the tests in the subsequent chapter, the largest size chosen for any of the test runs was 25 non-terminal nodes. As an example, if we attempt to distribute work to 64 processors, each requiring at least ten pieces of work of with a minimum of 25 non-terminal nodes, parallelism does not usually start up until a 6 or 7-ply search tree is attempted.

The lines after the two numbers each represent the host names that the slave processes should be started on. The absolute master process is not included in this list, because it is chosen by where the first program is run by the user.

The optional `#` argument after the host name gives the user the ability to run a different executable. If the `#` argument is missing, the program will attempt to spawn the same executable as the absolute master process.

In Figure 5.15, we see a representation of the same hierarchy as a process tree. Each rectangle or arced rectangle represents a process within the system. The host

```
5 10
slinky
ralph.cs.ualberta.ca
lego#special.slave
carebear
barbie
```

Figure 5.14: Flat Hierarchy, APHID Configuration File

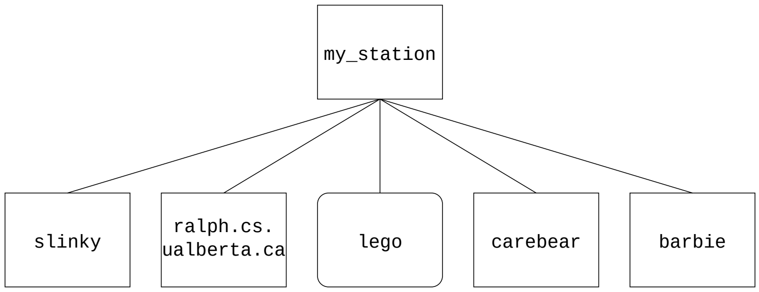


Figure 5.15: Flat Hierarchy, Process Tree Representation

that the process is being run on is represented in the box. For the process tree, we have assumed that the first process has been started on a host with the name `my_station`. The shape of the object on the process graph denotes what executable the process is running. For example, the process running on `lego` is not using the same executable (`special.slave`) as the processes running on all of the other hosts. Thus, the process on lego has an arced rectangle instead of a regular rectangle in the process graph. Each line in the process tree represents a master/slave relationship. In this simple example, there is a single master and five slaves.

The hierarchy given in Figure 5.14 is a flat hierarchy; all slaves report to a single master process. To make a given process the slave of a slave, we use tabs before the host name to represent levels within the hierarchy. A complicated hierarchy of processes is illustrated by the sample `aphid.config` file in Figure 5.16.

The first thing to note about the complex hierarchy is that the machine names are actually IP addresses on the Internet. With PVM installed on the target machines, APHID can establish a virtual machine that will start up and distribute work around

```
5 10
slinky.bu.edu
      tonka.bu.edu#special.slave
      lego.bu.edu#special.slave
sawnlk.cs.ualberta.ca
      sundance.cs.ualberta.ca#special.slave
      sunset.cs.ualberta.ca#special.slave
      ipiatik.cs.ualberta.ca#special.slave
      charron.cs.ualberta.ca#special.master
            charron.cs.ualberta.ca#special.slave
            charron.cs.ualberta.ca#special.slave
            charron.cs.ualberta.ca#special.slave
            charron.cs.ualberta.ca#special.slave
      sundance.cs.ualberta.ca#special.slave
xolas0.lcs.mit.edu#special.slave
```

Figure 5.16: Complex Hierarchy, APHID Configuration File

the world. It should be noted that the current version of the APHID library does not support multiple types of processors participating in the same search. For example, an SGI Origin 2000 could not have an Intel Pentium as a slave because the elementary data types are not equivalent on the two machines. Support for heterogeneous computing can be added to APHID, at the cost of additional CPU time in PVM's message packing and unpacking routines.

Each tabular indentation represents how deep in the hierarchy the given process is. For example, the process running on `slinky` is a slave of the absolute master, but the processes running on `lego` and `tonka` are slaves to the process on `slinky`. Also note that although `slinky` has only two slaves within its hierarchy, the process running on `sawnlk` has five slaves, and that the process on `charron` has four processes on the same machine that report to it. The hierarchy is easier to visualize when represented as a process tree. The complex hierarchy is given as a process tree in Figure 5.17.

In Figure 5.17, we note that each process is represented as a rectangle, arced rectangle or a ellipse to represent the executable that is run in each case. The rectangle represents processes that are using the same executable as the root of the processor tree. The arced rectangles represent processes that are running `special.slave`, and

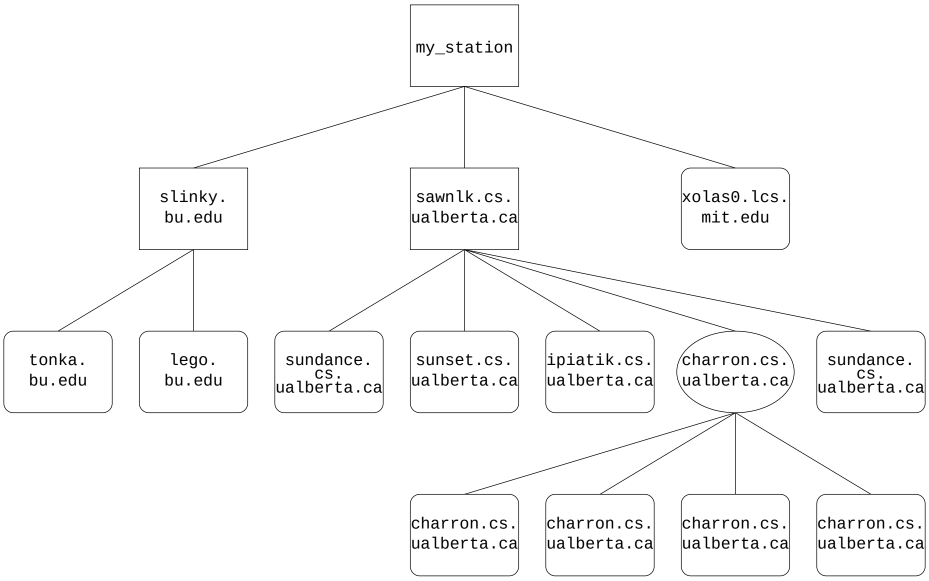


Figure 5.17: Complex Hierarchy, Process Tree Representation

the ellipse represents the process that is running the `special.master` executable.

In the current implementation of the start-up code, the *absolute master* process (the process at the top of the process tree) is responsible for starting up PVM, the message-passing interface, on each host used in the hierarchy. However, each master is responsible for spawning its immediate slaves. Thus, the process running on `slinky` is responsible for starting up the slaves on `tonka` and `lego`, and reports back to the absolute master process on `my_station` to report whether the spawning was successful. Only once all of the processes have been set up will the absolute master return from the `aphid_startup()` call described earlier.

5.5 Summary

We have described the implementation of an asynchronous game-tree search algorithm based on the model from Chapter 4. APHID is asynchronous in nature while a search is being executed; there are no global synchronization points along the principal variation or in between iterations of iterative deepening. The APHID algorithm is designed to work without using shared transposition tables. APHID uses the $\alpha\beta$

search window to naturally handle parallelism at type-2 nodes. APHID is designed to be portable across many application domains, and will work on any hardware platform that supports PVM. Another design goal of APHID is to use inter-process communication sparingly. This allows APHID to behave well on both loosely-coupled and tightly-coupled processors.

Within APHID itself, we have introduced a method of determining a hypothetical minimax value based on previous search information. We have also introduced a new distributed transposition table scheme which does not impact heavily on the interconnection network between a series of machines.

Possibly the most important point is that APHID integrates into existing applications without requiring drastic changes to the application. Other parallel $\alpha\beta$-based algorithms require significant changes to the sequential search algorithms used in practice. If the search algorithm has been designed without regard for multitasking or a specific parallel model, integrating a parallel algorithm into the code can be a significant task. By using the sequential algorithm and call-back functions to the user's code whenever possible, APHID represents a significant decrease in the effort required to achieve a working parallel game-tree search program over its synchronous counterparts.

Most authors illustrate a synchronous parallel algorithm with one application. Chess is usually chosen because synchronous algorithms yield large observed speedups when the branching factor is large. In the next chapter, we will illustrate APHID's performance with four different applications, each written by different authors and with different coding styles. This ambitious comparison of performance in multiple application domains is possible because of APHID's ease of integration into an existing sequential program.

Chapter 6

Experimental Results

6.1 Introduction

In Chapter 4, we compared the performance of an asynchronous parallel search algorithm to the performance of a synchronous parallel search algorithm in a theoretical framework. The comparison showed that it was possible for asynchronous algorithms to outperform synchronous algorithms on realistic game trees.

In this chapter, we will run some performance tests on the version of the APHID game-tree search library described in Chapter 5. We will test the asynchronous parallel algorithm in four different applications, written by four different authors. We will show where the asynchronous algorithm succeeds and fails in comparison to APHID's synchronous counterparts.

Section 6.2 describes the experimental methodology. Section 6.3 describes the standard test runs for the various applications tested in this chapter. Each application parallelized with the APHID algorithm is analyzed separately, and compared against the best results for synchronous parallel search algorithms in the respective domain. Section 6.4 describes a series of experiments that examine the structure of the transposition table in different application domains, and how this affects the overall performance of the APHID library.

6.2 Methodology

6.2.1 The Hardware

Previously published results for the APHID algorithm were run on a network of Sun SPARCstation IPC workstations [15, 16]. For the results in this chapter, two similar hardware configurations were used. The 8 and 16-processor tests were run on a 32-processor SGI Origin 2000 computer system at Boston University. The 32 and 64-processor tests were run on a 64-processor SGI Origin 2000 computer at the Cray Research Facility in Eagan, Minnesota. Both systems contained 195 MHz MIPS R10000 processors and at least 4 gigabytes of RAM available. This is sufficient to run all of the processes without swapping to secondary storage.

An advantage of using the SGI Origin 2000 system is that all of the processors have access to a global distributed shared memory. This allowed us to compare the performance of APHID using local, distributed and shared transposition tables (see Section 6.4). We can use the transposition table structure to emulate many hardware configurations. For example, we can emulate a network of fast workstations by using only local transposition tables.

Another advantage of using the SGI Origin 2000 system for large-scale parallel experiments is the number of installations where large numbers of processors are already available. Acquiring access to large parallel computers can be complicated.

6.2.2 The Applications

For our experiments, we have implemented APHID in four applications written by different groups of authors:

- Chinook, the Man-Machine World Champion checkers program,
- Crafty, Robert Hyatt's freeware chess program,
- Keyano, an Othello program written by Mark Brockington, and

- THETURK, a chess program by Yngvi Björnsson and Andreas Junghanns.

Each of the applications given here were compiled with SGI's `cc` compiler. Each sequential and parallel program was optimized with the `-O2` flag turned on. No time was spent attempting to optimize the search code for the SGI Origin 2000, aside from debugging some bizarre interactions between the code and the hardware, as we shall see in Section 6.2.5.

For the parallel tests, two different versions of the code were compiled. The first parallel program could be used at run-time as a master or a slave by APHID. The second program was an executable that could only be used by APHID as a slave. The only difference between the first and second program was that all of the `aphid_intnode_*()` calls were removed from the main search routine. This allowed the second program to search the pieces of work at a rate close to that of the sequential program. Without this optimization, a slave process in APHID can incur a 10-20% slowdown on the number of nodes visited per second by calling the `aphid_intnode_*()` routines.

6.2.3 Search Extensions and Reductions

For each application, some standards for testing must be imposed. Parallel and sequential algorithms often do not agree with each other about minimax values and best moves when the full version of the program is used [57]. For example, different search windows cause different search extensions to be turned on, causing different minimax values. Thus, all search extensions, search reductions and null move searching were turned off for the purposes of this experiment. Although a fixed depth is enforced on the programs, quiescence search was left in THETURK, CRAFTY and CHINOOK to prevent the evaluations from being significantly unstable. This allowed the parallel and sequential programs to return identical minimax values and principal variations in the majority of the test positions, yielding a meaningful observed speedup.

6.2.4 Search Depth and Time Constraints

A suitable benchmark set was chosen for each game (see Appendix A for the positions used). We originally attempted to run all of the tests to the same depth. This was successful in KEYANO, where the size of any pair of 15-ply searches varied by at most a factor of 3. However, the positions used in the chess and checkers programs produced widely varying search sizes for the same nominal search depth. Thus, some of the positions were searched to greater depths than other positions.

The size of each of the searches is important to the observed speedup, as we shall see in Section 6.3.1. In synchronous parallel game-tree search algorithms, the speedup can be improved arbitrarily by increasing the size of the search. Searching game trees that occur under tournament time controls is critical to assessing a parallel game-tree search algorithm's performance. Thus, we have attempted to limit the size of the individual tests so that the average time spent on the 64 processors should not exceed the usual time controls in the game being studied. This is 180 seconds in chess (40 moves in two hours), 60 seconds in Othello (approximately 30 moves in 30 moves), and 120 seconds in the game of checkers (30 moves in one hour).

6.2.5 Transposition Tables

Using a transposition table that is large enough to accommodate the search results discovered during the search is important to the performance of any game-playing program. Each application allowed the number of transposition table entries to be a power of 2. For each application, we chose the largest transposition table that kept the overall size of the transposition table and main program below 400 megabytes of RAM. Although both SGI Origin 2000 configurations had gigabytes of RAM, this limitation was chosen so that the algorithms could also be tested on an 8-processor SGI Challenge at the University of Alberta. It was unanticipated that this limitation would dramatically affect the speedup on a large numbers of processors, but we will see that 400 MB of RAM is inadequate for the large searches attempted in the chess and checkers programs.

It is very important to note that the size of the transposition table did not increase as the number of processors increased. The sequential tests used exactly the same number of transposition table entries as each of the parallel tests. If the parallel run used 64 processors, each process received 1/64th of the transposition table entries that the sequential process received. Thus, the experiments measured the scalability of the number of processors and not the scalability of memory. Other experiments have allowed the number of transposition table entries to increase in tandem with the number of processors. As mentioned in Chapter 3, the reader should avoid comparing speedups without understanding the conditions under which the speedups were achieved.

The standard set of tests for each program involved examining a fixed-depth game tree and using a shared memory transposition table, over a varying number of processors. Each program was tested on n=16, 32 and 64 processors, using a single-level hierarchy: one master allocated work to $n-1$ slaves for each test position.

Using SGI's distributed shared memory for storing the transposition table, a couple of optimizations were made to the algorithms to assist the performance of the tests. The first optimization was made to the APHID library for all of the shared memory tests. The APHID library usually allows the slave processes to search the root of the game tree when there is nothing to do. However, with SGI's distributed shared memory system, we experienced a severe performance slowdown at the start of the search when all of the processes attempt to access the same memory pages at the same time. Thus, the slaves are not allowed to search the root of the tree when there is no work available. This allowed the master to quickly distribute work without interference from the slaves.

The second optimization was made to the transposition table code for the 64 processor runs in both APHID and YBWC. The SGI distributed shared memory in the Origin 2000 allowed up to 32 processors to access the transposition table with no appreciable slowdown in performance. However, when we moved to 64 processors, the performance on the Origin 2000 drops significantly due to overloading the shared memory with too many requests. To counteract this feature of the SGI hardware, the applications were forbidden from reading or writing to the hash table when they are within one ply of the leaves. This optimization increased the search size by a small margin, while making the programs run significantly faster, yielding an overall performance gain.

6.2.6 Overheads in APHID

The overheads in the APHID algorithm will be illustrated in the analysis of the results. Since the overhead model used in this chapter is slightly different than that used by other authors, we must define the terminology used in this chapter.

The *total overhead* represents the additional computing time required by the parallel algorithm to achieve the same result:

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

where n is the number of processors. The total overhead can also be computed by examining the overheads. The three main overheads are using a processor exclusively as a master (the master overhead), the effective decrease in nodes per second examined (parallelization overhead), and the additional number of nodes searched by the parallel algorithm (total search overhead). There is no synchronization overhead in the APHID algorithm since the algorithm operates in an asynchronous manner. The breakdown of overheads can be expressed in the following formula[1] for total overhead:

$$\begin{aligned} \text{total overhead} \quad &= \quad (1 + \text{master overhead}) \times (1 + \text{parallelization overhead}) \\ &\quad \times (1 + \text{total search overhead}). \end{aligned}$$

The *master overhead* is the approximate penalty incurred by having a single processor being allocated completely to the handling of the master. This is simply $1/(n-1)$, the benefit of adding another slave to the other $n-1$ slaves.

[1]This formula is not the same as the formula presented in earlier publications [15, 16] where an additive formula for computing the total overhead was presented.

The *parallelization overhead* is the penalty incurred by the APHID library on the speed of the slaves. The difference between the rate at which the parallel slaves explore nodes and the sequential program's node rate is assumed to be the parallelization overhead. This parallelization overhead is derived partially from the overhead of using PVM, and partially from the work-scheduling algorithm on each slave. In effect, the parallelization overhead includes synchronization overhead, complexity overhead and communication overhead, as used in previous parallel $\alpha\beta$ models [57, 63, 81].

The *total search overhead* represents the number of additional nodes searched by the algorithm while attempting to determine the d-ply minimax value. In the case of APHID, we have two types of nodes that combine to make the total search overhead: the search overhead and the speculative search.

The *search overhead* represents the additional nodes searched to achieve the d-ply minimax value. This can be computed by dividing the nodes searched to generate d-ply search results in the parallel program by the nodes searched by the sequential program. 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. If we do not use a shared-memory transposition table, some of the increase in search overhead as we increase the number of processors can be attributed to information deficiency, since data is not shared efficiently between the processes.

The remainder of the increase in search overhead is attributable to the load-balancing algorithm. The APHID algorithm forces work to be recalculated when it is moved to another processor. When there are more processors in the system, the load-balancing algorithm is more active in balancing the workload, thus causing more search overhead.

Since we only search each position to d ply, the asynchronous nature of the slaves will result in some work being done at $(d+1)$ ply or more. The *speculative search* represents the amount of additional search that APHID has undertaken which is beyond what the sequential algorithm has attempted for a d-ply search. The speculative search can be computed by taking the number of speculative nodes searched and dividing that by the number of nodes searched in the sequential case. In our experiments, the speculative search results were not used so that the parallel program produced the identical results as the sequential version, verifying APHID's correctness.

It is important to note that ignoring the speculative search results will understate the potential of the APHID algorithm. 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 variations extended a ply ahead would be in the left-most branches of the tree. Thus, it is very likely we have searched the key variations not only to d-ply, but to $(d+1)$-ply. This allows APHID to find important variations much sooner than a synchronous parallel program with the same observed speedup.

If we wish to measure the results of the speculative search in each of the programs, we must be able to test the algorithm using a different measure other than observed speedup for a given search depth. There are a number of test sets that measure move quality in chess, where the performance metric is how quickly a program determines the best move in a tactical position. However, most of the published results for parallel chess programs have focused only on the observed speedup. Thus, it is difficult to compare any move quality experiment (aside from observed speedup) against the results in the literature. Furthermore, there are no tests that measure move quality in Othello. If we wish to use a common methodology for all of the programs to be tested, we must formulate an equivalent test set for Othello, a game that relies on positional play during the middle game. For these reasons, we have resorted to comparing observed speedups.

6.2.7 Results Reporting

There are two methods of reporting speedups and overheads for a large number of test positions. The first is to add all of the searches for the multiple positions together, and perform the speedup and overhead analysis on the combined data. The second method is to perform the speedup and overhead analysis on each individual position, and average the speedups and overheads.

Both methods have drawbacks. In game-tree search, some positions will yield smaller search trees than the average search, and a few positions will yield much larger search trees. Since smaller searches, in general, do not perform as well as larger searches, this will inflate the aggregate observed speedup if we use the first method. Using the second method will underestimate the aggregate observed speedups.

For example, let us take the results of Feldmann's Young Brothers Wait Concept [29]. Feldmann's analysis of the speedups and overheads used the first method to generate a speedup of 142.82 on 256 processors. However, close analysis of the 24 positions reveals that the four largest positions account for 45% of the time spent on the entire test set. The speedups of 280, 228, 124 and 220 bias the aggregate speedup upwards. If we use the second method to determine the average speedup that we would see over the 24 positions, the speedup would be only 119. Even if we discard the three smallest searches (combined, they take 0.2% of the total time for all 24 positions for the sequential runs), the average speedup is only 133 using the second method.

We have chosen to use the second method in this thesis, because it will not overestimate the observed speedup for a large number of test positions. However, it is important to note that the speedups and overheads are independently averaged over 24 positions. For each position, the speedups and overheads will follow the formulas given earlier, but the mean of the speedups and overheads presented in the next section will not necessarily follow the formula.

6.3 APHID - Standard Tests

The section, describing the main results for each program, will be subdivided into three subsections, dealing with each of the application domains in turn.

We will start with the Othello program Keyano, followed by the checkers program Chinook, and the two chess programs Crafty and TheTurk.

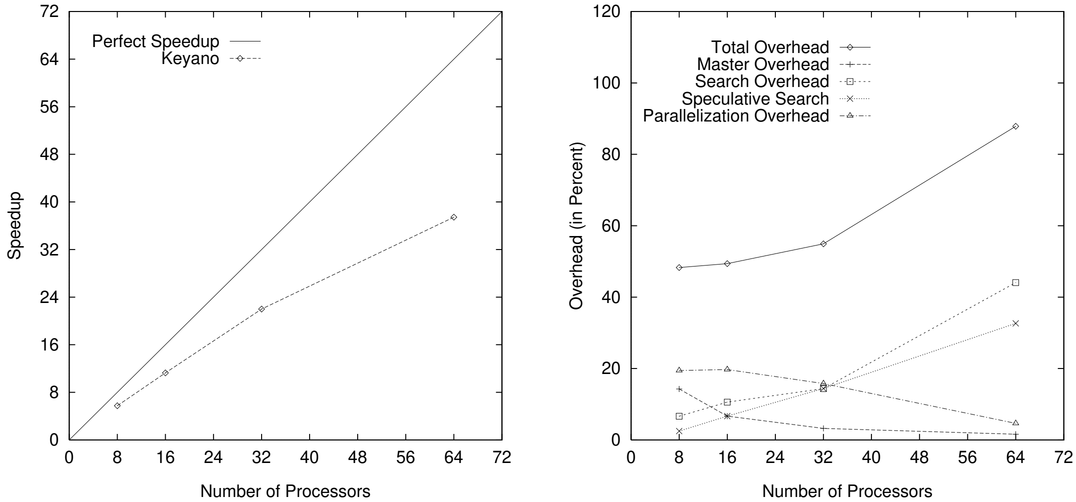


Figure 6.1: Speedups and Overheads for KEYANO (Fixed-Depth, Shared Memory)

6.3.1 KEYANO - Othello

KEYANO [14] is an Othello program written by Mark Brockington. KEYANO routinely finished in the top five in international computer Othello competitions, up until its retirement from tournament play in late 1996. Since 1994, the program has been used as a research tool for parallel game-tree search algorithms.

We will examine an implementation of the APHID algorithm in KEYANO, and analyze APHID's performance. We will compare and contrast the results with a hand-optimized version of the algorithm yielding the best observed speedups for synchronous parallel game-tree search, Young Brothers Wait Concept [29].

The 20 positions in Appendix A.3 come from the early midgame of the 1994 World Othello Championship match between Emmanuel Caspard and David Shaman. Each position was searched to a fixed depth of 15 ply for this experiment. Figure 6.1 gives the speedups and overheads for the fixed-depth shared memory version of KEYANO. The data is also given in a tabular format in Table 6.1.

In Figure 6.1, we see that the observed speedup is nearly linear in the number of processors. However, the results are tapering off as we approach 64 processors. On 64 processors, we see that the total time taken for the 20 test position is approximately 70 seconds per position. This is close to typical time control of 60 seconds per move

n	Speedup	Total Time	Total Overhead	Master Overhead	Search Overhead	Speculative Search	Parallelization Overhead
1	-	53526 s	-	-	-	-	-
8	5.74	9424 s	48.29%	14.28%	6.64%	2.48%	19.45%
16	11.27	4753 s	49.37%	6.67%	10.59%	6.67%	19.71%
32	21.99	2415 s	54.94%	3.22%	14.37%	14.41%	15.80%
64	37.44	1395 s	87.86%	1.59%	44.07%	32.65%	4.65%

Table 6.1: Speedup Data for KEYANO (Fixed-Depth, Shared Memory)

in the game of Othello.

The average overheads in APHID are illustrated in Table 6.1. Note that these overheads are the average of the observed overheads taken over 20 different searches. Thus, we do not expect the total overhead to be an exact combination of the overheads, as given by the formula in Section 6.2.6.

As the number of processors increases, we see that the master overhead and the parallelization overhead decreases. On 8 and 16 processors, the parallelization overhead is quite high. It is probable that there is a minor difference between the 32-processor and 64-processor SGI Origin 2000 machines used in the experiments, making the parallelization overhead drop at 32 processors. The decrease in the parallelization overhead from 32 to 64 processors is due to the transposition table code change described in Section 6.2.5.

In an algorithm that does not synchronize, it should not be surprising to discover that the largest portion of the total overhead is accounted for by the search overhead and the speculative search. As the number of processes increases, the processors get fewer pieces of work and the need for load balancing increases. This also allows some of the slaves to get further ahead than the overworked slaves, causing a rise in the speculative search.

As we shall see in later sections, the observed speedups for KEYANO are much higher than the equivalent speedups for the chess and checkers programs. One possible explanation for this discrepancy is that the Othello algorithm is not an efficient searcher. However, this is not the case. Experiments on Keyano have shown that the algorithm searches an average of 1.10 successors at a type-2 node. Furthermore,

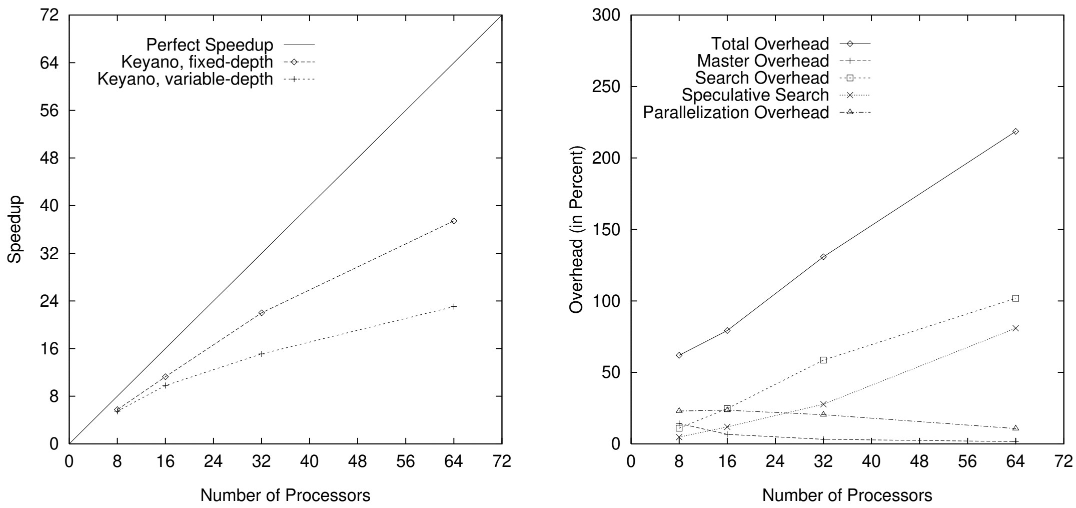


Figure 6.2: Speedups and Overheads for Keyano (Variable-Depth, Shared Memory)

the best move is searched first at any node within the game tree 89%-93% of the time [14]. These numbers are equivalent to the move ordering available in modern chess and checkers programs. Thus, this is not the reason behind the large observed speedups.

The real explanation for the observed speedups is that Keyano's game trees are true fixed-depth game trees. The subtrees that are examined by the slaves are roughly the same size, and very little load balancing is necessary in the fixed-depth case to keep all processors occupied on the current variation. To add the equivalent unbalancing that quiescence or capture search causes, we can use a version of Keyano with ProbCut enabled. This allows us to search 19-ply variable-depth game trees, instead of 15-ply fixed-depth game trees.

Figure 6.2 gives the speedups and overheads for the variable-depth shared memory version of Keyano. Note that the speedup curve for the fixed-depth shared memory version of Keyano is given on the left-hand graph as a reference point. The data for the variable-depth shared memory version of Keyano is also given in Table 6.2.

As we can see, the algorithm does not perform as well when we consider a variable-depth version of Keyano. The main differences between the fixed-depth and variable-depth algorithms can be summarized into three points: (1) the search overhead is much higher for the variable-depth searches, (2) the parallelization overhead is

n	Speedup	Total Time	Total Overhead	Master Overhead	Search Overhead	Speculative Search	Parallelization Overhead
1	-	29458 s	-	-	-	-	-
8	5.48	5738 s	61.93%	14.28%	10.98%	4.74%	23.02%
16	9.40	3149 s	79.27%	6.67%	24.68%	11.85%	23.55%
32	15.11	2081 s	130.87%	3.22%	58.60%	27.74%	20.43%
64	23.05	1326 s	218.61%	1.59%	101.86%	80.93%	10.7%

Table 6.2: Speedup Data for Keyano (Variable-Depth, Shared Memory)

marginally higher for the variable-depth searches, (3) the search size is significantly smaller for the variable-depth searches.

It is not surprising that the search overhead increases as we change the fixed-depth tree to a variable-depth tree. The load balancing associated with the APHID algorithm increase dramatically with variable search depths. Hence, the search overhead and speculative search are much higher in the variable-depth tests.

The parallelization overhead is marginally higher than the equivalent overhead for a fixed-depth search. The reason for this is that the APHID algorithm is spending more time handling smaller pieces of work in the variable-depth case than in the fixed-depth case.

Finally, we note that the average sequential search size is 1500 seconds for the variable-depth test, and over 2600 seconds for the fixed-depth case. Does this allow for greater parallelism in the algorithm? Schaeffer illustrated that the performance of the DPVS algorithm (a synchronous algorithm) relied on the depth of search [81]. However, the assertion that the size of the search is important to the overall speedup in asynchronous algorithms (such as APHID) should be confirmed experimentally.

It is easy to generate data concerning the depth of search and the speedup in Keyano, because all of the positions are searched to the same depth. Figure 6.3 gives the observed speedups for the fixed-depth shared memory version of Keyano as we increase the search depth from 12 to 15 ply. The overheads and speedups used to generate this figure are also given in Table 6.3.

As we can see, there are some interesting features to the observed speedups. We see that for a fixed number of processors, as the search depth increases, the speedup also

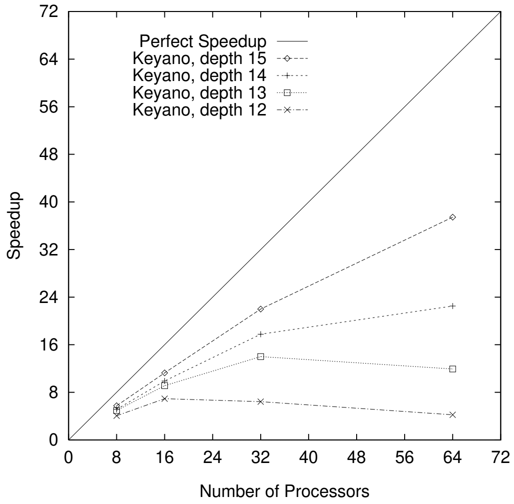


Figure 6.3: Speedups by Depth of Search for KEYANO (Fixed-Depth, Shared Memory)

	Depth 12		Depth 13		Depth 14		Depth 15	
n	Speedup	Total Time	Speedup	Total Time	Speedup	Total Time	Speedup	Total Time
1	-	1461 s	-	5620 s	-	14757 s	-	53526 s
8	4.07	360 s	4.95	1137 s	5.14	2919 s	5.74	9424 s
16	6.93	223 s	9.12	623 s	9.91	1521 s	11.27	4753 s
32	6.44	233 s	14.00	402 s	17.76	830 s	21.99	2415 s
64	4.23	402 s	11.92	513 s	22.51	699 s	37.44	1395 s

Table 6.3: Speedup Data by Depth of Search for KEYANO (Fixed-Depth, Shared Memory)

increases. This confirms our earlier hypothesis that the search size is an important factor in determining the observed speedup in APHID. If we search deeper game trees than the ones we examined, we would be able to achieve higher speedups. However, these deeper game trees would not be able to run on 64 processors using real time constraints. The reader must be very careful when comparing observed speedups without considering the search depth.

However, search size is not the only factor determining the observed speedup. If we examine the 14-ply fixed-depth shared memory test, and compare it to the 19-ply variable-depth shared memory test, we find the speedup curves to be similar. However, the fixed-depth search is half of the size of the variable-depth test on a single processor.

Another interesting feature of Table 6.3 is that speedups can decrease as APHID

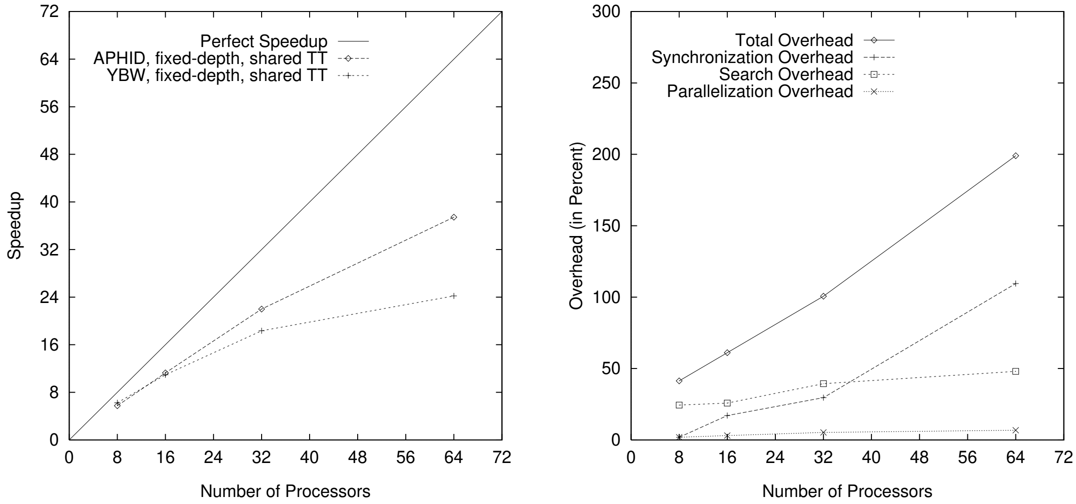


Figure 6.4: Speedups and Overheads for KEYANO (Young Brothers Wait, Fixed-Depth, Shared Memory)

uses more processors. This is possible if we require a greater number of pieces of work to be created by the master before allowing parallelism to start. For large parallel configurations, it may take a significant amount of time for APHID to catch up to a smaller parallel configuration. For example, it takes a 14-ply fixed-depth search for the n=64 processor configuration to finally catch up and overtake the n=32 processor configuration.

As a last data point for KEYANO, we would like to compare the APHID algorithm to the synchronous algorithm that generates the best results in the game of chess, Young Brothers Wait Concept. The version of YBWC implemented in KEYANO was originally coded in 1994 and taken directly from Feldmann's thesis. The author spent a considerable amount of time attempting to optimize the performance of the algorithm.

We tested YBWC and APHID under similar conditions. We used the same test set, searched to the same depth, and used the same size of shared memory transposition table for both parallel algorithms. The performance of APHID versus YBWC can be seen in Figure 6.4, with the raw data for the YBWC experiments to be found in Table 6.4.

The synchronous nature of the YBWC algorithm forces us to change the raw data

n	Speedup	Total Time	Total Overhead	Search Overhead	Parallelization Overhead	Synchronization Overhead	Time Spent Idle
1	-	53526 s	-	-	-	-	-
8	6.24	10828 s	41.37%	24.39%	1.9%	1.9%	(1.16%)
16	10.95	6122 s	61.12%	25.84%	3.1%	17.1%	(9.29%)
32	18.35	3046 s	100.63%	39.41%	5.3%	29.6%	(16.73%)
64	24.21	2217 s	199.05%	48.03%	6.8%	109.5%	(40.15%)

Table 6.4: Speedup Data for Keyano (Young Brothers Wait, Fixed-Depth, Shared Memory)

format given in Table 6.4. The two new columns are the Synchronization Overhead column and the Average Time Spent Idle column. The latter column represents the percentage of the time that each processor in the system was idle on average. These results can be converted into the numbers seen in the Synchronization Overhead column. As we can see, YBWC is rapidly starved out of work to do once we reach 64 processors: the processors are only busy 59.85% of the time.

The search overhead in YBWC starts at a higher level than APHID's search overhead on 8 processors, but the search overhead in APHID rapidly increases past the equivalent level in YBWC. However, it is important to note that the overhead in APHID is partially due to speculative search. When we run APHID on 64 processors, a 32.65% overhead is attributed to speculative search. If we intended to search the positions to 16 ply, these speculative search results would give APHID a head start on the next iteration. YBWC would be forced to attempt the entire search from scratch.

Earlier tests with YBWC in Keyano had a parallelization overhead of over 20% due to the iterative method presented in Feldmann's original description of the algorithm. However, once we have proceeded past the minimum granularity, we can use the recursive $\alpha\beta$ routine. This reduces YBWC's parallelization overhead to the levels seen in Table 6.4. These overheads are much lower than the equivalent overheads in APHID. However, APHID never slows down to wait for messages from another processor or to find work.

To test whether the implementation of YBWC is equivalent to previously published results, we can examine Weill's tests of YBWC and ABDADA on a CM-5 using

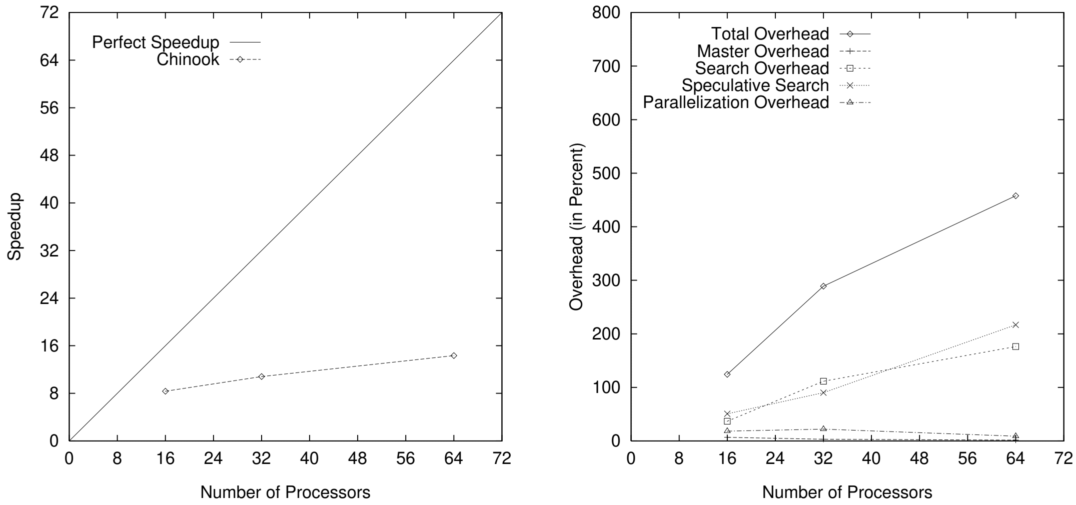


Figure 6.5: Speedups and Overheads for CHINOOK (Fixed-Depth, Shared Memory)

a different Othello program [95]. YBWC achieved a 9.5-fold speedup and ABDADA achieved a 11-fold speedup on 16 processors. Thus, the results generated by YBWC in Keyano are in line with other published results for the algorithm.

6.3.2 CHINOOK - Checkers

CHINOOK is the current Man-Machine World Champion checkers program. It was written by a team that includes Martin Bryant, Rob Lake, Paul Lu, Jonathan Schaeffer and Norman Treloar. The program is currently the top-rated checkers playing entity in the world, with a rating 180 points higher than the Human World Champion, Ron King. CHINOOK is currently retired from tournament play, but can be played via the Chinook home page on the World Wide Web[2].

The 20 positions in Appendix A.2 come from the 1992 Tinsley-Chinook Test Suite [57]. The positions were examined to depths varying from 23 to 29 ply, depending on the individual position. Figure 6.5 gives the speedups and overheads for the fixed-depth shared memory version of CHINOOK. The speedup and overhead data is also given in Table 6.5.

On 64 processors, the parallel searches take 226 seconds on average. This is larger

[2] `http://www.cs.ualberta.ca/~chinook`

n	Speedup	Total Time	Total Overhead	Master Overhead	Search Overhead	Speculative Search	Parallelization Overhead
1	-	58557 s	-	-	-	-	-
16	8.35	8042 s	124.26%	6.67%	36.7%	50.75%	18.4%
32	10.82	5943 s	289.08%	3.22%	111.4%	90.16%	22.1%
64	14.35	4529 s	457.89%	1.59%	176.3%	216.8%	8.9%

Table 6.5: Speedup Data for CHINOOK (Fixed-Depth, Shared Memory)

than the 120 second searches that we set out to achieve. The search depths for the experiment were chosen based on the 16 processor results and the likelihood that the results would scale well up to 64 processors. Unfortunately, that was not the case for CHINOOK. The limit of 120 seconds per move may be slightly higher if we consider captures, because captures are forced.

In comparison to the observed speedups for KEYANO, the speedups for CHINOOK could be described as disappointing. However, other authors that have attempted to parallelize checkers programs with synchronous algorithms have met with limited success. The best-known speedup for a synchronous algorithm on a high-performance checkers program is 3.32 for Lu's Principal Variation Frontier Splitting with Load Balancing [57]. Thus, the observed speedup of 14.35 on 64 processors for APHID is a four-fold improvement over previously published results for any synchronous algorithm in the domain of checkers.

It is important to note that comparing the observed speedups in this manner is not entirely fair. It is very likely that the synchronous algorithm would yield better observed speedups if it was tested on the SGI Origin 2000. The search size that CHINOOK is capable of reaching in 226 seconds on a 64-processor SGI Origin 2000 is significantly larger than the search depths achievable on the BBN Butterfly. Using the same test suite, Lu's algorithm was only able to search to a maximum depth of 17 to 21 ply with CHINOOK.

One factor that may be limiting APHID's observed speedup in CHINOOK in comparison to variable-depth KEYANO is that the root is handled in a different manner in CHINOOK. Instead of allowing APHID to control the entire search of the tree, CHINOOK forces APHID to search only the best move first in an attempt to determine the minimax value for the best move. Then, once that is complete, CHINOOK allows APHID to return to the root of the tree and verify that there are no better moves than the best move. This is an impediment to APHID's attempts to schedule both parts of work at the same time, and it does affect APHID's performance.

Another factor that limits APHID's observed speedup in CHINOOK is the size of the transposition table used. For each search, CHINOOK searches between 200 and 1300 million nodes sequentially, and we attempt to store the results in 16 million transposition table entries. It is clear that this is insufficient for some of the larger sequential searches. APHID searches approximately five times as many nodes as the sequential program when total search overhead is taken into account. Thus, if sequential CHINOOK does not have enough transposition table entries, then this problem must be more serious in the APHID version of CHINOOK.

In Table 6.5, we see that the parallelization overhead is close to the profile of the parallelization overhead for variable-depth searches in KEYANO. This is expected since both programs place similar requirements on the load balancing algorithm.

We can observe that the total search overhead is the dominant factor in the total overhead in both KEYANO and CHINOOK. However, in the case of CHINOOK, we see more speculative search than search overhead on $n = 16$ and $n = 64$ processors.

One reason for the large amount of speculative search is the special handling of the root of the game tree imposed by CHINOOK's source code. A second reason is that the forced capture rule in checkers yields subtrees with wide variations in size. In comparison to Othello and chess programs, this makes load balancing more challenging in CHINOOK.

A third reason for this phenomenon is the fact that the algorithm for subdividing large pieces of work is not as aggressive in CHINOOK as in KEYANO. If we take the exemption threshold as a power of the branching factor, we note that it would take over 6 ply of additional search (when the branching factor is 3) for CHINOOK to subdivide a piece of work. The exemption threshold for CHINOOK's fixed-depth shared memory test was 34 times the average piece of work, which is larger than

$3^{(6/2)} = 27$. For KEYANO (exemption threshold of 75, branching factor of 10), it takes at most the equivalent of 4 additional ply of search for a piece of work to be subdivided.

6.3.3 CRAFTY and THETURK - Chess

We have seen how APHID can yield larger observed speedups in domains with low branching factors such as checkers and Othello. However, the real test is the two chess programs: CRAFTY and THETURK. Here, synchronous algorithms achieve higher efficiencies because of the increased number of alternatives available at split nodes. It will be difficult to exhibit better performance than a synchronous game-tree search algorithm on a chess program.

THETURK is a chess program written by Yngvi Björnsson and Andreas Junghanns, two Ph.D. students at the University of Alberta. The program competed at the 1996 World Micro Computer Chess Championship. Although the program did not fare well in that tournament, the program has improved substantially since that tournament. The program is available to be played against on the World Wide Web[3]

CRAFTY is Robert Hyatt's freeware chess program. Robert Hyatt was the principal author of CRAY BLITZ, the top ranked chess-playing program in the early 1980s. He has parlayed his years of experience in writing chess programs into CRAFTY. CRAFTY is believed to be the strongest freely-available chess program. Unlike many other freeware chess programs, CRAFTY's code is legible and cleanly written. Because of this, the author was able to add APHID into CRAFTY without Robert Hyatt's assistance. The version of CRAFTY used in these experiments is the "Jakarta" version which played at the 1996 World Micro Computer Chess Championship.

The test set used is the Bratko-Kopec test set [48], the most popular test set used for benchmarking parallel chess programs. The positions from the test set can be found in Appendix A.1. Both CRAFTY and THETURK attempt to search the

[3] `http://www.cs.ualberta.ca/~games/TheTurk/index.cgi`

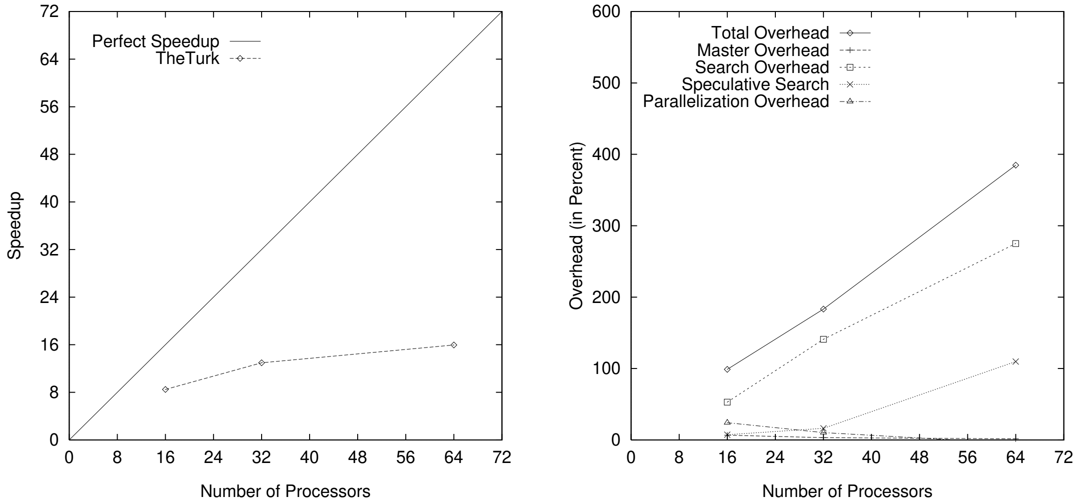


Figure 6.6: Speedups and Overheads for TheTurk (Fixed-Depth, Shared Memory)

n	Speedup	Total Time	Total Overhead	Master Overhead	Search Overhead	Speculative Search	Parallelization Overhead
1	-	139658 s	-	-	-	-	-
16	8.48	17609 s	98.88%	6.67%	52.91%	7.35%	24.2%
32	12.96	13046 s	183.29%	3.22%	140.94%	16.26%	10.3%
64	15.96	9888 s	384.78%	1.59%	275.04%	109.74%	-4.8%

Table 6.6: Speedup Data for TheTurk (Fixed-Depth, Shared Memory)

positions to depths of 11 or 12 ply.

Some of the positions are much smaller than the average search in the test set. The first position is a mate-in-3 that is solved in less than one second by both programs. Positions 6 and 8 are searched very quickly by modern programs because of the limited mobility and many transpositions available in the endgame positions. Thus, positions 1, 6 and 8 have been dropped from the averages, and the results given below are for the remaining 21 positions.

Figure 6.6 and Table 6.6 give the observed speedups and overheads for the fixed-depth shared memory version of TheTurk. Figure 6.7 and Table 6.7 give similar data for Crafty.

The total time taken by TheTurk to search the 21 positions on 64 processors does not accurately reflect the time control in the game of chess. On average, TheTurk's searches take 470 seconds each. Crafty's searches take 230 seconds per position.

Thus, the searches for both programs are marginally larger than those that we can expect to see in real-time constraints of 180 seconds per move. The search depths were chosen based on the observed speedups in the 16-processor results.

The observed speedups for TheTurk and Crafty are marginally better than the results for Chinook, but lower than the observed speedups for Keyano. This is surprising to researchers who are familiar with synchronous parallel game-tree search algorithms. It is important to remember that an asynchronous algorithm does not rely on spawning parallelism from a specific node. In APHID, we subdivide the tree at a level and make pieces of work out of all nodes at that level. Thus, the performance of APHID is not linked to the branching factor in the game tree being searched.

We hypothesized earlier that Chinook did not yield good speedups because the transposition table is barely adequate for the sequential program. By the same reasoning, we can state that the transposition table size limit is barely adequate for the sequential versions of Crafty and TheTurk. Although each program had 16 million transposition table entries, the chess programs searched between 300 million and 900 million nodes sequentially. We expect that a larger shared-memory transposition table will yield marginally faster sequential times and much faster parallel times due to reduced search overhead.

How do the speedups for the chess programs compare to results achieved by synchronous parallel search algorithms in the game of chess? As a representative synchronous parallel algorithm, let us examine Weill's tests that were run on a 32-processor Connection Machine 5. Weill's implementation of Young Brothers Wait achieves speedups of 12 on 32 processors, and his implementation of ABDADA achieves an observed speedup of 16 on 32 processors [95]. Weill's results are similar to other reported speedups with synchronous game-tree search algorithms. The majority of these synchronous algorithms have been tested under similar conditions to those used for the chess programs in this thesis (i.e. no search extension/reduction algorithms). Although it is not an objective comparison between algorithms, the observed speedups in the literature for synchronous algorithms are comparable, up to 32 processors, with the observed speedups seen for the APHID algorithm in both TheTurk and Crafty.

However, there is one important data point that is significantly larger than the reported speedups by any other author. Feldmann's Young Brothers Wait algorithm achieves speedups of 21.83 on 32 processors and 37.34 on 64 processors for 7-ply searches that take place in tournament time [29]. Results of this calibre have not been reported for any other algorithm or by any other author.

Why are Feldmann's observed speedups so large? Feldmann's results are based on a very slow program, Zugzwang. Zugzwang visited 523 nodes per second on a single processor, and only reaches 7-ply in 190 seconds on 32 Transputers. In comparison, Crafty is searching approximately 130,000 nodes per second on a SGI Origin 2000 processor, and accomplishes 11- and 12-ply searches in approximately 260 seconds using 32 processors. There is a dramatic speed difference between the Transputer and the 195 MHz MIPS R10000 used in the SGI Origin 2000, but that does not completely account for the 248-fold difference in search speed.

Zugzwang also spends an inordinate amount of time doing move ordering at each node. While the node is setting up some tables for move ordering, the process is able to send a message to another Transputer and receives a transposition table entry in return before Zugzwang is ready to read the message! No other chess program that the author knows of spends that amount of time assembling move ordering statistics before embarking on a search of a node. However, the effort does pay off: Zugzwang's move ordering is very strong. Instead of searching the best move first 96% of the time, Zugzwang could be written the way other chess programs are written. By using less expensive move ordering heuristics, the node rate will increase dramatically and will easily subsume the additional nodes that the program must search because of the poorer move ordering.

Another reason for the extraordinary speedups is that the transposition table was allowed to grow as more Transputers were added to the system. Thus, the observed speedups are illustrating the power of adding additional processors and

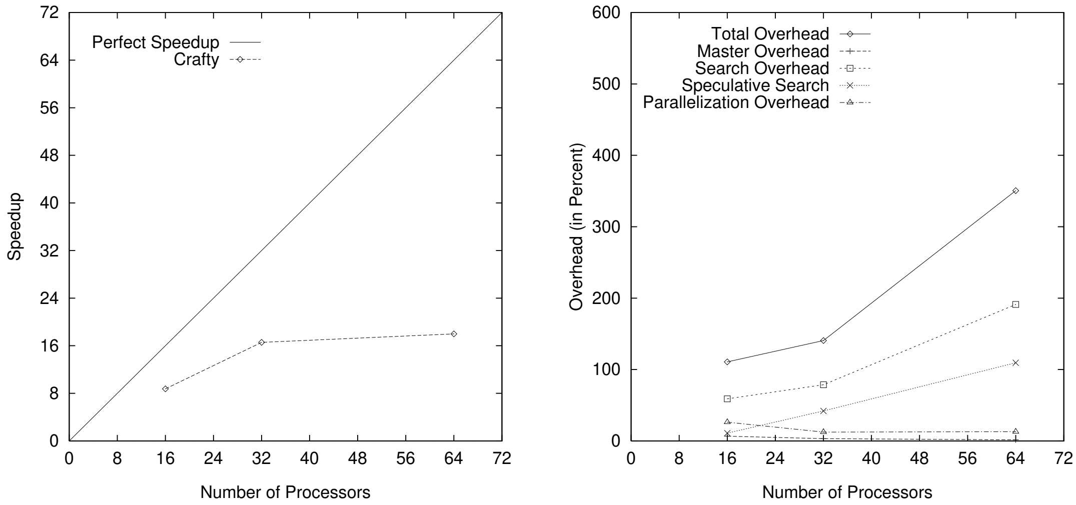


Figure 6.7: Speedups and Overheads for CRAFTY (Fixed-Depth, Shared Memory)

n	Speedup	Total Time	Total Overhead	Master Overhead	Search Overhead	Speculative Search	Parallelization Overhead
1	-	71596 s	-	-	-	-	-
16	8.76	9441 s	110.71%	6.67%	59.00%	10.90%	26.3%
32	16.56	5230 s	140.66%	3.22%	78.69%	42.01%	12.4%
64	18.00	4819 s	350.55%	1.59%	191.15%	109.50%	12.9%

Table 6.7: Speedup Data for CRAFTY (Fixed-Depth, Shared Memory)

transposition table memory, not just additional processors. Examining the 8-ply results, we see that three of the four largest searches for ZUGZWANG search less nodes in parallel than the associated sequential run. For the largest search (position 4 in the Bratko-Kopec test set), ZUGZWANG searches 61 million nodes sequentially with a small transposition table, and 41 million nodes in parallel on 256 processors with a much larger transposition table. It is likely that the number of nodes ZUGZWANG searches sequentially would decrease if much larger transposition tables were used during the sequential test.

In short, ZUGZWANG is a very slow sequential program, operating in an environment where messages are very fast in relation to the program, with superb move ordering and with memory usage that increases with the number of processors being tested. These four characteristics allow a synchronous algorithm to perform extremely well. With a fast sequential program, operating in an environment where messages are not fast in relation to the program, less than superb move ordering and constant memory usage in the sequential and parallel runs, it is unlikely that Feldmann's observed speedups can be duplicated with any synchronous parallel algorithm.

If we have a program without quiescence search, it is possible to generate equivalent speedups with asynchronous search. We have already demonstrated that APHID can generate speedups of that calibre in the fixed-depth version of KEYANO. However, quiescence search is integral to modern chess programs. Hence, it is unlikely that any asynchronous parallel algorithm would achieve observed speedups similar to Feldmann's results.

The parallelization overheads for CRAFTY and THETURK are very similar to the parallelization overhead exhibited in CHINOOK. This reflects the fact that APHID and the load-balancing algorithm (in particular) are doing approximately the same amount of work to distribute the work amongst all of the processors in all three programs.

A surprising observation from Table 6.6 is that the 64-processor numbers have a negative parallelization overhead. In other words, the slaves during the 64-processor runs searched more nodes per second than the sequential runs in THETURK. As mentioned earlier, we removed the transposition table lookups at the last ply of search to prevent the shared transposition table from slowing down the program. THETURK is the only program that also uses the transposition table during the quiescence search. The search algorithm does not know when the quiescence search will terminate. Thus, no transposition table accesses were allowed during the quiescence search in 64-processor tests with THETURK. The transposition table lookups slow down the quiescence search considerably on an SGI Origin 2000. Hence, the parallel program can run 4.8% faster than the sequential program.

In terms of performance, the search overhead is where the two chess programs differ. THETURK spends more time attempting to work on the current iteration, as reflected by the larger numbers in the Search Overhead column. This also indicates that more load balancing is necessary to get the work correctly distributed. The

	Local		Distributed		Shared	
Application	Speedup	Total Search Overhead	Speedup	Total Search Overhead	Speedup	Total Search Overhead
CHINOOK	4.68	257.05%	5.54	247.71%	8.35	87.37%
CRAFTY	5.70	188.53%	6.71	140.19%	8.76	69.90%
KEYANO - fixed	11.19	25.11%	11.31	22.25%	11.27	17.26%
KEYANO - variable	8.36	64.65%	8.52	56.27%	9.80	36.53%
THETURK	5.65	164.44%	6.59	123.15%	8.48	60.26%

Table 6.8: Speedups and Total Search Overheads for Various Transposition Table Configurations on APHID with 16 processors

search overhead in CRAFTY smoothly increases and are smaller than the equivalent numbers for THETURK.

The speculative search is at the same level in CRAFTY and THETURK. As the number of processors increases, both programs run out of work to balance at the end of each iteration. The equivalent speculative search lines indicate that both programs have similar distributions of subtree sizes on the slave processors at the end of the search.

6.4 Parallelism and the Structure of the Transposition Tables

We have made many claims on how the speedup is affected by the transposition table. In this section, we hope to present some numbers that will illuminate the nature of duplicate detection in the various applications.

Table 6.8 shows the speedups and total search overheads on 16-processor tests over various transposition table configurations. Because of the limited time available on the 64-processor SGI Origin 2000 configuration, it was not possible to complete all of the tests on 32 and 64 processors.

The *local* transposition tables in Table 6.8 indicate that each process maintains a separate transposition table, and no sharing between the tables is allowed. The *distributed* transposition table numbers represent the results when using the shadow transposition table algorithm described in Section 5.2.6. The *shared* transposition table uses a shared-memory transposition table that each process is able to read from and write to.

It is interesting to note that in Othello, varying the configuration of the transposition table makes very little difference. The same can not be said for the checkers and chess programs. For CHINOOK, there is a dramatic increase between using local transposition tables and a shared transposition table with the APHID algorithm. Although the difference is not as dramatic for the two chess programs (CRAFTY and THETURK), it is still worthwhile to consider using a distributed or shared transposition table.

6.5 Conclusions

We set out in the beginning of this chapter to show whether asynchronous algorithms could be competitive with synchronous algorithms in real applications searching real game trees. The APHID algorithm can be competitive with a synchronous algorithm when compared using observed speedup. If we also consider the additional benefit of speculative search on the key variations in the tree which is ignored by the observed speedup measure, the APHID algorithm must be heavily favoured over its synchronous counterpart.

APHID's results are independent of the branching factor within the tree. The synchronous search results, on the other hand, are very dependent on the branching factor; a larger branching factor yields more parallelism and less idle time. Thus, the asynchronous algorithm in APHID is a better choice in applications with small branching factors, such as checkers and Othello.

The observed speedups achievable by the APHID algorithm are similar to those achieved by synchronous parallel algorithms up to 32 processors for chess. Thus, a synchronous parallel algorithm may be a reasonable choice for wide variable-depth game trees. When APHID's ease of integration into existing legacy code is considered, the likelihood of better observed speedups with more transposition table memory, and the additional benefit of speculative search on some of the key variations at the next search depth, we feel that APHID is worth investigating in all game-tree search programs.

Once the author gets further access to an SGI Origin 2000, an interesting experiment that should be run is an examination of the effect of the transposition table size on both the sequential time and the parallel time in Chinook, Crafty and TheTurk. The author believes that the observed speedups will improve for all three programs if a larger shared-memory transposition table is used for both the sequential and the parallel tests.

Chapter 7

Conclusions and Future Work

The question we wished to answer in this document was: In the area of game-tree search, can asynchronous parallel algorithms outperform synchronous parallel algorithms? This document shows that asynchronous game-tree search algorithms can match or outperform the best synchronous game-tree search algorithms.

A similar question was posed by Feldmann in the English translation of his thesis [29]. If t is the search depth attained by the search algorithm, Feldmann's Open Problem 1 states:

> By the iterative deepening in a game tree, $O(t^2)$ synchronization nodes are generated, which decrease the processor workload. Is it possible to decrease the number of synchronization nodes without increasing the search overhead?

APHID reduces the number of synchronization points to a single point at the end of the search. Thus, this thesis demonstrates that it is possible to remove the synchronization nodes without dramatically increasing the search overhead.

7.1 Conclusions

In Chapter 3, the parallel game-tree search algorithms that have been published over the last 25 years are classified into a taxonomy. The taxonomy isolated the algorithmic properties from the implementation details. This separation allowed us to determine that synchronous game-tree search algorithms have been widely studied in the literature.

This concentration of research on synchronous parallel game-tree search algorithms has led many of the researchers in the same direction. A number of the modern synchronous parallel game-tree search algorithms are, in fact, remarkably similar to one another. When the implementation details are removed from the underlying parallel algorithm, it is clear that numerous researchers have re-invented the same algorithm. This fact has been obscured by the different hardware and software implementations used.

If synchronous game-tree search algorithms are over-studied, the opposite could be said for asynchronous game-tree search algorithms. Aside from Newborn's Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search (UIDPABS) and this document, no other research has focused on asynchronous parallel game-tree search algorithms that partition the game tree.

In Chapter 4, we compared a synchronous game-tree search algorithm to an asynchronous game-tree search algorithm in a theoretical framework. Using game trees that are designed to be similar to those searched by real game-playing programs, the asynchronous algorithm was shown to generate a greater parallel efficiency than the synchronous algorithm.

In Chapter 5, the Asynchronous Parallel Hierarchical Iterative Deepening (APHID) algorithm was introduced. APHID is based on extending Newborn's original work on the UIDPABS algorithm. UIDPABS only allowed for the tree to be split at the root node, while APHID allows the algorithm to be split at an arbitrary level within the game tree.

A game-independent parallel search library containing the APHID algorithm has been implemented and described in this document. This library can be inserted into a game-playing program written in the C programming language. This makes the APHID algorithm very easy to port among different applications and different

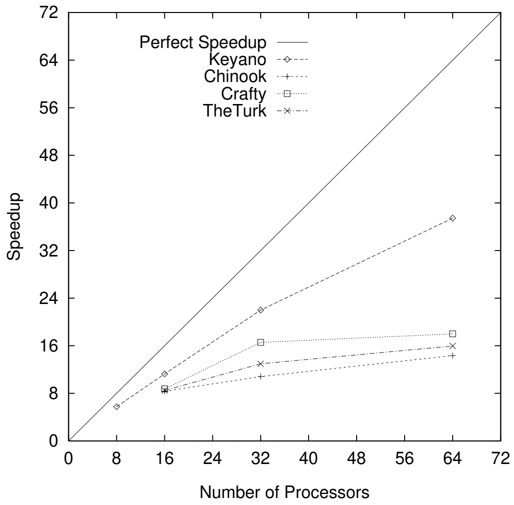


Figure 7.1: Speedups for All Programs (Fixed-Depth, Shared Memory)

	Keyano		Chinook		Crafty		TheTurk	
n	Speedup	Total Search Over.	Speedup	Total Search Over.	Speedup	Total Search Over.	Speedup	Total Search Over.
8	5.74	9.12%	-	-	-	-	-	-
16	11.27	17.26%	8.35	87.45%	8.76	69.9%	8.48	59.26%
32	21.99	28.78%	10.82	201.6%	16.56	120.7%	12.96	157.2%
64	37.44	76.72%	14.35	393.1%	18.00	300.65%	15.96	384.8%

Table 7.1: Speedup Data for All Programs (Fixed-Depth, Shared Memory)

hardware configurations.

In Chapter 6, we explored the observed speedups that can be achieved with the current version of the APHID library. In Figure 7.1 and Table 7.1, a concise summary of the tests is presented.

Synchronous game-tree search algorithms have a performance that is strongly correlated to the depth of search attempted and the branching factor in the game tree. The asynchronous algorithm in APHID does not depend on the branching factor. Instead, APHID's speedup is determined by the depth of search and the variability in the size of the subtrees examined by the slave processors.

In terms of observed speedup, the Othello program Keyano gives us the largest numbers. The reason for this is that Keyano searches a real fixed-depth game tree. Thus, little load balancing is required to maintain an average work load on each processor. When we compared APHID to an optimized version of YBWC, the observed speedup for the KEYANO/APHID combination is about 50% larger than the observed speedup for Young Brothers Wait on 64 processors. Although the search overhead was smaller in Young Brothers Wait, the processors were busy searching the game tree only 60% of the time.

For the other programs, we compared APHID's results against the observed speedups in the literature. The speedups for APHID in the checkers program CHINOOK may be considered as disappointing. However, the observed speedup is still four times larger than previously published results for a highly-tuned synchronous algorithm in CHINOOK. The observed speedups for the chess programs (THETURK and CRAFTY) are in line with the majority of previously published results for chess programs containing synchronous parallel game-tree search algorithms.

In all four of the programs, we have additional speculative search which is ignored by the observed speedup numbers. In a real tournament game, not only would we see the observed speedup when we measure search depth, but the key variations would sometimes be searched an additional ply or two deeper than they would be in the synchronous case.

In short, the speedups have been shown to be competitive or significantly stronger than synchronous methods on common game-tree searching applications. Thus, we have shown both that our original question can be answered affirmatively in both a theoretical framework and in practice.

7.2 Future Work

Although a lot of development effort has gone into the second version of the APHID library, there are a number of things that remain to be tested or implemented.

- **Library Usability**

  The first and foremost issue to be determined is the usability of the library by other researchers. The author has been responsible for the majority of the development work on the library over the last two years, and the code has not undergone beta testing by the general game-playing community.

- **Speculative Search and Move Quality**

  An interesting addition to the research on asynchronous parallel game-tree search would be quantifying the additional benefit that the speculative search can yield to a game-playing program. Although the benefits may be difficult to quantify in some of the programs given in this thesis, it would be interesting to run a test that measures APHID's ability to speed up a program's selection of a winning variation in a tactical position. By comparing a synchronous algorithm's ability on the same test set, we could quantify the difference that speculative search makes in terms of move quality.

  Another experiment that could be run is a match between a sequential program with APHID versus the same sequential program with a synchronous parallel algorithm. Using a varied number of random and equal openings, we could determine whether the APHID version can routinely defeat a synchronous parallel algorithm.

- **Aspiration Windows**

  APHID does not perform well on sequential search algorithms that manipulate aspiration windows at the root of the tree, such as the MTD family of algorithms. The current version of APHID can determine whether the algorithm will fail high or fail low for a search window at the root of the game tree. As each slave processor finishes its work on the current iteration, it attempts to start the next search depth via iterative deepening. If APHID was enhanced to understand whether a re-search at the same depth was likely, a new sub-iteration could be attempted before proceeding to the next iteration of iterative deepening. This would likely increase the search overhead and reduce the speculative search component of the total search overhead.

- **Determining Search Windows on Slave Processors**

  APHID's current method for predicting the $\alpha\beta$ search window on the slaves was chosen after numerous experiments. There may be better algorithms for determining the search window on slave processors.

  Another important point to be made is that the variables in the current algorithm were hand-coded by the author to achieve the best performance on each application. A fair amount of testing went into determining the constants for each application, and there may be better ways to automatically determine these constants within APHID. For example, default values for the search window sizes could be used and optimized over a series of runs by a mechanism internal to the APHID library.

- **Hierarchies**

  Although the hierarchies are available within the APHID library, the code for implementing the hierarchies has not been performance tuned. Due to limited availability of the SGI Origin 2000 systems for testing, it is difficult to obtain the time to analyze the issues surrounding the performance of hierarchies in APHID.

  It would also be nice to be able to demonstrate a situation where the hierarchy is necessary for APHID's performance. One probable scenario where this could be demonstrated is a tree search distributed over different computers on the Internet. The hierarchy could be used to ensure the high-level communication between the sites is relatively small. The intra-site communications can be organized so that the majority of the messages stay on the same site.

- **Additional Algorithms in APHID Framework**

  The APHID library has many custom attachments for efficiently executing game-tree search. The library could be generalized to encompass other tree searching algorithms, such as IDA*.

An interesting line of research is parallelism in the B* game-tree search algorithm. There are obvious avenues for parallelizing the B* search algorithm, such as executing many probe searches at the same time, or executing probe searches on more than one node concurrently. It is not clear whether synchronous or asynchronous search methods will be more successful in parallelizing the B* search algorithm.

It is hoped that this document, in conjunction with the APHID library, will be helpful for game-tree researchers to investigate these and other ideas surrounding asynchronous game-tree search.

Bibliography

[1] G. M. Adelson-Velsky, V. L. Arlazarov, and M. V. Donskoy. Some Methods of Controlling the Tree Search in Chess Programs. *Artificial Intelligence*, 6(4):361–371, 1975. **(36)**

[2] S. G. Akl, D. T. Barnard, and R. J. Doran. Design, Analysis and Implementation of a Parallel Tree Search Algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(2):192–203, 1982. **(47,49,51)**

[3] I. Althöfer. A Parallel Game Tree Search Algorithm with a Linear Speedup. *Journal of Algorithms*, 15:175–198, 1993. **(63)**

[4] T. Anantharaman, M. S. Campbell, and F.-h. Hsu. Singular Extensions: Adding Selectivity to Brute-Force Searching. *Artificial Intelligence*, 43(1):99–109, 1990. **(35)**

[5] H. E. Bal and R. van Renesse. A Summary of Parallel Alpha-Beta Search Results. *ICCA Journal*, 9(3):146–149, September 1986. **(45)**

[6] G. M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1978. Available as Tech. Rept. CMU-CS-78-116. **(47,49,50)**

[7] E. B. Baum, C. Garrett, W. D. Smith, and R. Tudor. Best Play for Imperfect Players and Game Tree Search; Part II - Experiments. Technical report, NEC Research Institute, Princeton, NJ, April 1995. **(39)**

[8] E. B. Baum and W. D. Smith. Best Play for Imperfect Players and Game Tree Search; Part I - Theory. Technical report, NEC Research Institute, Princeton, NJ, April 1995. **(39)**

[9] D. F. Beal. A Generalized Quiescence Search Algorithm. *Artificial Intelligence*, 43(1):85–98, 1990. **(36)**

[10] H. J. Berliner. Some Necessary Conditions for a Master Chess Program. In *Proceedings of IJCAI-73*, pages 77–85, Stanford, CA, 1973. **(35)**

[11] H. J. Berliner. The B* Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, 12:23–40, 1979. **(38)**

[12] H. J. Berliner and C. McConnell. B* Probability Based Search. *Artificial Intelligence*, 86:97–156, 1996. **(39)**

[13] M. G. Brockington. A Taxonomy of Parallel Game-Tree Search Algorithms. *ICCA Journal*, 19(3):162–174, 1996. **(8)**

[14] M. G. Brockington. Keyano Unplugged – The Construction of an Othello Program. Technical Report 97-05, University of Alberta, Department of Computing Science, Edmonton, Canada, June 1997. Presented at the "Game Tree Search in the Past, Present and in the Future" Workshop, NEC Research Institute, Princeton, NJ, Aug. 1997. **(144,146)**

[15] M. G. Brockington and J. Schaeffer. The APHID Parallel $\alpha\beta$ Search Algorithm. In *Proceedings of IEEE SPDP '96*, pages 428–432, New Orleans, Louisiana, October 1996. **(8,101,136,140)**

[16] M. G. Brockington and J. Schaeffer. APHID Game Tree Search. In H.J. van den Herik and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 69–91. Universiteit Maastricht, 1997. **(8,101,136,140)**

[17] A. Broder, A. Karlin, P. Raghavan, and E. Upfal. On the Parallel Complexity of Evaluating Game-Trees. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–413, January 1991. **(63,70)**

[18] A. L. Brudno. Bounds and Valuations for Abridging the Search for Estimates. *Problems of Cybernetics*, 10:225–241, 1963. Translation of Russian original in *Problemy Kibernetiki*, 10:141–150, May 1963. **(19)**

[19] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt. The ZRAM Parallel Search Bench and its Applications. *Annals of Operation Research*, 1997. To appear.
URL: `http://nobi.ethz.ch/ambros/aor_zram.ps.gz` **(68)**

[20] M. Buro. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal*, 18(2):71–76, 1995. **(36)**

[21] M. S. Campbell. Algorithms for the Parallel Search of Game Trees. Master's thesis, University of Alberta, Department of Computing Science, Edmonton, Canada, 1981. Available as Tech. Rep. TR 81-8. **(37,52,61)**

[22] M. S. Campbell and T. A. Marsland. A Comparison of Minimax Tree Search Algorithms. *Artificial Intelligence*, 20:347–367, 1983. **(61)**

[23] P. Ciancarini. Distributed Searches: A Basis for Comparison. *ICCA Journal*, 17(4):194–206, 1994. **(45)**

[24] V.-D. Cung. *Contribution à l'Algorithmique Non Numérique Parallèle: Exploration d'Espaces de Recherche.* PhD thesis, Université Paris VI, April 1994. In French. **(47,49,58)**

[25] V. David. *Algorithmique Parallèle sur les Arbres de Décision et Raisonnement en Temps Contraint - Etude et Application au Minimax.* PhD thesis, ENSAE, Toulouse, France, 1993. In French. **(47,49,58)**

[26] C. G. Diderich. 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, July 1992. In French. **(61)**

[27] C. Donninger. Null Move and Deep Search: Selective Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, 16(3):137–143, 1993. **(36)**

[28] C. Ebeling. *All the Right Moves: A VLSI Architecture for Chess.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1986. Also 1987 book, MIT Press, Cambridge, MA. **(33)**

[29] R. Feldmann. *Spielbaumsuche auf Massiv Parallelen Systemen.* PhD thesis, University of Paderborn, Paderborn, Germany, May 1993. English translation available: Game Tree Search on Massively Parallel Systems.
URL: `ftp://ftp.uni-paderborn.de/doc/techreports/Informatik/misc/phdFeldmann.ps.Z` **(30,46,47,49,55,75,143,144,157,163)**

[30] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed Game Tree Search. *ICCA Journal*, 12(2):65–73, 1989. **(55)**

[31] E. W. Felten and S. W. Otto. Chess on a Hypercube. In G. Fox, editor, *Proceedings of The Third Conference on Hypercube Concurrent Computers and Applications*, volume II-Applications, pages 1329–1341, Pasadena, CA, 1988. **(47,49,54)**

[32] C. Ferguson and R. E. Korf. Distributed Tree Search and its Application to Alpha-Beta Pruning. In *Proceedings of AAAI-88*, pages 128–132, Saint Paul, MN, August 1988. **(47,49,56,75)**

[33] R. A. Finkel and J. P. Fishburn. Parallel Alpha-Beta Search on Arachne. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 235–243, 1980. Also Tech. Rep. 394, University of Wisconsin, Madison WI. **(25)**

[34] R. A. Finkel and J. P. Fishburn. Parallelism in Alpha-Beta Search. *Artificial Intelligence*, 19(1):89–106, 1982. **(25,47,49,51)**

[35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. **(6,68,97)**

[36] J. J. Gillogly. Performance Analysis of the Technology Chess Program. Technical Report 189, Carnegie Mellon University, Pittsburgh, PA, 1978. **(71)**

[37] G. Goetsch and M. S. Campbell. Experiments with the Null-Move Heuristic. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 55–78. Springer-Verlag, 1990. Earlier version appear in 1988 AAAI Spring Symposium Proceedings, pages 14-18. **(36)**

[38] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker. The Greenblatt Chess Program. In *Proceedings of the Fall Joint Computer Conference*, volume 31, pages 801–810, 1967. **(27)**

[39] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994. **(68)**

[40] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968. **(64)**

[41] F.-h. Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1990. Also Tech. Rept. CMU-CS-90-108, Carnegie Mellon University, Feb. 1990. **(47,49,56,76,99)**

[42] S. Huang and L. R. Davis. Parallel Iterative A* Search: An Admissible Distributed Search Algorithm. In *Proceedings of IJCAI-89*, pages 23–29, Detroit, MI, August 1989. **(65)**

[43] R. M. Hyatt. *A High-Performance Parallel Algorithm To Search Depth-First Game Trees.* PhD thesis, University of Alabama, Birmingham, Alabama, 1988. **(54,55)**

[44] R. M. Hyatt. The Dynamic Tree Splitting Parallel Search Algorithm. *ICCA Journal*, 20(1):3–19, 1997. **(47,49,56,75)**

[45] R. M. Hyatt, B. W. Suter, and H. L. Nelson. A Parallel Alpha/Beta Tree Searching Algorithm. *Parallel Computing*, 10(3):299–308, 1989. **(47,49,54)**

[46] R. M. Karp and Y. Zhang. On Parallel Evaluation of Game Trees. In *Proceedings of SPAA '89*, pages 409–420, New York, NY, 1989. ACM Press. **(63,70)**

[47] D. E. Knuth and R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(3):293–326, 1975. **(19,20,24,76)**

[48] D. Kopec and I. Bratko. The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 57–72. Permagon Press, 1982. **(154,178)**

[49] R. E. Korf. Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985. **(65,66)**

[50] H.-J. Kraas. *Zur Parallelisierung des SSS*-Algorithmus.* PhD thesis, TU of Braunschweig, Braunschweig, Germany, January 1990. In German. **(61)**

[51] V. Kumar and L. N. Kanal. Parallel Branch-and-Bound Formulations for AND/OR Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):768–778, November 1984. **(38)**

[52] V. Kumar, K. Ramesh, and V. N. Rao. Parallel Best-First Search of State-Space Graphs: A Summary of Results. In *Proceedings of AAAI-88*, pages 122–127, Saint Paul, MN, August 1988. **(65)**

[53] B. C. Kuszmaul. *Synchronized MIMD Computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1994. **(47,49,59,76)**

[54] D. B. Leifker and L. N. Kanal. A Hybrid SSS*/Alpha-Beta Algorithm for Parallel Search of Game Trees. In *Proceedings of IJCAI-85*, pages 1044–1046, 1985. **(61)**

[55] G. Lindstrom. The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm. Technical Report UUCS 83-101, University of Utah, Department of Computer Science, Salt Lake City, UT, March 1983. **(47,49,52)**

[56] U. Lorenz and V. Rottmann. Controlled Conspiracy Number Search, September 1995. Diplomarbeit, University of Paderborn, Paderborn, Germany. **(39,62)**

[57] C.-P. P. Lu. Parallel Search of Narrow Game Trees. Master's thesis, University of Alberta, Department of Computing Science, Edmonton, Canada, 1993. **(43,47,49,57,71,137,141,151,152,180)**

[58] A. Mahanti and C. Daniels. A SIMD Approach to Parallel Heuristic Search. *Artificial Intelligence*, 60:243–282, 1993. **(66)**

[59] T. A. Marsland. Relative Performance of the Alpha-Beta Algorithm. *ICCA Journal*, 5(2):21–24, 1982. **(72)**

[60] T. A. Marsland. Relative Efficiency of Alpha-Beta Implementations. In *Proceedings of IJCAI-83*, pages 763–766, Karlsruhe, Germany, 1983. **(30)**

[61] T. A. Marsland and M. S. Campbell. Parallel Search of Strongly Ordered Game Trees. *ACM Computing Surveys*, 14(4):533–551, 1982. **(47,52)**

[62] T. A. Marsland and Y. Gao. Speculative Parallelism Improves Search? Technical Report 95–05, University of Alberta, Department of Computing Science, Edmonton, Canada, April 1995. **(47,49,60)**

[63] T. A. Marsland, M. Olafsson, and J. Schaeffer. Multiprocessor Tree-Search Experiments. In D. Beal, editor, *Advances in Computer Chess 4*, pages 37–51. Permagon Press, 1986. **(49,52,141)**

[64] T. A. Marsland and F. Popowich. Parallel Game-Tree Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, 1985. **(52)**

[65] T. A. Marsland, A. Reinefeld, and J. Schaeffer. Low Overhead Alternatives to SSS*. *Artificial Intelligence*, 31:185–199, 1987. **(38)**

[66] D. A. McAllester. Conspiracy Numbers for Min-Max Searching. *Artificial Intelligence*, 35:287–310, 1988. **(39)**

[67] J. McCarthy. Chess as the Drosophilia of AI. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 227–237. Springer-Verlag, 1990. Also "The Fruitfly on the Fly", *ICCA Journal*, vol. 12, no. 4, pp. 199-206. **(1)**

[68] M. M. Newborn. A Parallel Search Chess Program. In *Proceedings of the ACM Annual Conference*, pages 272–277, 1985. **(52)**

[69] M. M. Newborn. Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-10(5):687–694, 1988. **(4,47,49,53,85)**

[70] A. Newell, J. C. Shaw, and H. A. Simon. Chess Playing Programs and the Problem of Complexity. *IBM Journal of Research and Development*, pages 320–335, Oct 1958. Reprinted in *Computers and Thought* (eds. E.A. Feigenbaum and J. Feldman), pages 39–70. McGraw-Hill, New York, 1963. **(19)**

[71] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Book Company, New York, NY, 1971. **(64)**

[72] A. J. Palay. *Searching With Probabilities.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1983. Also published by Pitman, Boston, MA, 1985. **(36,39)**

[73] J. Pearl. Asymptotic Properties of Minimax Trees and Game-Searching Procedures. *Artificial Intelligence*, 14:113–138, 1980. **(30)**

[74] A. Plaat. *Research Re: Search & Re-search.* PhD thesis, Erasmus University, Dept. of Computer Science, Rotterdam, The Netherlands, 1996. **(23,34,62,71)**

[75] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting Graph Properties of Game Trees. In *Proceedings of AAAI '96*, volume 1, pages 234–239, Portland, Oregon, August 1996. **(32,38,43)**

[76] C. Powley, C. Ferguson, and R. Korf. Depth-First Heuristic Search on a SIMD Machine. *Artificial Intelligence*, 60:199–242, 1993. **(66)**

[77] V. N. Rao, V. Kumar, and K. Ramesh. A Parallel Implementation of Iterative-Deepening-A*. In *Proceedings of AAAI-87*, pages 178–182, Seattle, Washington, July 1987. **(65)**

[78] A. Reinefeld. An Improvement to the Scout Tree-Search Algorithm. *ICCA Journal*, 6(4):4–14, 1983. **(30)**

[79] A. Reinefeld. A Minimax Algorithm Faster than Alpha-Beta. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 237–250. University of Limburg, 1994. **(38)**

[80] A. Reinefeld and V. Schnecke. AIDA* – Asynchronous Parallel IDA*. In *Proceedings of 10th Canadian Conference on Artificial Intelligence (AI'94)*, pages 295–302, Banff, Canada, 1994. **(66)**

[81] J. Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6(2):90–114, 1989. **(47,49,53,141,147)**

[82] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212, 1989. **(29)**

[83] J. Schaeffer. Conspiracy Numbers. *Artificial Intelligence*, 43(1):67–84, 1990. **(39)**

[84] J. J. Scott. A Chess-Playing Program. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 255–265. Edinburgh University Press, 1969. **(26)**

[85] R. Shinghal and S. Shved. Proposed Modifications to Parallel State Space Search of Game Trees. *International Journal of Pattern Recognition and Artificial Intelligence*, 5(5):809–833, 1991. **(61)**

[86] J. R. Slagle and J. K. Dixon. Experiments With Some Programs That Search Game Trees. *Journal of the ACM*, 16(2):189–207, April 1969. **(18,22,27)**

[87] D. J. Slate and L. R. Atkin. Chess 4.5 - The Northwestern University Chess Program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, New York, 1977. **(26,28)**

[88] I. R. Steinberg and M. Solomon. Searching Game Trees in Parallel. In *Procceedings of the 1990 International Conference on Parallel Processing,* vol. 3, pages 9–17. Penn. State University Press, 1990. **(62)**

[89] G. C. Stockman. A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence*, 12:179–196, 1979. **(37)**

[90] K. Thompson. Computer Chess Strength. In M.R.B. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Permagon Press, 1982. **(2)**

[91] S. Tschöke and T. Polzer. Portable Parallel Branch-and-Bound Library (PPBB-Lib) User Manual, Version 1.1, 1996.
URL: `http://www.uni-paderborn.de/~ppbb-lib` **(68)**

[92] H. Usui, M. Yamashita, M. Imai, and T. Ibaraki. Parallel Searches of Game Trees. *Systems and Computers in Japan*, 18(8):97–109, 1987. **(61)**

[93] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton Press, Princeton, U.S.A., 1944. **(10)**

[94] O. Vornberger and B. Monien. Parallel Alpha-Beta versus Parallel SSS*. In *Proceedings of the IFIP Conference on Distributed Processing*, pages 613–625. North Holland, October 1987. **(55,61)**

[95] J.-C. Weill. *Programmes d'Échecs de Championnat: Architecture Logicielle, Synthèse de Fonctions d'Évaluations, Parallélisme de Recherche*. PhD thesis, Université Paris 8, January 1995. In French. **(47,58,59,151,156)**

[96] J.-C. Weill. The ABDADA Distributed Minimax-Search Algorithm. *ICCA Journal*, 19(1):3–16, 1996. **(49,59,76)**

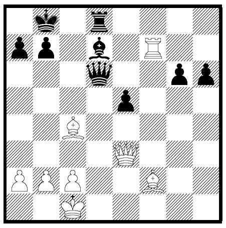
[97] A. L. Zobrist. A Hashing Method with Applications for Game Playing. Technical Report 88, University of Wisconsin, 1970. Reprinted in the ICCA Journal, 13(2):69–73, 1990. **(28)**

Appendix A

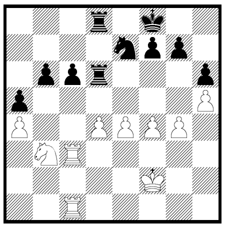
Test Positions

A.1 Chess – Crafty and TheTurk

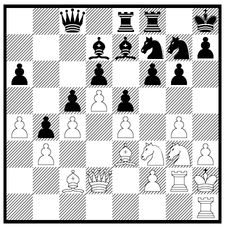
The positions used for the experiments in this document come from the Bratko-Kopec experiment [48].



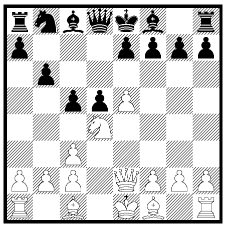
Position 1
Black To Move



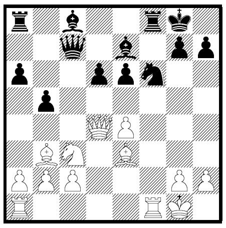
Position 2
White To Move



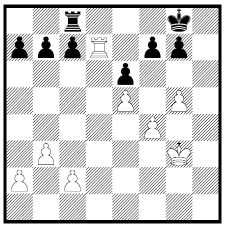
Position 3
Black To Move



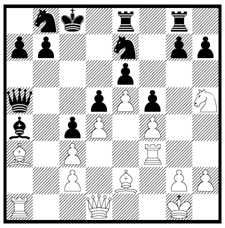
Position 4
White To Move



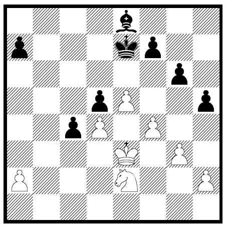
Position 5
White To Move



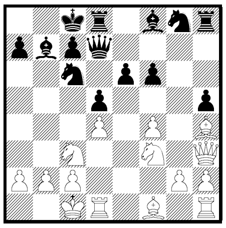
Position 6
White To Move



Position 7
White To Move

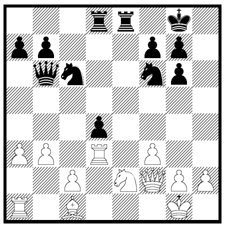


Position 8
White To Move



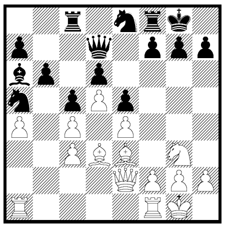
Position 9

White To Move



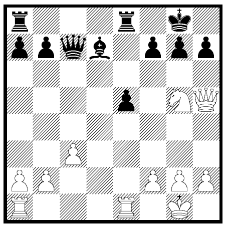
Position 10

Black To Move



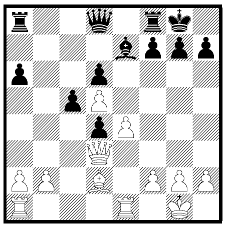
Position 11

White To Move



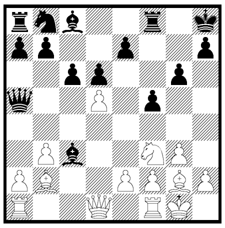
Position 12

Black To Move



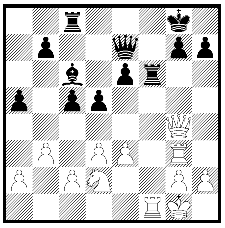
Position 13

White To Move



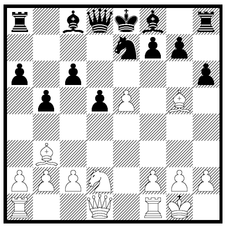
Position 14

White To Move



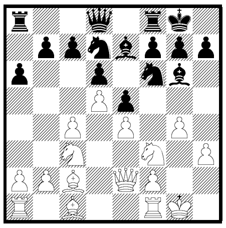
Position 15

White To Move



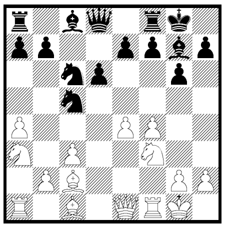
Position 16

White To Move



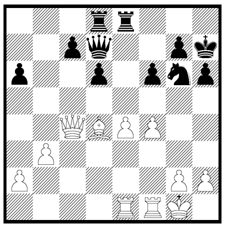
Position 17

Black To Move



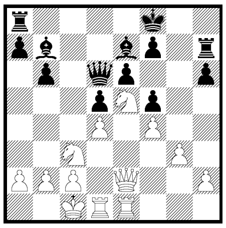
Position 18

Black To Move



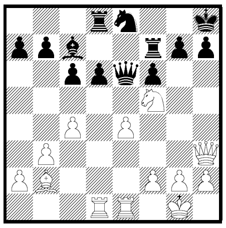
Position 19

Black To Move



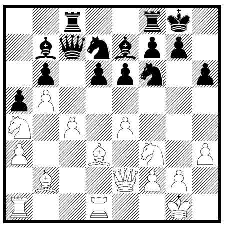
Position 20

White To Move



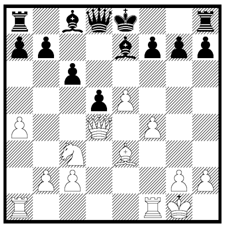
Position 21

White To Move



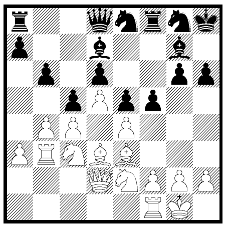
Position 22

Black To Move



Position 23

Black To Move

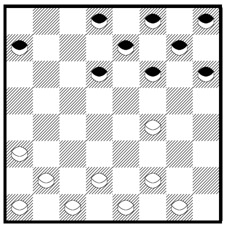


Position 24

White To Move

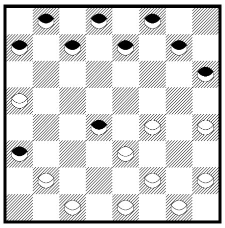
A.2 Checkers – Chinook

The positions used in the experiments are from the Tinsley-Chinook 1992 Test Suite [57].



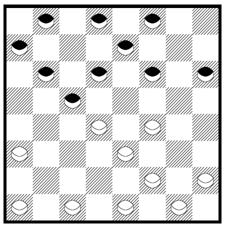
Position 1

Black To Move



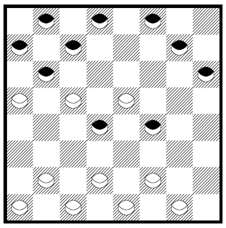
Position 2

Black To Move



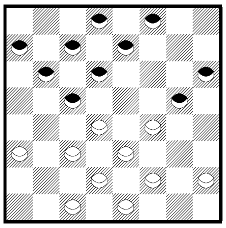
Position 3

White To Move



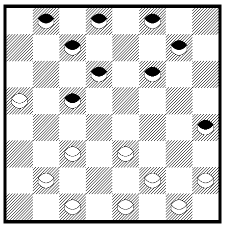
Position 4

Black To Move



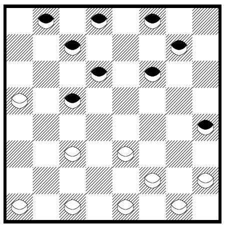
Position 5

Black To Move



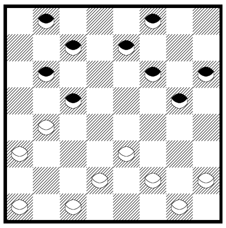
Position 6

Black To Move



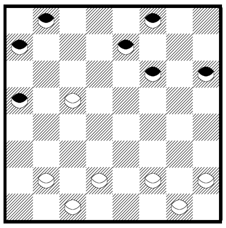
Position 7

White To Move



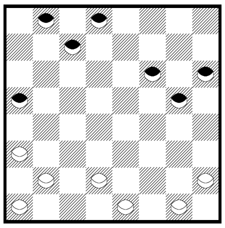
Position 8

Black To Move



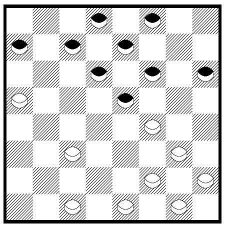
Position 9

Black To Move



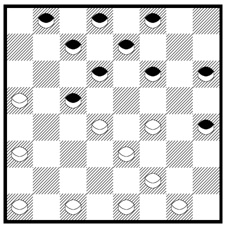
Position 10

White To Move



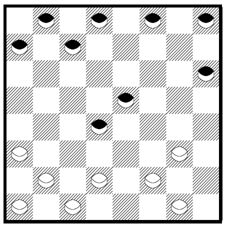
Position 11

Black To Move

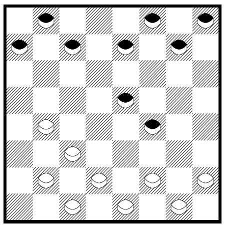


Position 12

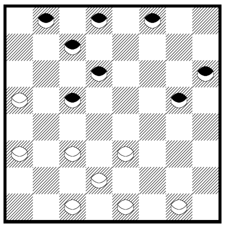
Black To Move



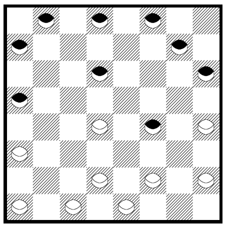
Position 13
White To Move



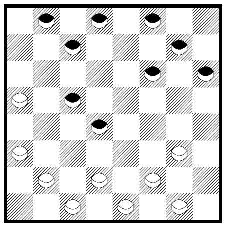
Position 14
Black To Move



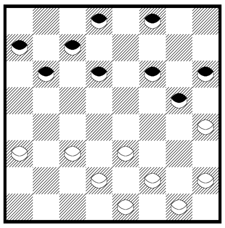
Position 15
White To Move



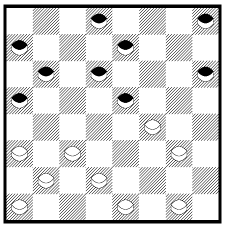
Position 16
Black To Move



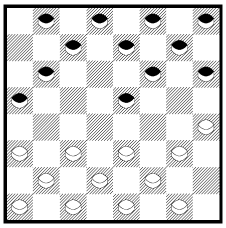
Position 17
Black To Move



Position 18
Black To Move



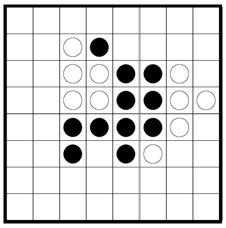
Position 19
Black To Move



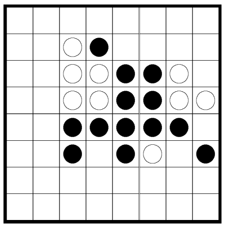
Position 20
White To Move

A.3 Othello – Keyano

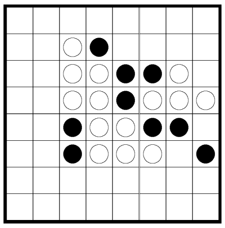
The positions used in the experiments for Keyano are taken from moves 18-27 of the two games in the 1994 World Championship final between Emmanuel Caspard and David Shaman.



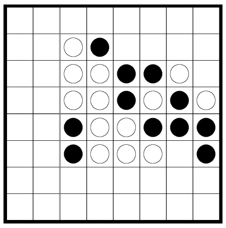
Position 1
White To Move



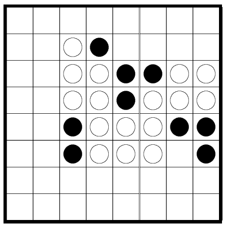
Position 2
Black To Move



Position 3
White To Move

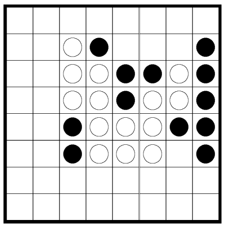


Position 4
Black To Move



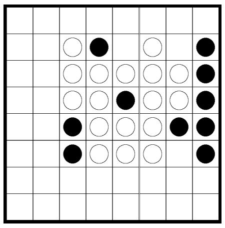
Position 5

White To Move



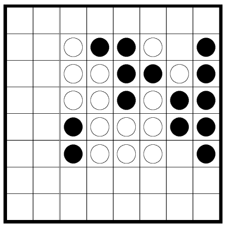
Position 6

Black To Move



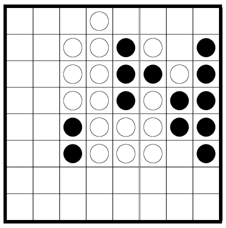
Position 7

White To Move



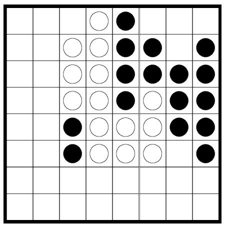
Position 8

Black To Move



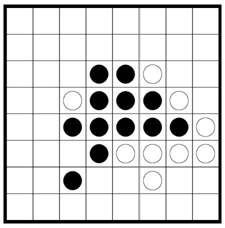
Position 9

White To Move



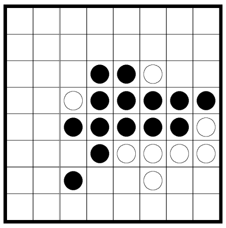
Position 10

Black To Move



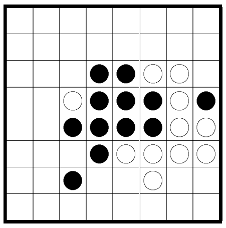
Position 11

White To Move



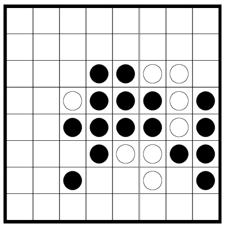
Position 12

Black To Move



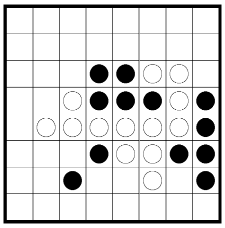
Position 13

White To Move



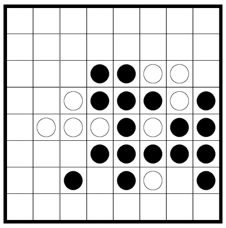
Position 14

Black To Move



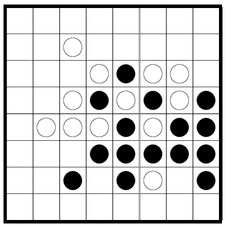
Position 15

White To Move



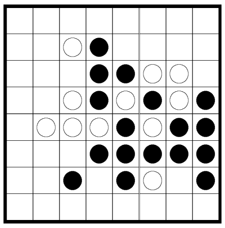
Position 16

Black To Move



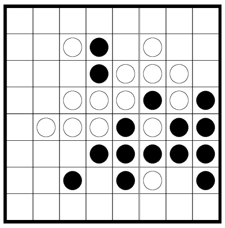
Position 17

White To Move



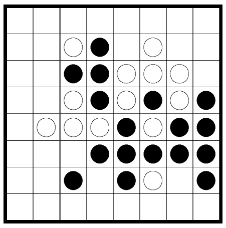
Position 18

Black To Move



Position 19

White To Move



Position 20

Black To Move

Appendix B

APHID's Interface with the Application

This appendix attempts to give a precise interface between the APHID library and the application. Since the APHID library is application-independent, some definitions of how things are implemented must be given to the library.

We will start by specifying the parameters used throughout the various sections in the appendix (Section B.1). Next, we will describe the application-dependent constants that are to be defined by the programmer in the `public.h` file (Section B.2. The `aphid_stub_` call-backs that APHID uses to retrieve or send application-dependent information are described in Section B.3. Finally, the various calls that are inserted into the existing application code are described. This list is broken down by calls that both master and slave processes use (Section B.4), followed by master-only functions (Section B.5) and slave-only functions (Section B.6).

B.1 Standard Parameters Used in APHID Interface

- `argv`: Standard argument list passed into the `main()` C routine. Used by APHID to instantiate the slaves with the same run-time parameters as the absolute master.

- depth: The number of ply the current node is away from the root of the game tree.
- plytogo: The number of ply until we reach the bottom of the game tree. In a search with no extensions or forward pruning, plytogo + depth should be constant.
- &move[i] or &bestmove: A pointer to an area of APHID_MOVESIZE bytes which specifies the move being played.
- p_hash: A pointer to an area of APHID_HASHTYPESIZE bytes that contains the current hash value.
- p_key: A pointer to an area APHID_HASHKEYSIZE bytes that contains a lock which can "guarantee" the board stored in the location of the hash table is correct.
- p_entry: A pointer to an area of APHID_TRANSENTRYSIZE bytes that contains the hash value just written into the local transposition table.
- alpha and beta: Search window used by $\alpha\beta$ implementation.
- value and score: Minimax values of the sub-trees of a node and the node itself, respectively.

B.2 Application-Dependent Constants

- APHID_HASHTYPESIZE: The size (in bytes) of the hash value used in the application. Typically, this will either be 4 or 8 bytes, depending on whether your hash value is 32 or 64 bits in length. Even if only 20 bits are commonly used for the hash value, the full length of the random number generated should be passed in.

- `APHID_HASHKEYSIZE`: The size (in bytes) of the key used to guarantee that two positions that hash to the same location are the same. In some cases, this may be the same as the hash value. However, some applications use the complete board representation as the key within the transposition table.

- `APHID_TRANSENTRYSIZE`: The size (in bytes) of each transposition table entry. This is required so that the program can copy the correct number of bytes for use by the distributed transposition table code in APHID.

- `APHID_MOVESIZE`: The size (in bytes) of the representation of a move in the application.

- `APHID_MINUSINF` and `APHID_PLUSINF`: The minimum and maximum possible values returned by the application's evaluation function.

- `APHID_INVALIDSCORE`: A value that is outside the range represented by the specified minimum and maximum evaluation function values.

- `APHID_LOG2_TABSIZE`: The size of the APHID table that you intend on using to share between the master and the slaves, taken to a base 2 logarithm. For example, a value of 14 indicates an APHID table with $2^{14} = 16384$ entries.

- `APHID_MAXSLAVEPLYSEARCH`: The maximum nominal search depth that you expect to hand to a slave to search. Once this depth is reached, the slaves stop searching the work granules.

- `APHID_MAXMASTERPLYSEARCH`: The maximum depth that we expect the master should reach. Note that this includes the hierarchy of all masters and any exemptions that may be applied.

- `APHID_NODESPERSECOND`: The number of leaf and internal tree nodes that the application usually visits within one second. The number will be used to determine an approximate measure of time while a game-tree search is taking place.

B.3 Call-Back Functions

- `int aphid_stub_encodeinit(char *msg)`: Provides the absolute master process with a 4000-byte buffer (pointed to by `msg`) to store all pertinent information about the root of the game tree, such as the position and the game history (if this is relevant to the search algorithm). The return value is the number of bytes written into the string `msg`.

- `int aphid_stub_decodeinit(int msg_ln, char *msg)`: All other processes in the system, aside from the absolute master, receive the message length and the message encoded by `aphid_stub_encodeinit`, and should use the information to prepare to search the game tree. A process should be in the same game state as the absolute master after the routine is finished.

- `void aphid_stub_movedownpath(int num_moves, char *movepath)`: Called by the slaves, a series of `num_moves` moves is given in `movepath`, with each move being `APHID_MOVESIZE` bytes long. The routine should play through the moves given in `movepath`, starting at the position at the root of the game tree.

- `void aphid_stub_moveuppath(int num_moves, char *movepath)`: Called by the slaves, this routine should completely undo any changes made when moving down the move path. After this routine is finished, the game state should be the same as it was after the end of the `aphid_stub_decodeinit` call.

- `int aphid_stub_iterativedeepening(int depth, int last, int max)`: Called by the slaves, this routine should return the search depth for the subsequent search of a leaf node. The parameters specify the current depth of the leaf node, the depth that the last search was completed to, and the maximum search depth that can be assigned, respectively. For most programs that do iterative deepening in 1-ply steps, this routine should simply return `last + 1`. For programs that do iterative deepening in steps of 2, `depth` and `last` can be used to ensure that the the value returned, when added to `depth`, has the correct parity.

- `void aphid_stub_preparesearch(int depth, int plytogo,`
`int winstats[], int *alpha, int *beta)`: Called by the slaves, this routine should set the initial window searched and place it in the integers pointed to by `alpha` and `beta`.

  To assist in the decision, the `winstats` array contains numerous statistics, such as whether a bad bound search has been signalled, the search window used at the root of the game tree, and the likelihood that the minimax value will not change. The statistics in the current array are:

  – `[0]`: A boolean value that represents whether the search window contained in the next two entries must be used (if the entry equals 1), or is simply the window used at the root of the game tree (if the entry equals 0).

  – `[1]`: Contains α from a search window (see entry `[0]`).

  – `[2]`: Contains β from a search window (see entry `[0]`).

  – `[3]`: Current guessed minimax value of the root of the game tree. Should be used as the center of the window.

  – `[4]`: Boolean value that indicates if we are doing a speculative search.

  – `[5]`: Contains the depth adjustment to the `plytogo` parameter. It may be needed by some programs to disallow null-moves and/or ProbCut.

  – `[6]`: An indicator to determine whether a null window search should be centered to the left $(mmx - 1, mmx)$ or to the right $(mmx, mmx + 1)$. If the value returned here is even, the window should lean to the right. Otherwise, it should lean to the left.

  – `[7]`: A count of how many left-branches of the current PV must be made certain before the current guessed minimax value is known. The window should be made marginally wider for each left-branch given in this array value.

  - `[8]`: An estimation of the distance between this node and the PV node. If this value equals the current depth, the node we are examining is the PV node and should be searched with the full search window provided in entries `[1]` and `[2]`.
  - `[9]`: Reserved for future purposes.

  It must be emphasized that this routine is critical to the performance of the APHID algorithm, and all of the variables returned are used. Making windows that are too wide or too small seriously affects the performance of the APHID algorithm.

- `void aphid_stub_clear_alarm()`: This routine should clear the global alarm information in the application. It will be called before each search by both the master and the slave processes.

- `int aphid_stub_alphabeta(int depth, int plytogo, int alpha, int beta)`: Called by both the master and the slaves, this routine should call your implementation of $\alpha\beta$, and return the minimax value back to the APHID library.

- `int aphid_stub_evaluate(int depth, int alpha, int beta)`: Called by the master the first time it visits a leaf of its tree, this routine should simply call your evaluation routine and return the score for the position reached at `depth` ply within the tree.

- `int aphid_stub_stopsearch(int pass_stats[])`: Called only by the absolute master process, this routine should check your timer and determine if your time limit has been exceeded for a search. If the time limit has been exceeded, this routine should return 1; otherwise, 0.

  To make the decision to terminate a search more robust, a number of statistics are passed in to this routine via the `pass_stats` array. The entries contains the following information.

- [0]: The number of uncertain leaf nodes touched in the latest pass.
- [1]: The total number of leaf nodes touched in the last pass.
- [2]: The lower bound on the minimax value at the root of the tree being examined in parallel.
- [3]: The guessed minimax value at the root of the tree being examined in parallel.
- [4]: The upper bound on the minimax value at the root of the tree being examined in parallel.
- [5]: Reserved for future use.

- `int aphid_stub_visited()`: Called by the slaves, this should return a global count of the number of nodes visited by the process.

- `int aphid_stub_insert_local_tt(int hash, char *tt_ptr)`: Called by slave processes, this routine should attempt to store the complete transposition table entry pointed to by `tt_ptr` into the transposition table location given by `hash`. The values come from information passed into a peer slave's `aphid_distt_insertentry` routine (Section B.6).

B.4 Interface Calls Used by Masters and Slaves

- `void aphid_startup(argv)`: In the first process run, PVM is spawned on the machines specified, and a slave process is spawned with the same argument list `(argv)` as the master, as specified by the `aphid.config` file. The APHID library never exits this function call when a slave process executes it.

- `void aphid_exit()`: This routine removes a process from the PVM group, and it should be called before any process exits (due to errors or normal completion). If the process is the absolute master, completion of this routine ensures that all of the spawned processes have been shut down successfully.

- `int aphid_master()`: Returns 1 if the process is a master in the hierarchy, and 0 otherwise.

- `int aphid_slave()`: Returns 1 if the process was spawned, returns 0 if it is the absolute master process which spawned the other processes. Note that a process can be both a slave and a master depending on the hierarchy specified in `aphid.config`.

B.5 Interface Calls Used by Masters Only

- `void aphid_initserach(int maxdepth)`: Called by the absolute master process, this procedure prepares to start a search in parallel. This routine calls `aphid_stub_encodeinit`, and then informs all of the other processes of the current state of the game. The parameter indicates the maximum depth that any process is nominally allowed to search, not including search extensions.

- `int aphid_rootsearch(int depth, int plytogo, int alpha, int beta)`: This routine is called by the absolute master, instead of calling the typical $\alpha\beta$ implementation. It allows a master process to do multiple passes of the tree until the search is completed, or `aphid_stub_stopsearch` signals that the search should be terminated. If the search is allowed to complete, this routine returns the minimax value of the tree that it has been asked to search.

- `int aphid_intnode_premove(int depth, char *moveptr)`: Called by the absolute master, this routine stores moves made before `aphid_rootsearch` into the move list.

- `void aphid_endsearch()`: The absolute master should call this routine when it is finished searching a tree that has been called. The routine stops all slaves from working on the leaves of the game tree and prepares the slaves to receive a new root position.

- `int aphid_horizon(int depth, int plytogo, char *p_hash, char *p_key)`: A master process calls this routine to determine if it has reached its artificial horizon. If the routine returns a 1, we have reached the parallelization horizon. We must continue searching if it returns a 0. `depth` and `plytogo` are used to determine whether the piece of work is large enough and at the right level. `p_hash` and `p_key` are required to detect exempted nodes.

- `int aphid_eval_leaf(int alpha, int beta, int depth, int plytogo, char *p_hash, char *p_key)`: This routine returns a minimax value for the leaf, based on the best information available to the master. `alpha` and `beta` represent the current search window. `depth` and `plytogo` are used to determine the best information, and the hash value and key are used to look up the position within the APHID table.

- `void aphid_intnode_start(int depth, char *p_hash, char *p_key)`: Called by a master process, this routine initializes bound gathering information for an interior node within the game tree. `p_hash` and `p_key` are required to determine if this node has been previously visited.

- `void aphid_intnode_move(int depth, char *moveptr)`: Called by a master process, this routine inserts the move pointed to by `moveptr` into a hidden move list that will eventually be sent to a slave in `aphid_eval_leaf`.

- `void aphid_intnode_update(int depth, int value)`: Called by the master process, this routine takes the value returned by the child $\alpha\beta$ call and uses it to update the hidden "bound" information gathered for every node in the master's tree.

- `void aphid_intnode_end(int depth, int score, int beta)`: This routine is called by a master for every internal node within the tree, and ensures that the score returned is consistent with previously gathered information about the node. `beta`, the upper bound on the search window, is used to determine the correct bound information within the master's tree.

B.6 Interface Calls Used by Slaves Only

- `int aphid_checkalarm(int force_check)`: This routine checks to see if a search should be terminated. If the parameter `force_check` is zero, the PVM message queue will be checked a few times every second. If the parameter is non-zero, the PVM message queue will be checked immediately. It is highly recommended that a slave process call the routine with `force_check = 0`. If the value returned by `aphid_checkalarm` is equal to 0, the search should continue. If the value returned is not equal to 0, the current search has been interrupted, and we should terminate it in a "nice" way. 1 is returned if the absolute master has terminated the search, and 2 is returned if we are searching speculatively and should stop the current search. However, for the purposes of the application, they should be treated the same way.

- `void aphid_distt_insertentry(int hash, char *p_entry, int plytogo)`: This routine will store the transposition table location pointed to by `p_entry` into the shadow transposition table. `hash` is used to determine where the transposition table information should be placed on the peer slave processors, and `plytogo` is used to determine which pieces of work to save. The routine has no return value, since it is not relevant for the slave to know if the entry was stored in the shadow transposition table.