UNIVERSITY OF ALBERTA

Parallel Search of Narrow Game Trees

by

Chien-Ping Paul Lu

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

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta

Fall 1993

Abstract

One of the key determinants of a game playing program's strength is the depth of the game tree search. Therefore, researchers have turned to parallelism to search deeper trees in the same amount of real time. Tree decomposition algorithms extract parallelism by creating split nodes, where the subtrees rooted at the node are searched concurrently. If the game trees are *narrow*, then the degree of parallelism offered by the low branching factor may be insufficient to keep all the parallel processors busy, resulting in starvation and poor speedups.

Chinook is a checkers (8 x 8 draughts) playing program developed at the University of Alberta. For the August 1992, Man Versus Machine World Draughts Championship match between *Chinook* and Dr. Marion Tinsley, a parallel *Chinook,* or *ParaChinook*, was developed. This thesis describes the parallelization of *Chinook*'s Alpha-Beta search.

The initial implementation of *ParaChinook* was based on the Principal Variation Splitting (PVSplit) algorithm, developed by computer chess researchers. However, chess game trees have an average branching factor of 35 to 40, whereas checkers game trees have an average branching factor of 2.84, which is over an order of magnitude smaller. PVSplit exhibits high levels of starvation on the narrow checkers game trees, therefore Principal Variation Frontier Splitting (PVFSplit) is developed. PVFSplit attempts to create more parallel work by allowing multiple split nodes in a variation splitting framework. Although, PVFSplit improves the speedup by 52% (with 16 searchers) compared to PVSplit, the parallel performance is still low in absolute terms.

The parallel performance of both PVSplit and PVFSplit is evaluated using a test suite of 20 positions and running on a BBN TC2000 shared memory multiprocessor. The issues of starvation and straggler processes are discussed, quantified and addressed. The addition of load balancing code to deal with stragglers further improves the speedup. However, it is concluded that any PVSplit-based algorithm will suffer from poor speedups on the narrow checkers game trees. In light of this, some ideas for a future version of *ParaChinook* are discussed.

Acknowledgments

Thank you to Jonathan Schaeffer, my friend, teacher and supervisor.

Thank you to the other members of my thesis committee, Dr. Tony Marsland and Dr. Edo Nyland. Their patience and guidance is appreciated. Thanks are also due to Steve Sutphen and my friends and colleagues in the Department of Computing Science. Mark Brockington's Sneakernet was much appreciated.

Thank you to the Massively Parallel Computing Initiative (MPCI), directed by Dr. Eugene Brooks, at Lawrence Livermore National Laboratory for access to the BBN TC2000 parallel computer. I owe a particular debt of gratitude to Brent Gorda of MPCI. Much of the implementation of *ParaChinook* was completed on a Silicon Graphics 4D/480 multiprocessor. Thank you to Silicon Graphics International for the loan of the machine and for sponsoring the World Draughts Championship (1992) match between Dr. Marion Tinsley and *Chinook*, in London, England.

The financial support of the Department of Computing Science through a Teaching Assistantship is gratefully acknowledged. Thank you to the Natural Sciences and Engineering Research Council (NSERC) for a Postgraduate Scholarship. Thanks also to Petro-Canada, Inc. for a Graduate Research Award.

Thank you to the members of the human checkers community for their kindness during the U.S. National Tournament (1992) in Hot Springs, Arkansas, and during the Tinsley-*Chinook* Match (1992) in London, England. In particular, I would like to thank Herschel Smith for his support in Hot Springs. And thanks are due to Dr. Tinsley for his sportsmanship in agreeing to play *Chinook*.

A very big thank you to my friends and family.

As for musical inspiration, I've been known to talk at length about that when asked. :-)

Table of Contents

List of Tables

List of Figures

List of Formulas

Chapter 1

Introduction

Efficiently parallelizing the Alpha-Beta search algorithm is difficult in general [Sch89a], but it is made more difficult by *narrow game trees*, which have a low branching factor. This thesis examines the problem of parallelizing the game tree search of the *Chinook* checkers[1] (8 x 8 draughts) playing program [SCT92,STL93]. Two recurring themes are how the low branching factor of checkers game trees makes it difficult to create sufficient amounts of parallel work and how the branch-and-bound nature of Alpha-Beta search makes it difficult to complete the work efficiently in parallel.

Although pioneering work in artificial intelligence (AI) and machine learning by Arthur Samuel used the game of checkers as the application domain [Sam59,Sam67], researchers have largely ignored computers checkers for the past 34 years. One notable exception is the *Duke* checkers program [Tru78]. Of course, there have been commercially available checkers programs, but the technical lessons from those projects have not been widely published. In truth, much of the published work in computer game playing in the past three decades has been in computer chess.

Whereas chess game trees are wide and have the potential for a relatively high *degree of parallelism* [SuG91], checkers game trees are narrow and have less work available for parallel processing at each node. Chess game trees have a relatively high average branching factor of 35 to 40 [Gil78]. This thesis reports that checkers game trees in the proposed test suite have an average branching factor of 2.84, which is over an order of magnitude less than in chess. As parallel processors are added to the game tree search strategy, they quickly exceed the degree of parallelism available at a node in a narrow game tree.

The situation is further complicated by the fact that the Alpha-Beta tree search algorithm is a special case of the branch-and-bound family of algorithms. As with other branch-and bound

---

1 For a brief summary of the rules of checkers, see Appendix A.

algorithms, Alpha-Beta has the ability to logically eliminate portions of the search tree based on the results obtained by searching other subtrees. Therefore, searching subtrees in non-optimal order, through poor work scheduling or because searches are done in parallel without the benefit of information from an earlier search, may result in unnecessary computation.

1.1 A Brief History of *Chinook*

Chinook is a checkers program developed at the University of Alberta. The project began in June 1989 with Joe Culberson, Jonathan Schaeffer and Duane Szafron and over the years, a variety of individuals have become involved in the effort. The project has the short-term goal of developing an interactive program capable of defeating the human world checkers champion and the long-term goal of solving the game of checkers.

Towards the short-term goal, the program has had some significant successes. In August 1990, *Chinook* won the Mississippi State Open (human) checkers tournament and placed second in the U.S. National Open behind World Champion Dr. Marion Tinsley. In both tournaments, *Chinook* defeated human checkers grandmasters. More importantly, by placing second in the U.S. National Open, behind the World Champion, *Chinook* earned the right to play for the World Championship in a 40-game match. *Chinook* is the first computer program to earn the right to contest for such a title.

Chinook is not invincible and has also had some setbacks. Since August 1990, *Chinook* has lost one competitive game to another computer program, and it has narrowly lost exhibition matches against human grandmasters. In August 1992, Dr. Tinsley and *Chinook* played the 40-game match for the Man Versus Machine World Draughts Championship in London, England, sponsored by Silicon Graphics International. Dr. Tinsley defeated *Chinook* with 4 wins, 2 losses and 33 draws.[2] To *Chinook*'s credit, its two wins represent only the 8th and 9th losses by Dr. Tinsley in over 40 years of competitive play.

For the August 1992 match, a parallel version of *Chinook*, *ParaChinook*, was implemented for the Silicon Graphics SGI 4D/480, 8-processor, shared memory multiprocessor. Each processor had a clock speed of 33 MHz and the common random access memory was 256 megabytes. However, the results presented in this thesis were actually obtained on a BBN TC2000 shared memory multiprocessor. The TC2000 version of *ParaChinook* is the August

---

[2] The 40th game was not played because with a 2 game lead, Dr. Tinsley had already won the match.

1992 program, ported to the new computer to take advantage of the larger number of processors and with minor improvements added as a result of the on-going research.

The long-term goal of the *Chinook* project is to prove the game theoretic value of the game of checkers. From the initial board position, either one player can force a win or neither player can force a win. Proving the game would determine if it is a win for Black, a win for White or is a draw. There is strong empirical evidence that the game is a draw. The state space contains 5×10^{20} nodes [SCT92] and proving the game of checkers would set a milestone as one of the largest combinatorial search spaces to be solved.

An important component of the both the short and long term goals is the endgame databases computed through retrograde analysis [Tho86,LSL93]. Currently, *Chinook* has a database of all of the endgames with 7 pieces or less on the board, the so-called 7-piece databases, and the game theoretic value of each position. By looking up a position in the database, it is possible to determine the position's value with perfect accuracy. In late-July 1993, all of the materially balanced (i.e. equal number of pieces of each colour), and thus the most useful, 8-piece databases were completed. Because the endgame databases represent perfect information, they will play a crucial role in solving the game. The endgame databases have already played a major role in *Chinook*'s success in tournament play and in the August 1992 match with Dr. Tinsley.

The short-term goal remains unfulfilled, but *may* be close at hand as a re-match with Dr. Tinsley is possible in late 1993 or early 1994. The addition of the entire materially balanced 8-piece databases, of which only 40% was available in August 1992, will significantly improve *Chinook*'s playing strength. The long-term goal, which once seemed very distant, is also made more attainable by the completion of the key 8-piece databases.

1.2 Overview of the Thesis

This thesis describes the parallelization of *Chinook*'s Alpha-Beta search. The depth-first Principal Variation Splitting (PVSplit) algorithm [MaC82], popular with computer chess researchers, formed the basis of the first *ParaChinook*. However, the poor performance of PVSplit led to the development of the Principal Variation Frontier Splitting (PVFSplit) algorithm, which is a variation of PVSplit. The parallel performance of both PVSplit and PVFSplit is evaluated on a BBN TC2000 shared memory multiprocessor, using a test suite of 20 positions. The issues of starvation and stragglers are discussed, quantified and addressed. It is concluded that any PVSplit-based algorithm will suffer from poor speedups on the narrow checkers game trees. However, a hybrid approach using both a non-depth-first search algorithm and PVFSplit may be promising for a future version of *ParaChinook*.

The thesis begins by reviewing the concepts of sequential Alpha-Beta game tree search. It is a non-exhaustive survey of the large body of research in Alpha-Beta search and game playing programs. It then surveys the research on parallel Alpha-Beta search and discusses the *ParaChinook* implementation of Principal Variation Splitting and the experiments used to evaluate parallel performance. It discusses Principal Variation Frontier Splitting, which is based on PVSplit, but improves on its parallel performance. It also addresses the issue of stragglers and their impact on performance. Finally, the thesis concludes with a discussion of the experimental results and possibilities for future work.

Chapter 2

Game Tree Search

Computer programs play games, such as checkers and chess, by considering different combinations of possible moves for itself and possible replies by the opponent. Each legal combination of move and counter-move is a possible line of play. Ultimately, the most desirable line of play, often called the *principal variation* (PV), is decided upon. The systematic searching of different combinations is a well-known problem solving strategy and, in effect, the program is trying to look ahead at the possible continuations of the game. Intuitively, as the search is deepened, the amount of analysis increases and the chance of overlooking a strong move or a blunder decreases. Therefore, the depth of search is an important determinant of the quality of the program's play.

2.1 Game Trees and Minimax Search

To support the systematic analysis of different lines of play, the program builds a game tree. The *root* of the tree is the current position on the game board, a *node* in the tree is a legal board position and a *branch* is a legal move. The side which has the choice of move at the root position will be referred to as the *player* and the other side as the *opponent*.

The notion of *ply* describes the distance between nodes and is similar to the general notion of the *depth* of a tree. The immediate children of the root are said to be at ply 1, or are one ply away from the root. Although the ideal is to search lines of play until the end of the game, practical limitations on time and computing resources may limit the depth of search. *Leaf nodes* exist where the lines of play are not searched further and represent the program's look-ahead *horizon*. *Terminal nodes* are leaf nodes where the program is able to analyze to the end of the game, possibly with the aid of a database of positions of known values, such as an *endgame database* [SCT92].

In two player zero-sum games, such as checkers, an advantage for the player implies an equal and opposite disadvantage for the opponent. *Minimax search* models the give-and-take nature of these games. A heuristic *evaluation function* is used to assign a value to a leaf node that is not a terminal node. Positive numbers represent positions advantageous for the player, and negative numbers represent disadvantageous positions. From the point of view of the opponent, these node values are negated because the opponent is the adversary.

The interior nodes are assigned values through a minimizing-maximizing procedure that *backs up* the leaf values towards the root of the tree. It is assumed that the player always plays towards a leaf node with a high value, so an interior node of the player is assigned the arithmetic maximum value of its immediate child nodes. Similarly, the interior nodes of the opponent are assigned the minimum of its child nodes. Because the two sides alternate turns at moving, the interior nodes at even plies are always maximums and nodes at odd plies are always minimums. Note that the minimax value of the tree is a function of the leaf nodes and not of the root position itself. In fact, the principal variation leads to the leaf node that gives the root position its value. If different leaf nodes can have the same value, then there may be multiple principal variations, each backing up the same value to the root position.

The naive method of determining the minimax value of the root of the game tree is to visit all of the leaf nodes and apply the recursive backup procedure at all interior nodes. Clearly, this is a computationally expensive task as the depth d of the tree increases. In fact, the number of nodes increases exponentially with respect to the *branching factor*, the average number of branches from a node. Adapting the notation and terminology of previous authors (for example, [KnM75, MaC82]), the *Number of Bottom Positions* (NBP) [SlD69] or leaf nodes in the *full minimax tree* of uniform width w is:

$$\mathrm{M}_{\mathrm{NBP}}(w,d) = w^d$$

Formula 2.1 — Leaf Nodes in Full Minimax Tree
(M is for minimax, w is uniform width, d is fixed depth)

The total number of leaf and interior nodes in the full minimax tree is called the *Node Count* (NC) (terminology from [Sch86]) and is given by:

$$\mathrm{M}_{\mathrm{NC}}(w,d) = \sum_{i=1}^{d} w^i = \frac{w^{d+1}-1}{w-1}$$

Formula 2.2 — Node Count in Full Minimax Tree
(M is for minimax, w is uniform width, d is fixed depth)

Analytically, it is more convenient to consider only the leaf nodes as given by the NBP measure since the equations are simpler and the number of leaf nodes provides a lower bound on the number of nodes in the tree. Empirically, the NC measure is also useful.

Figure 2.1 shows an example of a full minimax tree of uniform width 2 and fixed depth 3. Note that, in practice, game trees are rarely of fixed depth and uniform width. However, Formulas 2.1 and 2.2 still provide useful measures of the size of the full minimax game tree.

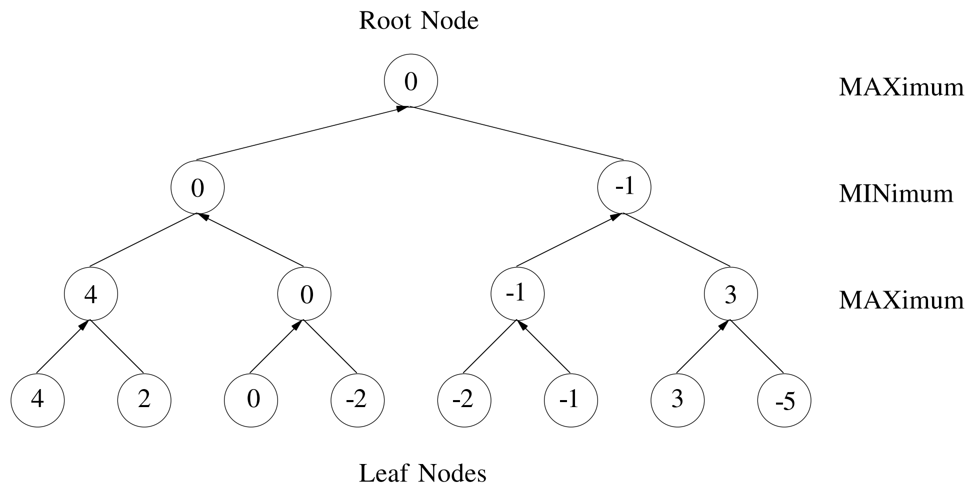


Figure 2.1 — Full Minimax Tree
(w = 2, d = 3. Arrows show path of values backed up from the leaf nodes.)

It is theoretically possible to determine the minimax value of a given game tree without visiting all of the nodes that naive minimax search visits. In fact, there are several algorithms designed to search a minimax tree without building a full minimax tree. Of them, the well-known Alpha-Beta algorithm remains the most popular. The number of leaf nodes in the smallest possible Alpha-Beta game tree or *minimal tree* is given by Formula 2.3 [SlD69,KnM75].

$$\mathrm{MT}_{\mathrm{NBP}}(w,d) = w^{\left\lceil \frac{d}{2} \right\rceil} + w^{\left\lfloor \frac{d}{2} \right\rfloor} - 1$$

Formula 2.3 — Leaf Nodes in Minimal Tree
(MT is for minimal tree, w is uniform width, d is fixed depth)

Of course, Formula 2.3 is still exponential with respect to depth, but it is a significant improvement over the full minimax tree.

2.2 Sequential Alpha-Beta Search

Alpha-Beta is a depth-first search algorithm for minimax trees that computes the correct value of the root position without necessarily visiting all of the tree nodes. Subtrees that are provably irrelevant to the value of the root position may be ignored or cut off for a savings in the search effort. In fact, Alpha-Beta is a special case of the more general class of branch-and-bound algorithms [KnM75]. In the terminology of [MaP85],[3] nodes where all of the subtrees are searched are called *ALL nodes* and nodes where some of the subtrees can be ignored are called *CUT nodes*. A *cutoff* is said to occur at CUT nodes. To detect cutoffs, Alpha-Beta maintains and updates a *search window*, (α,β). Node values strictly within the window, that is greater than α and less than β, are still relevant to the search. Node values outside the window are pruned. Alpha-Beta pruning is not a heuristic because it logically eliminates parts of the game tree and still computes the correct minimax value of the root position.

As an example of Alpha-Beta pruning, consider Figure 2.2. The minimax game tree is identical to the tree in Figure 2.1.

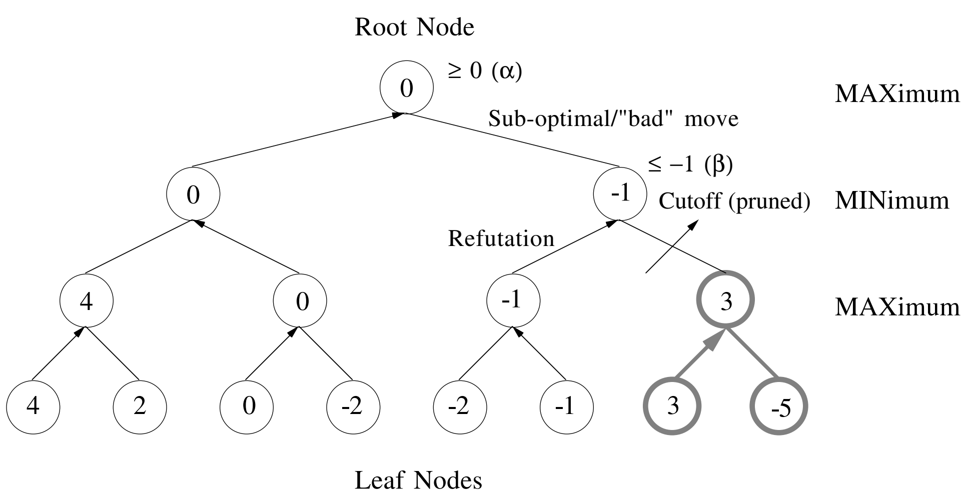


Figure 2.2 — Example of Alpha-Beta Pruning

Assuming a left-to-right order in the depth-first search, the left subtree of the root node returns a value of 0, which is used as the α value, or lower bound, for the search of the right subtree of the root node. When the -1 value is backed up to the MINimum ply, it becomes the β value,

---

3 Although this paper uses the terminology in the context of parallel search, they apply equally well to sequential search.

or upper bound, for the search. However, since $\alpha \geq \beta$, the search window is empty and the indicated subtree can be cut off or pruned. In game tree terminology, the move searched just before the cutoff is called the *refutation*, and the move that led to the cutoff node is sub-optimal, or informally, a "bad" move which has been refuted.

A variety of search algorithms are presented in this thesis in a Pascal-like pseudo-code to improve the clarity of the discussion. The pseudo-code is based on the presentation by Schaeffer [Sch87] and others (for example, [MaC82]). The program header in Figure 2.3 should be logically added to all of the pseudo-code to follow. Although the use of global variables is generally discouraged, their advantages will become apparent in later chapters.

```
PROGRAM
    GameTreeSearch;

VAR { Global Variables }
    TreeDepth : integer;                          { Depth of search tree to build }
    Width   : array[ 1..MAX_DEPTH ] of integer;   { Branching factor}
    Moves   : array[ 1..MAX_DEPTH ] of
              array[ 1..MAX_WIDTH ] of moveType;  { Moves leading to children }
    Index   : array[ 1..MAX_DEPTH ] of integer;   { Index of next move }
    Alpha   : array[ 1..MAX_DEPTH ] of integer;   { Alpha bound }
    Beta    : array[ 1..MAX_DEPTH ] of integer;   { Beta bound }
    Score   : array[ 1..MAX_DEPTH ] of integer;   { Best score so far }
    Best    : array[ 1..MAX_DEPTH ] of integer;   { Index of best move }

BEGIN { Program }
```

Figure 2.3 — Game Tree Search Program Header
(Global Variables Pseudo-Code)

The pseudo-code for the Alpha-Beta algorithm is shown in Figure 2.4. The details of some of the functions, `Evaluate()`, `legalMoves()` and `orderMoves()`, have been omitted. Also, if `p` is a position and if `move` is a legal move, then the notation `p.move` refers to the position reached by playing `move` in `p`.

The pseudo-code includes the common negamax and fail-soft enhancements to the basic Alpha-Beta algorithm. The *negamax framework* swaps and negates the bounds of the search window and also negates the returned value. It is equivalent to the minimizing-maximizing formulation of the original Alpha-Beta algorithm, but is more convenient because it always computes the arithmetic maximum instead of alternating with the arithmetic minimum. The pseudo-code also contains the *fail-soft* enhancement introduced by Fishburn [Fis81]. By initializing the value of `Score[ ply ]` to $-\infty$, the value returned by fail-soft Alpha-Beta is the maximum score of the subtrees visited, even if that value is less than the alpha bound itself. If no cutoff occurs, `Score[ ply ]` contains the value of the best subtree. As we will see later, this bounds information can be useful in certain search strategies. Although technically a misnomer, the fail-soft version of Alpha-Beta is often what is intended when referring to the Alpha-Beta algorithm, including in this thesis.

```
FUNCTION
   AlphaBeta( p : position; alpha, beta, ply : integer ) : integer;

VAR
   index, value : integer;                    { Index and value of current move }
   move : moveType;                           { Current move being searched }

BEGIN { AlphaBeta }

{ Check if at leaf node }
   if ( ply = TreeDepth ) then                { At leaf node? }
      return( Evaluate( p ) );                { Yes.  Statically evaluate }

{ Generate legal moves and order them }
   Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
   if ( Width[ ply ] = 0 ) then               { Any legal moves in this position? }
      return( Evaluate( p ) );                { No.  In checkers, this is a loss }
   orderMoves( Moves[ ply ] );                { Yes. Do move ordering }

{ Set up global variables }
   Index[ ply ] := 1;                         { Start with best ordered subtree }
   Alpha[ ply ] := alpha;
   Beta[ ply ] := beta;
   Score[ ply ] := -∞;                        { Initialize as per fail-soft }
   Best[ ply ] := UNKNOWN;

{ Search each subtree }
   while ( Index[ ply ] ≤ Width[ ply ] ) do
     begin { Recurse }                        { Search each subtree }
      index := Index[ ply ];                  { An unique move leads to the subtree }
      move := Moves[ ply ][ index ];
      Index[ ply ] := Index[ ply ] + 1;       { Advance index }

      value :=                                { Negamax the value and parameters }
         -AlphaBeta( p.move, -Beta[ ply ], -Alpha[ ply ], ply + 1 );

      if ( value > Score[ ply ] ) then        { If subtree is better, remember  it }
        begin { Update Best }
         Score[ ply ] := value;
         Best[ ply ] := index;
        end;  { Update Best }

      if ( Score[ ply ] ≥ Beta[ ply ] ) then         { Cutoff? }
         Index[ ply ] := ∞;                          { Yes.  Force exit from loop }
      Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );   { Improve window? }
     end; { Recurse }

   return( Score[ ply ] );                    { Done. Return best score }
END; { AlphaBeta }
```

Figure 2.4 — Negamax Version of Fail-soft Alpha-Beta Algorithm
(Boldface shows points of recursion or where the recursion ends.)

2.3 Move Ordering

Achieving quick cutoffs at CUT nodes through *move ordering* is one key to the efficiency of the algorithm. For example, in Figure 2.2, if the subtree that was pruned had been searched before its sibling (to the left), then there would not have been a cutoff. Ideally, a move that causes the cutoff should be searched first.

In a *best ordered* (or *perfectly ordered*) game tree, the best move is searched first and a cutoff occurs immediately after it, if it is a CUT node. At an ALL node, the best move gives the node its value. Assuming uniform width and fixed depth, if a game tree is best ordered at ALL nodes and at CUT nodes, Alpha-Beta builds the minimal tree. In a *worst ordered* tree, the subtrees are ordered from worst to best, thus the best subtree is searched last. The cutoff occurs only after all of the subtrees have been searched at the CUT node giving no savings in search effort. Alpha-Beta builds the full minimax tree if the moves are worst ordered. *Randomly ordered* game trees, where the best subtree is equally likely to be in any order, have also been studied. However, they do not represent what is observed in practice [Hsu90].

A game tree is *strongly ordered* if the first subtree searched is best 70 percent of the time and if the best subtree is in the first quarter of the branches 90 percent of the time [MaC82]. Although the classifications are non-analytical, it is generally agreed that the trees of efficient search implementations are strongly ordered. We can summarize the relationship between games trees built with different move orderings by [MaC82]:

Minimal Tree < Strongly Ordered Tree < Randomly Ordered Tree « Full Minimax Tree

Figure 2.5 — Relative Sizes of Game Trees with Different Move Orderings
(Minimal trees are best ordered. Worst ordered trees are full minimax trees.
Uniform width and depth is assumed.)

Several heuristics are commonly used to build strongly ordered game trees in practice. As with most rules of thumb, there are cases in which the heuristics fail. Furthermore, there is often no analytical characterization of the probability that the heuristic is correct in a given situation.

One heuristic combines iterative deepening with transposition tables. *Iterative deepening* is a technique that builds successively deeper game trees. The move ordering information gathered by building the game tree of depth n is used to help order the moves for the game tree of depth $(n + s)$, where s is the step between iterations. The best move for a node can be stored in a *transposition table*, which is hashed according to the board position represented by the node. During the next search iteration, the transposition table is queried and the suggested best move from the previous iteration is searched first. The heuristic is that the best move of the shallower search is likely to be the best move of the deeper search. More generally, iterative deepening is also used to regulate the time taken to decide on the best move for the root position and the transposition table also allows identical subtrees, created by transposing lines of play, to be cached and searched only once.

The *history heuristic* [Sch86,Sch89b] orders all of the moves at a node and not just the best move. It uses tables to accumulate a history of how often a move is shown to be best. The history score of a move is increased each time it is empirically shown to be best. Therefore, moves can be sorted according to their history scores. The heuristic is that the best moves at nodes in other parts of the tree have a history of being best and are likely to be best at the current node as well. If the transposition table cannot suggest a possible best move because the table entry is non-existent or has been overwritten, the history heuristic can suggest a best move.

Of course, there is no guarantee that the suggested best move of either the transposition table or history heuristic is actually best. In those cases, the ordering of the sibling moves by the history heuristic is in decreasing empirical probability of being the actual best move. In practice, the suggested best move of the transposition table is usually searched first and then the remaining moves are searched in the order suggested by the history heuristic. It is important to order all of the moves at a node. In the context of Figure 2.4, the function `orderMoves()` is considered to implement a move ordering mechanism, such as transposition tables supplemented with the history heuristic.

The popular *killer heuristic* [SlA77] is a special case of the history heuristic. There are several other move ordering mechanisms also in use [Sch89b,Mar92].

2.4 Aspiration Windows

The window (α,β) with which the search of a node is initiated is referred to as the *full window*. A search window of $(-\infty,+\infty)$ declares that all node values are relevant to the search since all possible node values fall inside the *infinite window*. However, it is possible to initiate a search with an artificially narrowed *aspiration window*. With window-based minimax algorithms, there is a well-known relationship between the size of the search window and the size of the search tree generated [Sch86]. Generally, the smaller the window, the smaller the tree that is searched. Intuitively, if the window represents the parts of the search space that are of interest, then a smaller search window will build a smaller tree because parts of the search space have already been declared *a priori* to be of no interest. Within certain limitations, aspiration windows can be used to search trees more efficiently.

With Alpha-Beta, the returned minimax value is correct only if it falls within the initial search window. If the minimax value is not inside the initial window, then the search is considered incomplete. The *aspiration search* generated an improper cutoff and did not search the part of the tree that determines its minimax value. Therefore, although there may be strong heuristic reasons to eliminate some node values from consideration *a priori*, there are no algorithmic justifications for using anything less than a full window. An aspiration window is a heuristic gamble that the final minimax value lies within the smaller search window, and can therefore be found with a smaller search tree.

One example of using an aspiration window is at the root of the game tree. If the previous iteration of iterative deepening returned a value of v for the root position, the next search can be initiated with a finite aspiration window covering a range centered on v. The size of the window is implementation dependent. If the score of the root position is stable between iterations, then it is likely that the next iteration will return the a value within the aspiration window.

Taking the concept of aspiration windows and fail-soft Alpha-Beta to the extreme results in the notion of *minimal windows* [Pea80,Fis81], and *minimal window searching*. Assuming a strongly ordered game tree, it is likely that the first subtree searched either determines the value of the node or causes a cutoff. We need to search the first branch of each node with a full window because we want to know the exact minimax value, say v, of the subtree. However, we do not need to know the minimax value of the other subtrees. We can settle for merely proving that the other branches are no better than the first branch. Therefore, we search all subtrees other than the first one with the minimal window $(v,v + 1)$ which has *zero width*. The window is different but the fail-soft Alpha-Beta search algorithm does not change. Minimal window search is not used to determine the actual value of a subtree, but to determine bounds on its relative value.

Recall that with fail-soft Alpha-Beta, the value returned from a search, even if the exact minimax value lies outside of the initial window, provides bounding information. Suppose the full window of the node is (α,β) and the minimal window used is $(v,v + 1)$. If the search returns a value z, and $z \leq v$, the search is said to *fail low* and while the exact value of the subtree is not known, it is known to be no better than v. We determined this as cheaply as possible since a minimal window was used. If $z \geq \beta$, then a normal Alpha-Beta cutoff has occurred and the node does not need to be expanded any further. Otherwise, if $v < z < \beta$, the search is said to *fail high*. It needs to be re-searched with a full window to find its true minimax value. A narrower window of (z, β) can also be used because the fail-soft Alpha-Beta returned z, which is a lower bound on the actual minimax value of the subtree.

The main disadvantage of minimal window searching is the danger of having to re-search a fail high subtree. The main advantage of minimal window searching is its ability to determine the value of a node with potentially less effort. Only the best move subtree is searched with a full window and the sibling subtrees are proven inferior with a relatively inexpensive minimal window search. Overall, the search savings due to minimal window searches of strongly ordered game trees outweigh the costs of re-searches [Mar83].

2.5 Structure of the Minimal Alpha-Beta Tree

The minimal game tree searched by the fail-soft Alpha-Beta algorithm has a predictable and important structure. Cutoffs can only occur at certain nodes and can never occur at other nodes. In terms of Knuth and Moore's classification [KnM75], Type 1 and Type 3 nodes are ALL nodes. The Type 2 nodes are CUT nodes. The pseudo-state diagram in Figure 2.6 below shows the relationship between the different node types.

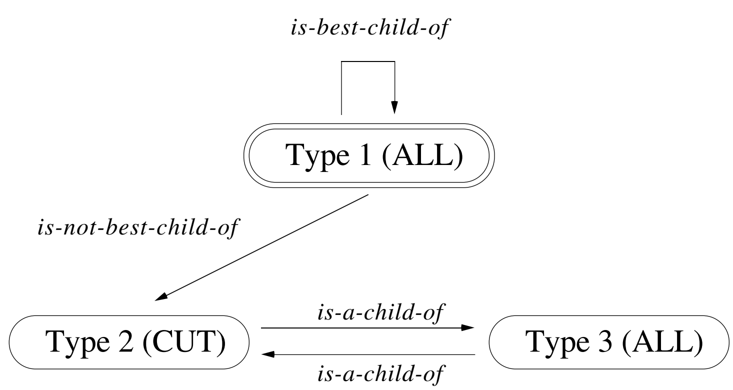


Figure 2.6 — State Diagram For Node Types

Cutoffs can occur at CUT nodes, but in worst case move ordering, all subtrees may actually be searched. With strongly ordered game trees, the worst case scenario is infrequent. Figure 2.7 shows the *ALL/CUT node structure* of the minimal Alpha-Beta game tree.

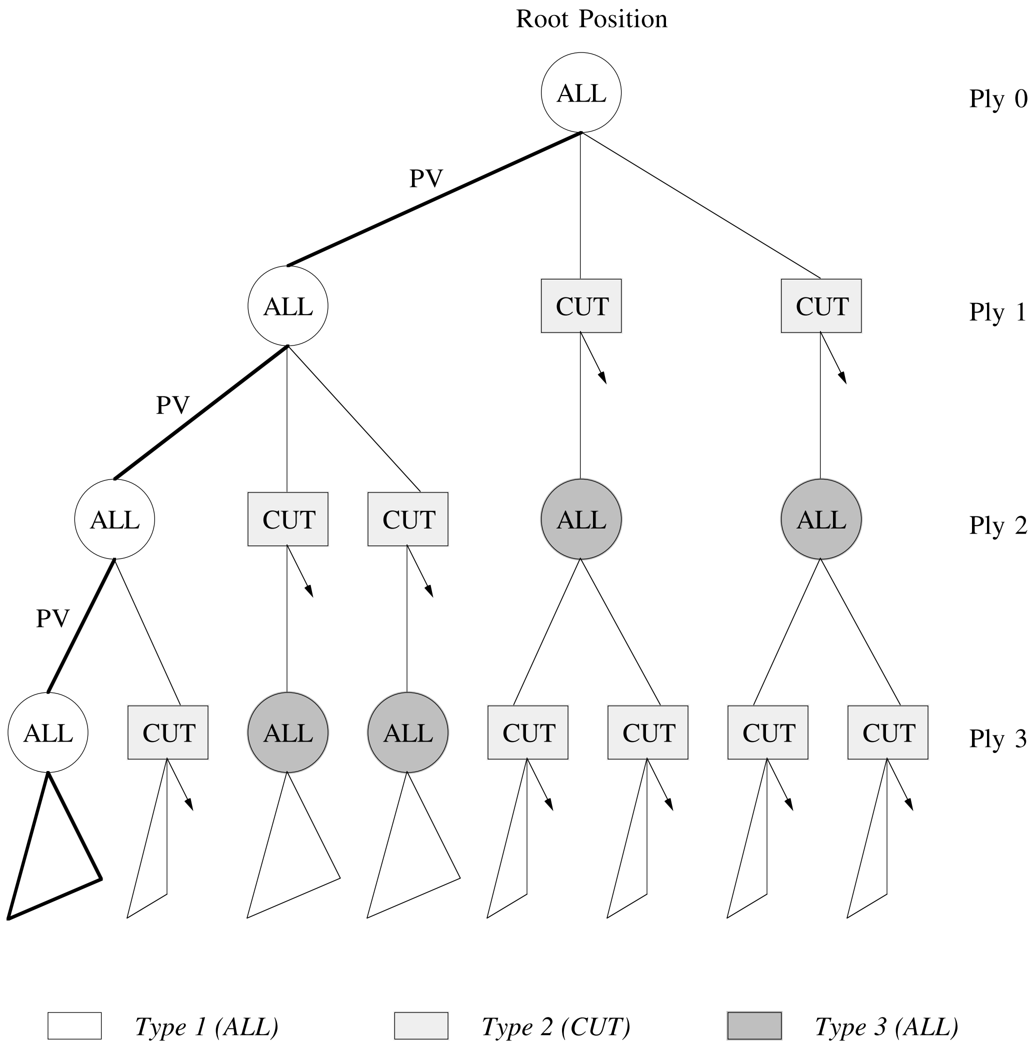


Figure 2.7 — ALL/CUT Node Structure of the Minimal Alpha-Beta Game Tree
(The principal variation (PV) moves are the best subtrees at Type 1 ALL nodes.)

The structure of the game tree in Figure 2.7 is true in the absence of aspiration window searches, which is the case for the fail-soft Alpha-Beta algorithm. When using minimal window searches,[4] there is one important change to the structure: at Type 3 (ALL) nodes, it may be possible to achieve a cutoff before all of the subtrees are searched. That is to say, Type 3 (ALL) nodes may in fact behave like CUT nodes if minimal windows are used. Intuitively, if the search window is of zero width, then any variation in the value of a subtree is enough to cause a cutoff. In fact, it is the possibility of generating a cutoff at a Type 3 (ALL) node that makes minimal window search more efficient than a full window search.

---

4 Which should not be confused with the notion of the minimal Alpha-Beta tree.

2.6 Principal Variation Search

Recognizing the usefulness of minimal window search with the strongly ordered game trees seen in practice, a new variation of Alpha-Beta search, *Principal Variation Search* (PVS), was introduced [Fis80,Fis81,MaC82]. The principal variation determines the minimax value of a given node, therefore PVS searches the first subtree, as determined by the move ordering, with a full window. If a fail high situation occurs during a subsequent minimal window search, the original choice for the best move was in fact not best and the fail high move needs to be re-searched. In effect, PVS heuristically narrows the windows of the non-principal variation moves to minimal windows.

Figure 2.8 gives the pseudo-code for Principal Variation Search. The main difference between the pseudo-code for Alpha-Beta and for PVS is the special treatment of the first ordered move. In effect, PVS unrolls the `while`-loop and searches the principal variation with a full window. The other subtrees are searched with a minimal window based on the value of the principal variation. The shading in the pseudo-code highlights the key differences between Alpha-Beta and PVS. Also note that PVS recursively calls itself and never calls Alpha-Beta, which was a technique introduced to minimal window searching by Marsland [Mar83] and, independently, Reinefeld in his *Negascout* algorithm [Rei83].

```
FUNCTION
    PVS( p : position; alpha, beta, ply : integer ) : integer;

VAR
    index, value : integer;                  { Index and value of current move }
    move : moveType;                         { Current move being searched }

BEGIN { PVS }

{ Check if at leaf node }
    if ( ply = TreeDepth ) then              { At leaf node? }
        return( Evaluate( p ) );             { Yes.  Statically evaluate }

{ Generate legal moves and order them }
    Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
    if ( Width[ ply ] = 0 ) then             { Any legal moves in this position? }
        return( Evaluate( p ) );             { No.  In checkers, this is a loss }
    orderMoves( Moves[ ply ] );              { Yes. Do move ordering }

{ Set up global variables }
    Index[ ply ] := 2;                       { Continue with other subtrees, later }
    Alpha[ ply ] := alpha;
    Beta[ ply ] := beta;

{ Search first ordered move with full window. Unroll the loop by one }
    Score[ ply ] :=
        -PVS( p.Move[ ply ][ 1 ], -Beta[ ply ], -Alpha[ ply ], ply + 1 );
    Best[ ply ] := 1;

    if ( Score[ ply ] ≥ Beta[ ply ] ) then   { Cutoff after pv? }
        Index[ ply ] := ∞;                   { Yes.  Will never enter loop }
    Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );       { Improve window? }

{ Search each sibling subtree with minimal window }
    while ( Index[ ply ] ≤ Width[ ply ] ) do
      begin { Recurse }                      { Search each subtree }
        index := Index[ ply ];               { An unique move leads to the subtree }
        move := Moves[ ply ][ index ];
        Index[ ply ] := Index[ ply ] + 1;    { Advance index }

        value :=                             { Negamax value and minimal window }
            -PVS( p.move, -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );

        if ( value > Score[ ply ] ) then     { Score improved, but did it fail high? }
         begin { Update Best }
            if ( value  > Alpha[ ply ] and value < Beta[ ply ] ) then
                Score[ ply ] :=              { Yes.  Re-search with full window }
                    -PVS( p.move, -Beta[ ply ], -value, ply + 1 );
            else
                Score[ ply ] := value;       { No }
            Best[ ply ] := index;
          end;  { Update Best }

        if ( Score[ ply ] ≥ Beta[ ply ] ) then      { Cutoff? }
            Index[ ply ] := ∞;                      { Yes.  Force exit from loop }
        Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );   { Improve window? }
      end;  { Recurse }

    return( Score[ ply ] );                  { Done. Return best score }
END;  { PVS }
```

Figure 2.8 — Principal Variation Search (PVS)
(Shading indicates key changes from `AlphaBeta()`)

The efficiency of PVS depends on move ordering. With best ordered trees, the first subtree searched is always on the principal variation, therefore there are never any fail high situations. Searching the best move first also provides the actual value of the node, say v, so that the remaining moves can be searched with a minimal window $(v,v + 1)$ centered on the correct value. Those minimal window searches will never fail high. However, when the move ordering deviates from best ordered, the best move is not expanded first and so an erroneous minimal window of $(v',v' + 1)$ where $v' < v$ is used. The minimal window search of the actual best move will fail high and the resulting re-search will return the value v. It is important that the newly determined value v be used for the minimal windows of the remaining subtrees. There may be other moves with the same value v as the best move. If those maximal subtrees are searched with $(v',v' + 1)$ they will also fail high and be needlessly re-searched. Fortunately, with the strongly ordered game trees encountered in practice, the first move is often best.

2.7 Alpha-Beta as a Proof Procedure

An interesting way to think of the Alpha-Beta algorithm is to view it as a guess-and-check proof procedure. Intuitively, the algorithm "guesses" at the best move at each node in the game tree. Then, it searches that line of play further to "check" if the move is indeed best or it finds a counter-example or *refutation* to the optimality of the chosen move. In practice, the guess of the best move and the guess of the best refutation are both determined by the various move ordering mechanisms in use.

Proving that a move is best requires searching all other possible moves and showing that they are inferior to the best, or principal variation, move. No move on the principal variation can be eliminated from consideration. That is why the principal variation consists only of Type 1 (ALL) nodes.

Proving that a move is inferior only requires showing that there is at least one response by the other side for which all of the counter-responses lead to inferior positions. For example, if the player chooses a move that allows the opponent to win a checker and all of the player's responses cannot win back the checker, then the inferiority of the player's original move is established. Although there may be other refutations available to the opponent, only one is needed and the remaining subtrees are cut off. This is the intuition behind Alpha-Beta pruning. A Type 2 (CUT) node is reached by an inferior move from the parent node and all of the moves at the following Type 3 (ALL) node cannot overcome that previous inferior move because they lead to inferior positions themselves, as in the above example. This explains the strict alternating pattern between Type 2 and Type 3 nodes indicated by Figures 2.6 and 2.7. It also explains why the alternation between Type 2 and Type 3 nodes begins at the point of deviation from the principal variation.

If a non-principal variation move is actually superior, as in a fail-high situation, then the behaviour of the Type 2 and Type 3 nodes are reversed. Because the original hypothesis that the non-principal variation is inferior is false, the search cannot actually prove the inferiority of the move. Instead of the other side having at least one refutation, it has no refutation and all subtrees of the Type 2 node must be searched to discover this, contrary to its predicted behaviour as a CUT node. And for each move from the Type 2 node, there is at least one response that improves the position for the other side. Once that move is discovered, a cutoff may occur at the Type 3 node, which is contrary to its predicted behaviour as an ALL node.

Chapter 3

Parallel Alpha-Beta Game Tree Search

There is a substantial body of literature on the parallel search of game trees. This is partly because deeper searches yield potentially better play, at least in chess [Tho82], and partly because the basic Alpha-Beta algorithm is an important technique that is a challenge to parallelize efficiently. This chapter discusses the fundamental concepts and techniques of parallel game tree search based on Alpha-Beta pruning. Selected previous research in the area is surveyed. Also, the differences between chess and checkers as the application domain are highlighted. The ideas introduced in this chapter form the foundation of the subsequent chapters describing *Chinook*'s parallel search.

3.1 Parallel Performance

To search a deeper game tree in the same amount of time, researchers have turned to parallelism. Searching one ply deeper in constant time requires a four to eight-fold increase in search power when chess is the application domain [Gil78]. In checkers, an extra one ply requires speeding up the search by a factor of two [SCT92].

A popular measure of parallel performance in game tree search, and parallel computing in general, is the *speedup*. Intuitively, speedup is the measure of how much faster a problem is solved using n processors instead of just one processor. If using n processors results in a n-fold speedup, then it is a *linear speedup*. However, true linear speedup is rare and highly dependent on the nature of the problem. Speedup is defined as:

$$\text{speedup} = \frac{\text{“best” solution time on one processor}}{\text{solution time on } n \text{ processors}}$$

Formula 3.1 — Definition of Speedup

The notion of "best" solution time on one processor is both important and contentious. Typically, the best sequential solution time used is derived from the best implementation available to the authors instead of from an analytical definition of what is best. Although this is a practical approach to the quantity of the speedup, the parallel version of a poor sequential algorithm may be misleading as to the quality of the speedup. For the exact same algorithm, using an unoptimized sequential implementation and slower processors can actually improve the speedup values [SuG91]. Despite its drawbacks, there is little consensus on an alternative measure of overall parallel performance and therefore, this thesis will also use the speedup metric.

Closely related to speedup is the notion of *speedup efficiency*. It represents the amount of speedup potential that is actually achieved. It is defined as:

$$\text{speedup efficiency} = \frac{\text{speedup on } n \text{ processors}}{n}$$

Formula 3.2 — Definition of Speedup Efficiency

Figure 3.1 shows a typical speedup graph with lines and curves representing linear speedup, linear speedup with a smaller slope and diminishing returns for additional processors. There is an additional curve representing parallel *slowdown*, where adding processors beyond a certain threshold decreases performance, and can even make the parallel implementation slower than the sequential implementation. Speedup and speedup graphs will be recurring themes throughout this thesis.

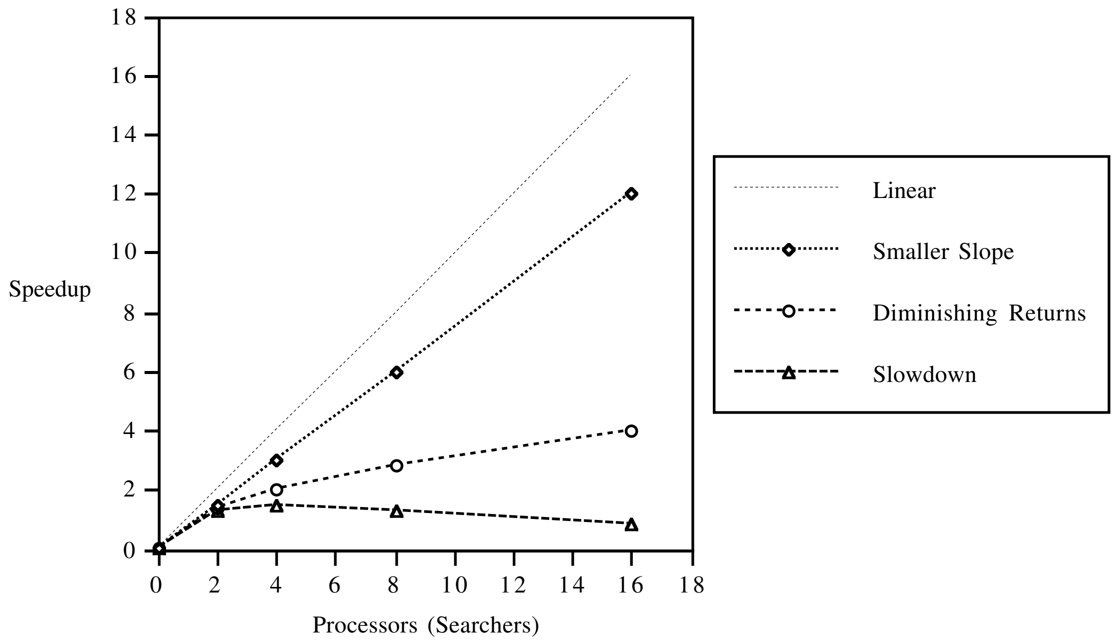


Figure 3.1 — Example Speedup Graph

An important concept related to parallel performance is the notion of work *granularity* . A sequential algorithm does not have to incur the costs associated with coordinating, communicating with or managing the work given to parallel processors. Each point of interaction between parallel processes is a drain on performance. Granularity captures the notion of how much actual work is completed by the algorithm between points of interaction. *Fine-grained* parallelism and *low granularity* work both imply short intervals of useful computation between points of interaction. Therefore, *coarse-grained* parallelism or *high granularity* work, where there is a large amount of computation between interaction points, is preferable.

Another important concept is *load balancing*. At a given point in the computation, the amount of parallel work available, called the *degree of parallelism*, may be less than the number of available processors. Therefore, some processors receive no work and the result is *starvation*. Work can also be unevenly distributed, particularly if it is impossible to predict the amount of computation required for a piece of work. A processor that is given a disproportionately large piece of work may become a *straggler*, forcing the other processors to wait at a synchronization point. If the number of synchronizing processors, the *synchronization fan-in*, is large then the performance losses due to stragglers is also large.

There are two broad strategies for dealing with load balancing problems. First, instead of allowing processors to be idle, a piece of work already assigned to a processor can be preempted, further subdivided and redistributed. This technique can also be used to avoid starvation and to deal with stragglers. Second, if it is possible, the parallel algorithm can try to create many pieces of work *a priori*. If the parallel processors are overloaded with many units of work, then idle processors are simply given additional work. The danger is that subdividing existing pieces of work, or creating more units of work by making them smaller, also reduces the granularity of the work.

3.2 Sources of Parallelism

To date, there have been four broad strategies used to extract parallelism from game playing programs. A combination of these strategies can be used in practice.

With *component parallelism*, the computationally expensive parts of the program are parallelized. Among the hardware assisted chess machines *Belle* [CoT82], *Hitech* [BeE89] and *Deep Thought* [HAC90], parallelizing the move generation and evaluation function has been a popular and successful strategy at the circuit level. Among the software-only programs, the low granularity of component parallelism has made it a less attractive approach, with *Cray Blitz* [HGN90] being a notable exception.

Another approach is *system parallelism*. In the distributed chess program *ParaPhoenix* [Sch89a], one team of parallel searchers applies the relatively expensive chess knowledge of *Phoenix* in its Alpha-Beta search. A second team of parallel searchers, *Minix*, uses a limited set of knowledge and heuristics to do a speculative, special purpose Alpha-Beta search for tactical advantages. In effect, *ParaPhoenix* consists of two systems, each contributing to the program's play, but each building different game trees. *Minix* is reported to improve the play of *ParaPhoenix* and, given the diminishing returns for additional *Phoenix* searchers, it is an alternate way to utilize additional processors. However, aside from *ParaPhoenix*, system parallelism in computer game playing has not gained the attention of researchers.

For speeding up the underlying game tree search, *parallel aspiration search* was one of the first strategies to be tried [Bau78]. In parallel aspiration search, the Alpha-Beta search window is partitioned into disjoint ranges and given to different searchers. Each searcher is given a smaller window, therefore the searches are finished more quickly. However, each searcher must still build a minimal tree, regardless of the window size, thus limiting the potential speedup to an upper bound of 5 to 6 [Bau78].

Tree decomposition is another broad strategy that has attracted attention and, arguably, garnered the most success. It is the focus of this thesis. A *split node* exists where the different subtrees rooted at the node can be searched concurrently by parallel searchers in a process known as *tree-splitting* [Fis81]. Each searcher can be a different parallel processor. Alpha-Beta pruning and other search enhancements can be used to order and combine the parallel search results. A spectrum of algorithms have been proposed, differing prominently in how many split nodes are allowed to exist concurrently and in how the split nodes are chosen. Four representative algorithms are considered in detail. *Principal Variation Splitting* (PVSplit) [MaC82] allows exactly one split node at any given time. *Dynamic PVSplit* (DPVSplit) [Sch89a], the *Zugzwang* approach [FMM89] and the *Delayed Branching Tree Expansion* (DBTE) algorithms [Hsu90] allow multiple split nodes. The main difference among the multiple split node approaches is in how the split nodes are selected. PVSplit and DPVSplit are depth-first search algorithms. The *Zugzwang* approach and the DBTE algorithms are not as limited in the way they traverse the game tree.

3.3 Parallel Overheads

Although tree decomposition offers the potential for speedups, it also introduces certain overheads that do not occur in sequential Alpha-Beta search and which detract from linear speedup.

Search overhead (SO) is the ratio between the number of nodes in the tree built by the parallel and sequential algorithms to return the minimax value of the tree. Formally, search overhead is [MOS85]:

$$\text{SO} = \frac{\text{nodes searched on } n \text{ processors}}{\text{nodes searched on one processor}} - 1$$

Formula 3.3 — Definition of Search Overhead

There are two main sources of search overhead, both related to information deficiency. First, there could be a lack of information regarding cutoffs. A parallel algorithm that performs subtree searches concurrently at a CUT node does not know *a priori* which subtree will cause a cutoff and make the other searches irrelevant. Second, there could be a lack of information regarding optimal search windows. Particularly with minimal window techniques, the values of previously searched subtrees can be used to improve the windows of subsequent searches. Without the benefit of previous search results, the windows of parallel searches may be wider than needed or even incorrect, resulting in larger subtrees or the overhead of re-searching subtrees. The lack of timely information about cutoffs and windows results in search overhead. It is ironic, given all the effort invested in parallel algorithms, that Alpha-Beta is an inherently sequential algorithm due to this information dependency.

Interestingly, it is possible to observe negative search overhead. Even the best sequential search implementations do not always achieve perfect move ordering at cutoff nodes. If a parallel algorithm can exploit this flaw and achieve a cutoff more quickly, it might result in a negative search overhead.

Synchronization overhead (SY) is incurred when an algorithm forces a searcher to wait for another searcher to finish. To combat information deficiency, some parallel algorithms use synchronization to gather all of the information regarding a node so that it can be used in subsequent searches. As well, search extensions, pruning shortcuts and other search enhancements may cause the work given to one searcher to be substantially more than another searcher. This leads to a load balancing problem when a straggler searcher forces other searchers to wait at a synchronization point.

Of course, there are strategies to reduce both the search and synchronization overheads. It is a difficult problem because there is a close relationship between the two overheads and trade-offs must be made carefully.

Communication overhead (CO) is caused by parallel searchers exchanging data, using locks and semaphores to coordinate access to shared resources and to communicate events. Of course, the sequential algorithm does not need to communicate data with parallel processors. For convenience, whatever is not attributable to search or synchronization overhead is defined to be communication overhead.

3.4 Respecting the Structure of the Alpha-Beta Tree

The Alpha-Beta cutoff has the potential to create parallel search overhead. Under perfect move ordering at a CUT node, the subtree that causes the cutoff is searched first and all of the other subtrees can be pruned. However, if a CUT node is used as a split node, and several subtrees are searched in parallel, there is likely to be waste of search effort because complete pruning is no longer possible. Consequently, it has been observed that one way to reduce search overhead is to search the subtrees of a CUT node in sequential order rather than in parallel.

Therefore, a heavy-handed rule is that ALL nodes should always be split and CUT nodes should never be split. Later on, we will consider exceptions to this rule, but the basic wisdom remains a key component of any parallel search algorithm based on tree decomposition. Therefore, if an algorithm decides to split or not to split a node with consideration given to the tree's node structure, it is informally said to *respect the ALL/CUT node structure of the game tree*. In fact, all of the algorithms discussed below incorporate this concept in some form.

3.5 Principal Variation Splitting

Principal Variation Splitting (PVSplit) [MaC82] is a depth-first, tree-splitting algorithm that grew, in part, out of the sequential Principal Variation Search (PVS) algorithm discussed in Chapter 2. It creates parallel work by splitting nodes along the principal variation. As seen in Figure 2.7, each of the nodes on the principal variation is an ALL node, therefore PVSplit respects the ALL/CUT node structure of the game tree. Recursing down the principal variation concentrates the parallel effort where it is potentially most useful, and backing up the principal variation carries important window information for the search of the sibling subtrees.

PVSplit forms the foundation of *Chinook*'s parallel search and will be discussed in detail in Chapter 4. However, it is important to emphasize two aspects of PVSplit for the current discussion. First, PVSplit only splits Type 1 (ALL) nodes and it splits the nodes in a depth-first, in-order fashion as values are backed up along the principal variation. PVSplit uses the backed up value to improve the search windows of the sibling subtrees. Second, PVSplit only allows a single split node at a time. All of the searchers must synchronize at the current split node before the algorithm can back up one ply and create the next split node. Consequently, PVSplit is particularly susceptible to synchronization overheads.

3.6 Dynamic Principal Variation Splitting

The *Dynamic Principal Variation Splitting* (DPVSplit) algorithm [Sch89a] extends basic PVSplit in a way to deal with the synchronization overhead caused by stragglers. In essence, searchers that are otherwise idle at the synchronization point can be dynamically reassigned to help with busy searchers. As searchers are reassigned to help other searchers, a processor tree hierarchy is formed with the straggler processor at the root of the processor tree. The searchers are able to share their work because they apply the PVSplit algorithm to their assigned subtrees instead of regular PVS.

Two disadvantages with DPVSplit include the potential for unbalanced processor trees and the reduction in the granularity of parallel work. As idle searchers are assigned and reassigned to busy searchers, it is possible to form an inefficient pipeline of processors instead of a balanced tree, in the worst case. A deeper hierarchy of processors implies more synchronization points and more synchronization losses. Furthermore, splitting up the work of one searcher for a reassigned searcher implies that the work created by the "dynamic" aspect of DPVSplit is smaller than the work created by the basic PVSplit. In effect, DPVSplit allows multiple split nodes to be introduced when necessary to deal with stragglers. However, the additional split nodes are all lower in the game tree than the original split node on the principal variation. Consequently, the subtrees at the new split nodes are smaller and potentially suffer the drawbacks of fine-grained work. In practice, DPVSplit does improve the performance of PVSplit. However, it appears that it does not change the overall asymptotic shape of the speedup curve.

Both PVSplit and DPVSplit are depth-first strategies that largely depend on a single controller. DPVSplit introduces a limited amount of distributed search control when idle searchers are reassigned to help busy searchers. In that case, the straggler process becomes both a manager of searchers and a searcher. However, with both algorithms, the search control remains largely centered on the principal variation.

The advantage of centralized control is that all of the information on cutoffs and search windows can be concentrated and applied at the split node, reducing search overhead. The disadvantage is that the single control point can be come a bottleneck, increasing synchronization overhead. The addition of more searchers increases the synchronization fan-in and if the branching factor of the split node is low, there may be insufficient parallel work for the searchers. Intuitively and empirically, PVSplit-based algorithms perform best with small numbers of processors and trees with high branching factors. The ratio between the number of searchers and the branching factor, called the *parallelism overload factor*, is an important measure of how likely there is to be starvation and how high the synchronization fan-in will be.

Another advantage of PVSplit-based algorithms is that, since they are depth-first searches, the amount of storage required for the search control is linearly proportional with respect to the depth of the search and the number of searchers. The low storage overhead for depth-first search algorithms is a key strength in actual implementations.

3.7 Other PVSplit-Based Algorithms

Some of the other efforts in PVSplit-based parallel algorithms include the *Cray Blitz* [HSN89] and *Waycool* [FeO88] chess programs. The *Waycool* program is interesting for its reported 101-fold speedup on a 256 node hypercube multicomputer. However, as pointed out by several researchers, including Hsu [Hsu90], that speedup value does not take into consideration the beneficial growth in transposition table size as processors are added.

Although both projects introduced interesting load balancing techniques that are effective for their individual programs, it is difficult to compare their relative merits. In fact, the inability to meaningfully and quantitatively compare different search ideas, except when implemented within a single chess program, is a fundamental limitation of research in this area. Nonetheless, qualitative comparisons of parallel search ideas, particularly between PVSplit-based algorithms and other parallel search paradigms, can be instructive.

3.8 The *Zugzwang* Approach

One of the key design decisions made for the *Zugzwang* distributed chess program [FMM89] is to completely distribute the parallel search control. Each searcher executes the same search and control code and are virtually identical in functionality. Each searcher is responsible for requesting work from other searchers. Each searcher is able to receive work and search it locally or redistribute subproblems to other searchers. The lack of a central controller implies that the potential bottleneck noted for PVSplit and DPVSplit may be avoided.

Upon receiving a problem, namely a node to be searched, a searcher breaks up the node into subproblems which it can divide further and search locally, or if asked, can be passed on to another searcher. The creator of the split node, as in PVSplit, is its master and is ultimately responsible for its completion. Given a choice of subproblems to redistribute, the subtree closest to the root of the original problem is preferred. Alpha-Beta pruning and other search enhancements are also used. There can be multiple split nodes within a single searcher and multiple split nodes among the parallel searchers. It should be noted that each searcher under DPVSplit performs a similar procedure. The differences arise in how work is shared between searchers.

To supplement the cyclic process of receiving work, dividing the work and, possibly, reassigning some of the work are two principles described as Young Brothers Wait and Helpful Master.

Viewing the different subtrees rooted at a node as siblings, *Young Brothers Wait* states that, although the work has been divided, the moves ordered after the first, or eldest, subtree are not available for redistribution until the first subtree has been completed. Consequently, if the split node is a CUT node and the cutoff occurs after the first subtree, no search overhead is incurred through the premature reassignment of work. Furthermore, information from the first subtree that may improve the search windows of the siblings can be applied before the work is given out. The Young Brothers Wait concept is a restatement of PVSplit's criteria for creating a new split node: the first ordered subtree must be searched first.

Interestingly, the behaviour of Young Brothers Wait is the same at both CUT and ALL nodes. Both types of nodes are completely splittable after the first search result is returned. There is no reported mechanism for preventing the splitting of CUT nodes, although it would be simple to implement. Even if the elder subtree does not cause a cutoff, the likelihood of each subsequent subtree causing the cutoff is high in a strongly ordered tree, assuming that the node is behaving as a CUT node. Young Brothers Wait avoids search overhead only in the case of best move ordering, which is not always the case in practice.

With the *Helpful Master* concept, the master of a split node that is waiting for search results to be returned can decide to *assign itself* to help a searcher finish a subtree for which the master is waiting. Thus, the Helpful Master is a load balancing strategy that also has a completely distributed control structure. Unlike DPVSplit, each searcher is empowered to decide for itself if it will help another searcher. The distributed search initiation and the Helpful Master results in a dynamic and effective form of load balancing.

In general, *Zugzwang*'s search begins with a single searcher being given the root position of the game tree. As it divides the node, it follows the Young Brothers Wait concept and ultimately creates work than can be redistributed to other searchers. They, in turn, create more work which can be used to load balance the search. The Helpful Master concept also aids in load balancing the search, particularly in the case of a straggler process.

The strongest and most unique feature of the *Zugzwang* approach is the complete distribution of search control among peers. It is a natural approach for the distributed hardware used to implement the chess playing program. On the one hand, the strategy reduces the synchronization fan-in at split nodes and avoids the bottleneck of single split node algorithms. On the other hand, the strategy creates more synchronization points and the frequent interactions between searchers to receive work and communicate information may result in low-granularity parallelism. From the reported speedups and the success of the *Zugzwang* program in tournament play, the ideas seem promising.

3.9 Delayed Branching Tree Expansion Algorithms

Hsu introduced a family of multiple split node, non-depth-first, parallel Alpha-Beta search algorithms [Hsu90]. His design goal was to allow the use of several hundred, and even thousands, of parallel searchers in a new generation chess machine.[5] The family of Delayed Branching Tree Expansion (DBTE) algorithms avoid the concurrent expansion of sibling subtrees of a CUT node, but the individual algorithms differ in the strategy used to select the next split node. The details of the Leftmost-First (LF), Best-First (BF) and First-Come-First-Served (FCFS) strategies can be found in [Hsu90]. Of the three strategies, only the Leftmost-First DBTE (LF-DBTE) is considered in depth.

The DBTE algorithms are influenced by the hardware model chosen for the new chess machine: a host machine builds the upper plies of the game tree in memory and decides on the split nodes to be given to the next available searcher from a pool of processors. Therefore, a given DBTE algorithm is the scheduling or priority queue strategy for split nodes. The host can be a workstation or a collection of workstations. The parallel searchers are chess search engines using custom chips for the move generator and the static evaluation function. Therefore, the new chess machine exploits both fine-grained component parallelism and coarse-grained search parallelism through tree decomposition.

At the heart of the DBTE algorithms is the notion of the critical tree, which is related to the minimal tree. The critical tree is the actual proof tree and represents nodes that must be searched in order to determine the minimax value of the root position. Hsu distinguishes between the notion of "critical work" that is implied by the critical tree and "mandatory work" that is the basis of work by Akl, Barnard and Doran [ABD82] and Finkel and Fishburn [FiF83]. In fact, Hsu points out that the notion of mandatory work described by Finkel and Fishburn is "not even mandatory for the full window [Alpha-Beta] search which uses a starting window of $(-\infty, +\infty)$" [Hsu90]. However, he also notes that even the critical nodes are not necessarily searched if an aspiration window is used. Still, "the set of the critical nodes does have the merit of being worth evaluating when there is *nothing better* to do" [Hsu90, pg. 73]. In essence, the critical nodes are the preferred choices for the multiple split nodes.

---

5 Dr. Hsu and his colleagues created the ChipTest, Deep Thought and Deep Blue chess machines [HAC90].

As with the PVSplit-based algorithms, the DBTE algorithms explicitly respect the ALL/CUT node structure of the Alpha-Beta game tree. However, the DBTE algorithms allow for multiple split nodes, which is not the case for the basic PVSplit. The use of multiple split nodes is a fundamental way of growing the search tree in both *Zugzwang* and the DBTE algorithms. Unlike the *Zugzwang* approach, the DBTE algorithms have a centralized scheme for search control. The searchers of the chess machine are not general purpose processors. A piece of work given to a DBTE searcher is indivisible. The host machine controls the search and performs the splitting of nodes.

The DBTE algorithms have been analytically shown to possess two important properties. First, the Leftmost-First DBTE algorithm is proven to provide speedup that is asymptotically close to linear for a fixed number of processors, as the depth of the best-first ordered search tree is increased. Therefore, regardless of the number of searchers, the speedup of the search improves as the size of the search tree grows. In Hsu's view, this is a necessary but not sufficient condition for a good parallel Alpha-Beta algorithm. Second, Hsu proves that the Leftmost-First DBTE algorithm theoretically never searches a node that the weak Alpha-Beta algorithm does not search. The weak Alpha-Beta algorithm does not enjoy cutoffs due to bounds information from ancestor nodes more than two plies away, the so-called *deep cutoffs*. Therefore, the normal Alpha-Beta algorithm, with deep cutoffs, never searches more (and usually fewer) nodes than weak Alpha-Beta. Leftmost-First DBTE searches no more nodes than the weak Alpha-Beta as well. Therefore, given the asymptotic optimality of the speedup and the upper bound on nodes searched provided by the weak Alpha-Beta algorithm, Hsu concludes that the Leftmost-First DBTE algorithm has a non-trivial lower bound on its parallel Alpha-Beta speedup.

Hsu's important analytical results do not preclude the fact that other parallel algorithms may also possess the same theoretical speedup properties. However, the reporting of such analytical properties of a parallel Alpha-Beta algorithm is sufficiently rare that the Leftmost-First DBTE theoretical results are noteworthy. Certainly, no such analytical results of the PVSplit-based algorithms and the *Zugzwang* approach have been reported. Although results based on an actual implementation of DBTE have not been reported, Hsu did perform simulations of the algorithm to support his theoretical results.

The DBTE algorithms remain a promising alternative to other algorithms. Its main disadvantages are that it gives up the deep cutoffs and that it potentially requires a lot of storage for the nodes of the expanding tree, since it is not a depth-first strategy. However, the need for multiple split nodes is clear as the number of parallel searchers increases. Furthermore, the desirable theoretical properties of the algorithm distinguish it from other algorithms proposed to date.

3.10 Application Domains: Chess and Checkers Game Trees

PVSplit, DPVSplit, the *Zugzwang* approach and the DBTE algorithms have primarily been applied to or proposed for chess game trees. However, checkers game trees have two distinguishing features from chess game trees. First, the average branching factor in checkers is much less than chess. In chess, a typical position has a branching factor ranging from 35 to 40 [Gil78]. In the commonly used Bratko-Kopec chess test suite [KoB82], the branching factor is about 35. The checkers game trees of the test suite used for this thesis have an average branching of 2.84.[6] That is to say, the branching factor in chess game trees is over an order of magnitude higher than in checkers game trees. This has clear implications for tree decomposition strategies such as PVSplit. Second, the distribution of branching factor values in checkers trees is bimodal. There is a significant difference in branching factor depending on whether the node represents a capture or non-capture position. In checkers, if a capture is possible, it must be taken. Consequently, the average branching factor for capture nodes in the test suite is 1.28 and for non-captures nodes it is 7.93. Because there are more capture nodes than non-capture nodes, the weighted average is 2.84. The *bimodal distribution* of branching factor values has implications if a chosen split node is also a capture position.

---

6 The Tinsley-*Chinook* 1992 test suite will be discussed later and in Appendix B. All reported branching factors for checkers game trees are measured quantities.

Figure 3.2 graphically shows the difference in branching factor between chess and checkers game trees.

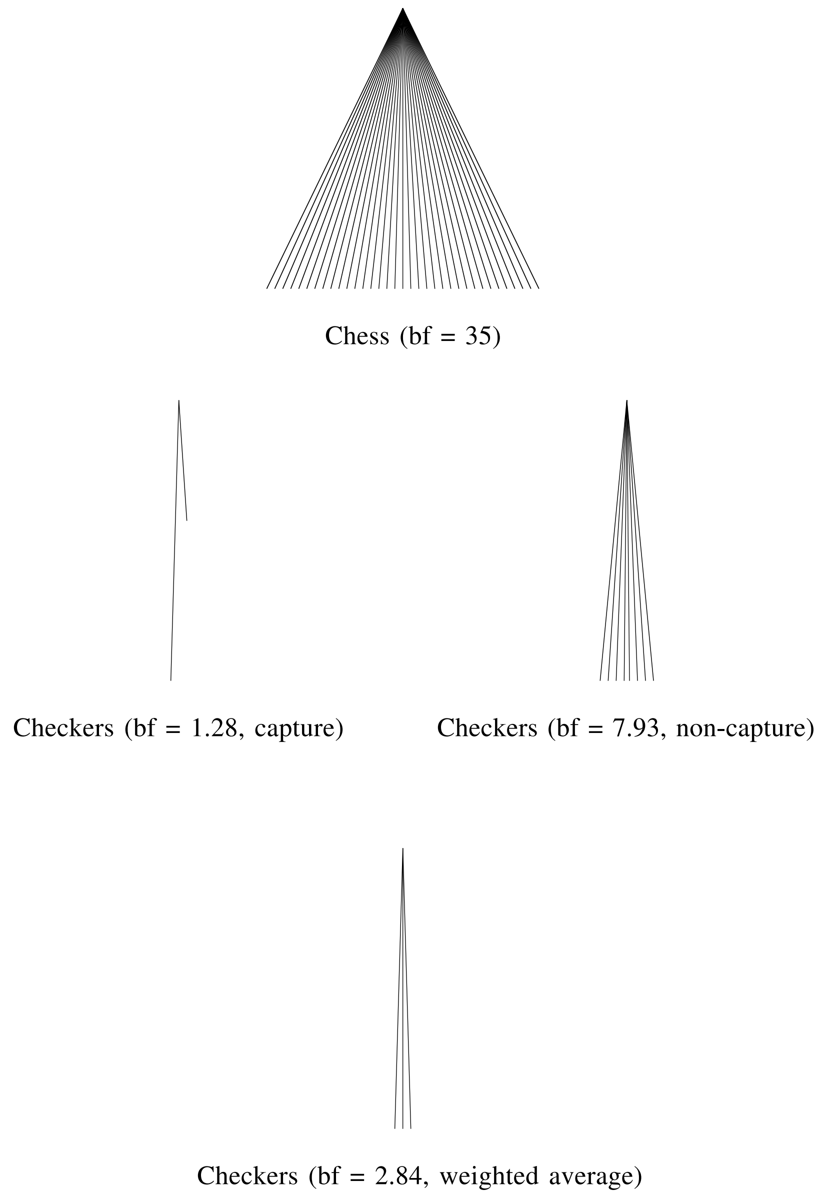


Figure 3.2 — Chess and Checkers Game Trees

Because PVSplit and other tree decomposition strategies create parallel work by splitting nodes, a high branching factor means that there is a large quantity of parallel work at each split node. Starvation is not a serious problem with wide chess game trees and relatively low numbers of searchers. Similarly, if the degree of parallelism offered by the branching factor is higher than the number of searchers, the load balancing problem can be ameliorated by giving a new piece of work to a searcher who finishes early.

In checkers, if the split node is a capture position, it is likely that all but one of the searchers will be starved since the average branching factor is less than 2 at a capture node. Furthermore, since PVSplit recurses down one of the branches before splitting the node, the effective degree of parallelism at a split node which is a capture position is less than one. If the number of searchers is greater than 8, it is possible that some searchers will never receive work. When searchers can starve due to an insufficient branching factor at a split node, the game tree is said to be *branch narrow*, or simply *narrow*. It is purposely a relative term that depends on the parallelism overload factor concept discussed earlier.

Clearly, the low branching factor of checkers game trees means PVSplit may not be able to effectively utilize even a moderate number of parallel searchers. Nonetheless, it was decided that a PVSplit-based version of *Chinook* needed to be implemented as a first step to understanding how the narrow checkers game trees would affect parallel game tree search performance.

Chapter 4

Parallel Chinook

Early on in the creation of a parallel *Chinook*, or *ParaChinook*, it was decided that the Principal Variation Splitting (PVSplit) algorithm would be the basis of the first implementation. PVSplit's popularity and success with chess researchers inspired some confidence. Also, the centralized search control and its similarities with the algorithm used in sequential *Chinook* meant that debugging may be easier. Furthermore, the first intended use of *ParaChinook* was for the Man Versus Machine World Draughts Championship in August 1992, and the parallel computer available for that match supported only eight parallel processors. We hoped that the relatively low parallelism of the hardware would not overwhelm the low degree of parallelism of narrow game trees. For a variety of reasons, PVSplit was the pragmatic choice for an initial *ParaChinook.*

4.1 *ParaChinook*'s Principal Variation Splitting

As discussed in Chapter 3, Principal Variation Splitting (PVSplit) [MaC82] is a depth-first, tree-splitting technique that concentrates the parallel searchers along the principal variation. Only Type 1 (ALL) nodes are on the principal variation and only one split node can exist at a time.

Figure 4.1 contains the pseudo-code for an enhanced version of PVSplit, similar to Parallel Minimal Window Search (PMWS) [MaP85], that uses minimal windows for non-principal variation subtrees. The shaded regions highlight the key differences between the parallel PVSplit algorithm and the sequential Principal Variation Search (PVS) algorithm previously described. Of course, there are strong similarities between the two algorithms.

Only one process, the *controller*, executes the PVSplit code. Work is assigned to parallel *searchers* who themselves use the sequential PVS algorithm. The controller does a minimal amount of computation and searcher management, thus it does not require a dedicated physical processor. It can share a processor with a searcher without having a significant impact on performance. Having separate controller and a searcher processes is just a convenient abstraction and programming technique.

```
VAR { New Global Variables }
    WorkOut   :   array[ 1..MAX_DEPTH ] of integer;    { Results outstanding }
    Value     :   array[ 1..MAX_WIDTH ] of integer;    { Results of each subtree }

FUNCTION
    PVSplit( p : position; alpha, beta, ply : integer ) : integer;

VAR
    index, value : integer;                    { Index and value of current move }
    move : moveType;                           { Current move being searched }

BEGIN { PVSplit }

{ Check if at threshold for minimum granularity.  Do not recurse to leaves. }
    if ( ply = ( TreeDepth - MinGranularity ) ) then
        return( PVS( p, alpha, beta, ply ) );   { Yes.  Do sequentially. }

{ Generate legal moves and order them }
    Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
    if ( Width[ ply ] = 0 ) then               { Any legal moves in this position? }
        return( Evaluate( p ) );               { No.  In checkers, this is a loss }
    orderMoves( Moves[ ply ] );                { Yes. Do move ordering }

{ Set up global variables }
    Index[ ply ] := 2;                         { Continue with other subtrees, later }
    Alpha[ ply ] := alpha;
    Beta[ ply ] := beta;
    WorkOut[ ply ] := 0;                       { Search results still outstanding }

{ Search first ordered move with full window. Unroll the loop by one }
    Score[ ply ] :=
        -PVSplit( p.Move[ ply ][ 1 ],
                 -Beta[ ply ], -Alpha[ ply ], ply + 1 );
    Value[ 1 ] := Score[ ply ];
    Best[ ply ] := 1;

    if ( Score[ ply ] ≥ Beta[ ply ] ) then    { Cutoff after pv? }
        Index[ ply ] := ∞;                     { Yes.  Will never enter loop }
    Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );      { Improve window? }
```

```
{ Search each sibling subtree with minimal window }
{ Split node incomplete if there are local subtrees unsearched or if there are searchers still to return }

    while ( ( Index[ ply ] ≤ Width[ ply ] ) OR ( WorkOut[ ply ] > 0 ) ) do
      begin { while }
        if ( "∃idle searcher" AND Index[ ply ] ≤ Width[ ply ] ) then
          begin { Local work out }                { Search each local subtree }
            index := Index[ ply ];                { An unique move leads to the subtree }
            move := Moves[ ply ][ index ];
            Index[ ply ] := Index[ ply ] + 1; { Advance index }

            Value[ index ] :=                     { Negamax value and minimal window }
                -FarmWork( p.move,
                    -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );
            WorkOut[ ply ] := WorkOut[ ply ] + 1;    { Keep track of work given }
          end; { Local work out }

        else if ( index := ReturnedResult() ) then
          begin { Local results in }
            value := Value[ index ];
            WorkOut[ ply ] := WorkOut[ ply ] - 1;    { Work received }

            if ( value > Score[ ply ] ) then { Score improved, but did it fail high? }
              begin { Update Best }
                if ( value > Alpha[ ply ] and value < Beta[ ply ] ) then
                  begin
                    Value[ index ] :=             { Yes.  Re-search }
                        -FarmWork( p.move, -Beta[ ply ], -value, ply + 1 );
                    WorkOut[ ply ] := WorkOut[ ply ] + 1;    { Still not finished }
                  end;
                else
                    Score[ ply ] := value;       { No }
                Best[ ply ] := index;
              end; { Update Best }

            if ( Score[ ply ] ≥ Beta[ ply ] ) then   { Cutoff? }
              begin
                Index[ ply ] := ∞;                { Yes.  Force exit from loop }
                interruptSearchers();             { Tell searchers to stop and return }
              end;
            Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );
                                                  { Improve window? }
          end; { Local results in }
      end; { while }
    return( Score[ ply ] );                       { Done. Return best score }
END; { PVSplit }
```

Figure 4.1 - *ParaChinook*'s Principal Variation Splitting (PVSplit)
(Shading indicates key changes from `PVS()`. Boxes group key functionality.)

In Figure 4.1, notice that some new global variables have been defined. The `WorkOut[]` array keeps track of how many searchers are still busy with work rooted at the current split node. PVSplit forces all of the searchers working on the current split node to synchronize before it can back up to the parent node. The new `Value[]` array keeps track of the individual subtree search results. Although the subtrees are farmed out to the searchers in the order determined by `orderMoves()`, they are likely to be of unequal size, and the results will be returned in a different order. Therefore, it is important to keep track of the individual subtree values.

As discussed in Chapter 3, the notion of granularity is important in parallel processing, therefore PVSplit does not recurse to the leaf nodes. Instead, at some threshold number of plies above the leaf nodes, it completes the subtree search with sequential PVS. That way, the work farmed out to searchers when the node is split is at least that minimum granularity in depth.

The heart of PVSplit is the final `while`-loop. The guard condition of the loop is whether there are any locally rooted subtrees still to be searched or whether there are searchers who have not synchronized at the split node by returning their search results. Within the loop are two main blocks of code, highlighted in Figure 4.1 by the use of boxes. The first block of code distributes subtrees to the parallel searchers using `FarmWork()`. Farming work out to a searcher under PVSplit involves finding an idle searcher, packaging the work into a data structure and then signaling the searcher that it has new work. Notice that each of the searches have minimal windows centered on the current best value in `Alpha[ ply ]`. Also, the algorithm can farm out as many concurrent pieces of work as the degree of parallelism allows. If there are more searchers than subtrees at the split node, some of the searchers will starve. The second block of code receives the results returned by searchers using `ReturnedResult()`. It is a non-blocking function call that checks if a search result has been returned by a searcher. If there is no result, control returns to the top of the `while`-loop. If a result is returned, the function returns the index of the work and the body of the `if`-statement is entered. If there is a fail-high situation, that same subtree must be re-searched with a full window, thus the second call to `FarmWork()`. If a cutoff occurs, the searchers are interrupted using `interruptSearchers()` to force a quick synchronization and the loop is immediately exited. Without a cutoff, when all of subtrees have been searched and all searchers have synchronized, PVSplit returns the value of the best subtree.

The implementation of *ParaChinook*'s PVSplit contains two enhancements omitted from the pseudo-code to improve the clarity of the presentation. First, if a search has been farmed out with sub-optimal minimal windows, then that search can be interrupted and re-started with better minimal windows. Specifically, if the value of `Alpha[ ply ]` is increased by a search result, then any other search initiated with a lower value of `Alpha[ ply ]` is interrupted and re-started. Re-starting a minimal window search is different from re-searching with a full window due to a fail-high situation. This action is designed to reduce the search overhead that results from sub-optimal window choice. Second, the root node of the game tree is not a split node as it is with basic PVSplit. Instead, each of the non-principal variation subtrees are searched in sequence with a parallel PVSplit. This is a result of the way sequential *Chinook* is implemented. It is also a pragmatic change to normal PVSplit since a fail-high situation at the root node needs to be detected as quickly as possible, and that is best done by searching each subtree with a parallel PVSplit. Of course, forcing the searchers to synchronize at the root position results in higher idle time due to greater synchronization overhead.

4.2 Experimental Design

To evaluate the strengths and weaknesses of the implementation, a series of experiments were performed. As discussed earlier, the purpose of implementing a parallel algorithm is to search game trees faster, therefore speedup (Formula 3.1) is the primary metric. However, to properly evaluate the overall performance, measurements such as total nodes searched in parallel and time spent at synchronization points were also recorded. The reported total nodes searched include interior and leaf nodes, corresponding to the node count (NC) metric referred to in Formula 2.2. The time spent at synchronization points was measured in real time and converted into the percent of the total search time. The reported percent idle time is the average of all the searchers. It includes the impact of both starvation and stragglers.

The hardware used for the experiments was a BBN TC2000 shared memory multiprocessor [BBN89]. The TC2000 uses a non-uniform memory access (NUMA) architecture via a multi-stage interconnection network (MIN) to implement shared memory. However, *ParaChinook* is not specific to the BBN and could easily be adapted to any shared memory architecture. In fact, *ParaChinook* was originally developed on bus-based shared memory multiprocessors from Silicon Graphics and, through the use of conditional compilation, can be compiled for either machine. All timings were done using the system's real time clock and on a dedicated machine in benchmark mode, with no contention for the processors or for the interconnection network.

The software used for the experiments is the version of *ParaChinook* that competed against Dr. Tinsley in London, August 1992, with some minor changes. The sequential search code is virtually identical to the August 1992 version. One major difference in the software is that the endgame databases, which played such an important role in London, were not used in the parallel search experiments. The databases are stored on disk and the impact of input-output should be factored out from the parallel search performance. Also, because the local software environment of the TC2000 did not allow multiple user processes on the same processor, the controller was placed on a dedicated physical processor. Nonetheless, the speedup graphs are presented as a function of the number of searchers and do not include the additional processor for the controller .

A single transposition table of 8,388,608 (=2^{23}) entries, requiring 128 megabytes of physical memory, was used in all of the sequential and parallel searches. Adding processors to the searcher pool did not increase the size of the transposition table, as is the case with some of the previous research projects (for example, [FeO88] and [FMM91]). None of the sequential searches, and only a few of the parallel searches, exceeded 8 million total nodes, therefore the effect of transposition table collisions was minimized. A larger transposition table was not used because of the limitations of the hardware and because clearing such a large data structure before each test position would have required too much valuable benchmark time. The transposition table was located in shared memory and access to it was coordinated through the use of spin-wait locks. The parallel searchers also shared a single history table. Access to the history table was not protected by locks because the overhead of forcing mutual exclusion was believed to be greater than the disadvantages of losing an update to the history scores.

The test suite used for the experiments consisted of twenty board positions that occurred in the August 1992 match. Appendix B provides the details of the *Tinsley-Chinook 1992 test suite*. All of the speedups, search overheads and idle times reported in this chapters are the average of the individual positions in the test suite. Since there are no standard test suites in computer checkers, such as the Bratko-Kopec suite in computer chess [KoB82], the main goal of the selection process was to fairly represent *ParaChinook*'s speedup performance in a variety of board positions.

In designing a fair test suite of positions, a loose methodology was followed. First, the candidate positions came from actual games played in the August 1992 match. Therefore, none of the positions are synthetic or unlikely to occur in practice. Second, the initial candidate positions were from the 10th move, for both sides, of each of the 39 games of the match. By the 10th move, the differences in opening play have resulted in substantially different board positions. Third, since the game of checkers is predicted to be a draw [SCT92], positions in which one side has an advantage greater than that of a king were rejected, even if the game eventually finished in a draw. Sequential *Chinook* was used to determine the value of the position. This criteria eliminated a large number of positions because it is rare for the board position to be equal throughout an entire game. Fourth, the depth of the search for a position was adjusted until the real time of the sequential search was within the range likely to be seen at tournament time controls. The parallel search was conducted to the same nominal depth. At 20 moves in 60 minutes, the average time spent deciding on a move is 3 minutes, but the actual amount spent per move varies considerably. This fact is reflected in the variety of sequential times in the test suite searches. Fifth, the sequential search to a fixed depth had to choose the same move as actually played in the match game. Considering the high caliber of play in the match, particularly on the part of Dr. Tinsley, the move played in the match is likely to be the best move in the position. Sixth, positions in which the parallel search selected the same move with the same value as the sequential search were *preferred* over other candidates. Different move choices and substantially different move values imply that the parallel and sequential searches are building different game trees. To keep the speedup metric as meaningful as possible, the game trees cannot be overly different.

Many candidate positions were considered and rejected according to the above criteria. Rarely did the sequential and parallel searches choose the same move with the same value, and for every configuration of searchers. Different search extensions are invoked depending on the search window, so the parallel and sequential searches may have different extensions for different lines of play. Also, the parallel search may examine a subtree, that the sequential search would have cut off, and discover it is better. Therefore, search extensions and exploring different parts of the game tree account for the differences in values and move choices. However, if all of these anomalous positions were rejected, there would be fewer than 20 positions in the test suite. In fact, the 10th move of each game did not yield enough test positions, therefore several of the final test suite selections are from different points in the game. Still, since the purpose of the experiment is not to design an ideal test suite, twenty "reasonable" positions were ultimately chosen. The twenty positions in the final test suite represent a compromise between the goals stated above and what was practical given the focus of the thesis.

4.3 Experimental Results: Principal Variation Splitting

Table 4.1 shows the average speedups achieved by PVSplit on the test suite, for 2, 4, 8 and 16 searchers. Appendix C contains the speedups for each test suite position. The best average speedup is 1.92 when using 16 searchers, for a speedup efficiency of 12.0%. Given the diminishing returns for more searchers, larger numbers of searchers were not tested. Since an extra ply of search in checkers requires a two-fold speedup, the increase in the depth of search due to parallelism is less than a single ply.

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.38	0.260	10.1	20.1	17.9	5.25
4	1.79	0.591	16.7	36.0	45.5	6.36
8	1.88	0.548	15.9	35.4	70.1	4.20
16	1.92	0.574	15.3	34.1	84.8	2.18

Table 4.1 — Principal Variation Splitting (PVSplit) Parallel Performance (speedup, search overhead and idle time)

Figure 4.2 shows the corresponding speedup graph. The vertical bars indicate one standard deviation in the average speedup for the 20 test suite positions. The speedup curve quickly levels off after 4 searchers and appears to be nearly flat between 8 and 16 searchers.

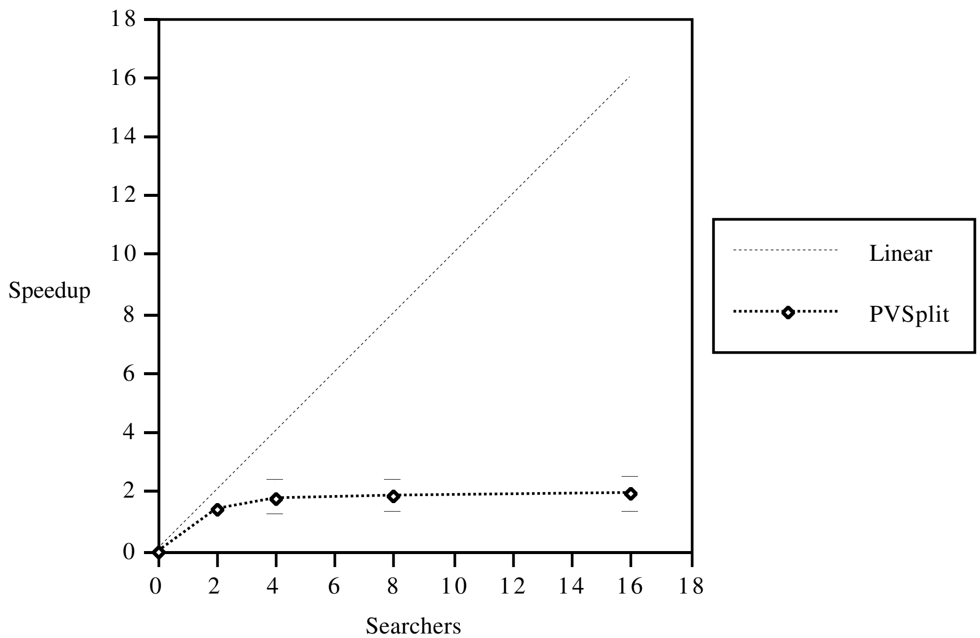


Figure 4.2 — Principal Variation Splitting (PVSplit) Speedups
(vertical bar indicates one standard deviation)

Given that the speedup curves for PVSplit in computer chess (for example, [Sch89a]) are better than in Figure 4.2, why are *ParaChinook*'s speedups not higher? Towards that, consider the average search overhead and percent idle times which are also in Table 4.1.

Figure 4.3 is the corresponding graph of search overhead expressed as a percentage over the sequential game tree and with respect to the number of processors used. The vertical bar indicates one standard deviation. As expected, by not splitting a node until the first subtree has been searched, the search overhead due to information deficiency is minimized. Some search overhead is to be expected because, although the game tree is believed to be strongly ordered, it is not perfectly ordered. Recall that PVSplit will initiate as many concurrent searches as possible when the split node is created. When the first subtree is not best, searches will have to be interrupted as a result of the fail-high situation and the subsequent re-search. Each interrupted search represents search overhead. However, PVSplit does an excellent job of limiting search overhead on the test suite to less than 17% on average.

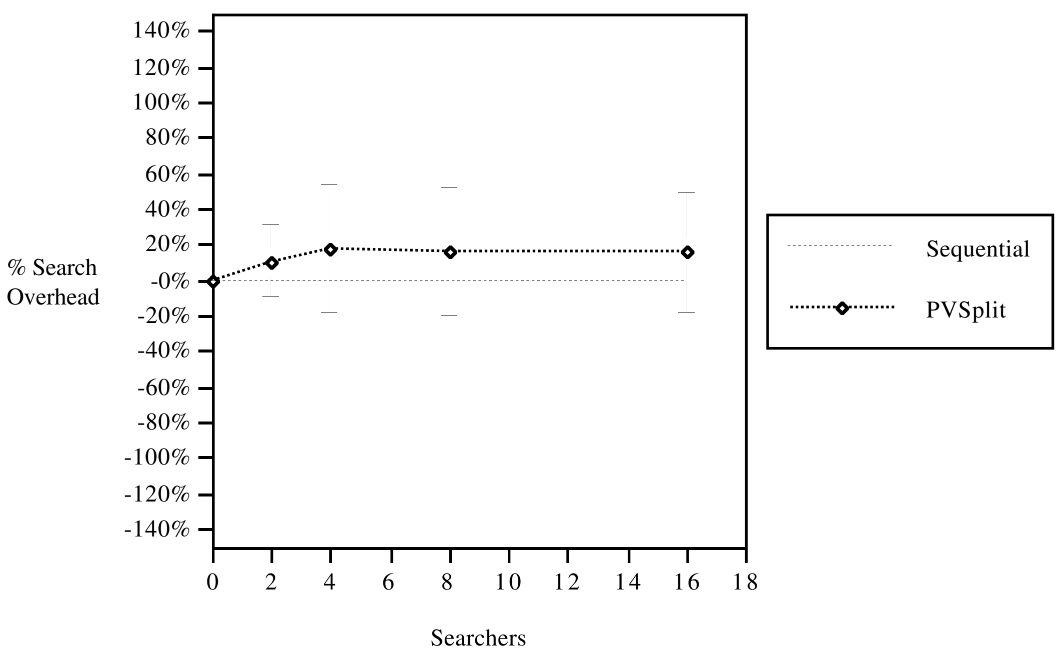


Figure 4.3 — Principal Variation Splitting (PVSplit) Search Overhead
(vertical bar indicates one standard deviation)

Clearly, search overhead cannot account for the poor speedups. In fact, the real reason is revealed in the amount of idle time. The graph of the percent idle time is Figure 4.4.

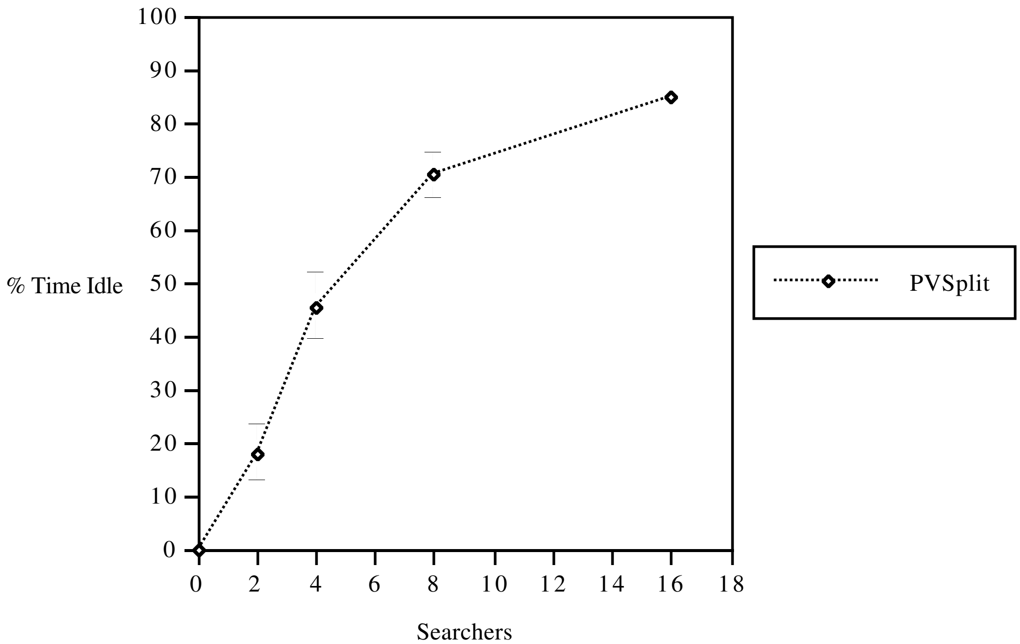


Figure 4.4 — Principal Variation Splitting (PVSplit) Percent of Time Idle
(vertical bar indicates one standard deviation)

As predicted, the low branching factor of narrow checkers game trees results in searcher starvation. As the number of searchers increases, the amount of idle time due to starvation increases dramatically. In the case of 16 searchers, each searcher is idle for an average of 84.8% of the time. Since the average branching factor is 2.84, then with 16 searchers, over 13 of them (i.e. over 82%) are predicted to be starved for work at a split node, on average. In fact, with 16 searchers, the parallelism overload factor is greater than 5. In other words, keeping 16 searchers occupied with work requires at least 5 concurrent split nodes. Otherwise, with such a low degree of parallelism available at each split node, the algorithm is unable to take advantage of more parallel searchers.

Also contributing to idle time is the straggler effect, which increases as the number of searchers and the synchronization fan-in increases. Idle time due to starvation, and to a lesser effect stragglers, is the primary cause of poor speedups for PVSplit.

Figure 4.5 is a different way of looking at the relationship between speedup, idle time and the various parallel overheads. It graphs the percent of real time each searcher contributes to the speedup, the search and concurrency overhead associated with parallelism and the time spent idle due to starvation and stragglers. In effect, Figure 4.5 represents the utilization, or lack of utilization, of the computational capacity of a searcher, normalized by the real time of the search computation.

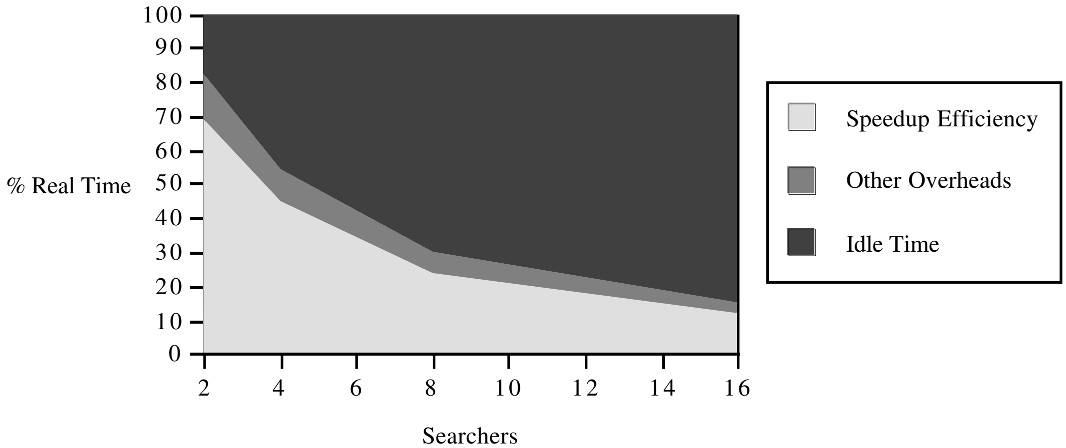


Figure 4.5 — Principal Variation Splitting (PVSplit) Normalized Utilization Profile

The region of Figure 4.5 representing speedup efficiency is computed using Formula 3.2. The region representing idle time is a measured quantity and is the same value as presented in Table 4.1. The region representing other overheads is defined as whatever remains after speedup efficiency and idle time have been accounted for. Both search overhead and communication overhead are included under other overheads.

What Figure 4.5 illustrates is how idle time quickly dominates the utilization profile as searchers are added. PVSplit is conservative in the sense that it allows only one split node at a time and it does not create a split node until the first subtree has been searched. As discussed already, the main rationale for that split node policy is to reduce the amount of information deficiency and consequently reduce the amount of search overhead. However, the figure shows that the trade-off between search overhead and idle time is perhaps too much in favour of reducing search overhead. Although it is important to limit the wasted computation associated with search overhead, the amount of starvation resulting from the single split node policy with a narrow search tree is a greater problem. The experimental data suggests that search overhead should be traded-off for reduced starvation and thus reduced idle time.

Chapter 5

Improving Parallel Chinook

The experimental results show that Principal Variation Splitting's (PVSplit) poor speedups are mainly due to processor starvation. However, the impact of uneven load balancing, due to stragglers, is also a factor in causing the poor speedups. In this chapter, we consider techniques to deal with both the starvation and the load balancing problem. Experimental data is gathered using the Tinsley-*Chinook* 1992 test suite to show the benefits of the new strategies.

5.1 Reducing Starvation: Principal Variation Frontier Splitting

Since the low branching factor of checkers game trees leads to searcher starvation, the issue of how to create sufficient parallel work needs to be addressed. One obvious possibility is to allow multiple split nodes, which increases the degree of parallelism. The *Zugzwang* approach and the DBTE algorithms depend fundamentally on multiple split nodes, but adopting one of those algorithms would require a substantial reworking of *ParaChinook*. If the variation splitting framework of PVSplit is to be maintained, how the extra split nodes are chosen raises some interesting design decisions.

First, since most of the search effort is along the principal variation, the candidate split nodes should probably be on the principal variation as well. More generally, if the algorithm recursively calls itself for non-principal variation lines of play, then the current variation from root to leaf node should contain the candidate split nodes. Dynamic PVSplit (DPVSplit) also recognizes this observation.

Second, the subtrees rooted at the upper nodes of the principal variation represent the coarsest-grained units of work because the subtrees are deeper. In other words, split nodes located closer to the root of the tree contain higher granularity work than split nodes located closer to the leaves. Therefore, if there is no work to be given out at the current split node, then the new split node should be above the ply of the current split node instead of below it. DPVSplit also allows multiple split nodes, but those split nodes are closer to the leaves of the tree and potentially suffer from low granularity.

Third, and lastly, there should be a preference for splitting ALL nodes before CUT nodes. In a strongly ordered game tree, it is likely that the first subtree will generate the cutoff at a CUT node. Therefore, concurrently searching multiple subtrees of a CUT node in an attempt to create more parallel work may simply be creating unnecessary work. However, if given a choice of either splitting a CUT node or allowing searchers to starve, the CUT node could be split.

Based on these observations, a new variation of PVSplit is introduced. *Principal Variation Frontier Splitting* (PVFSplit) supports multiple split nodes along the current variation. The *frontier splitting* technique is analogous to the general notion of the frontier nodes of a tree, where the search is being expanded. The algorithm is a depth-first strategy that addresses the practical issues of work granularity and the ALL/CUT node structure of the Alpha-Beta game tree. Consequently, a newly created split node is located on the current variation, it is closer to the root of the game tree than the current split node and splitting ALL nodes is preferred to splitting CUT nodes.

The impact of frontier splitting is two-fold. First, searchers do not need to starve if the parallel work at a split node is exhausted. A new split node with more work may be created. Second, as a positive side effect, frontier splitting reduces the effective synchronization fan-in at split nodes. By the time the last search result of a split node is returned, the other searchers are, ideally, already working at a different split node instead of waiting to synchronize. However, strictly speaking, frontier splitting does not completely eliminate synchronization fan-in. Because the split nodes can only exist on the current variation, the processors must still synchronize at the root node. Rather than synchronizing at each split node, as is the case of PVSplit, the root node is the only synchronization point under PVFSplit. Of course, even this synchronization requirement can be eliminated if one wishes to allow multiple variations from the root position to be searched in parallel. The current version of *ParaChinook* does not allow multiple variation splitting.

The biggest drawback of PVFSplit is that the searches rooted at the new split node may have to be initiated without the benefit of the bounds information returned by the first subtree. PVFSplit trades off the potential of more search overhead for the potential benefits of reduced starvation through multiple split nodes. Without bounds information, we are faced with either allowing a full window search of the subtrees or we can use heuristics to create an artificial minimal window. In the current implementation of PVFSplit, a minimal window is heuristically created for all split nodes greater than 7 plies from the leaf nodes. The best score at the deepest split node is the basis of the minimal window. If no such score is available, then the value of the current subtree from the last iteration of iterative deepening is used.

Figure 5.1 contains the pseudo-code for the Principal Variation Frontier Splitting algorithm. The shading highlights the main differences between PVSplit and PVFSplit.

The `Value[]` array is now two-dimensional since each node of the current variation could potentially be a split node under PVFSplit. Some new local variables (`newPly`, `heurAlpha`, `heurBeta` and `pNew`) are also defined.

The call to `isAnALLNode()` before assigning work from the current split node makes explicit what was implicit in the pseudo-code and discussion of PVSplit: if the current variation is not the principal variation, it may contain CUT nodes. Therefore, if the current split node is an ALL node, then the subtrees are farmed out to parallel searchers. If the current split node is a CUT node, then each subtree is searched in sequence with a recursive call to PVFSplit.

A new body of pseudo-code deals with choosing and assigning work from a new split node before PVFSplit can recurse out of the current split node. This is the key to allowing multiple split nodes on the current variation. The function `GetOtherSplitNode()` selects the split node from among the candidate nodes along the current variation. If the current split node is at ply n, then the nodes at plies n - 1, n - 2, n - 3, etc. back to the root are in decreasing order of preference. Another consideration is whether the node is an ALL node or a CUT node. Every ALL node on the current variation must already be split and exhausted of work before a CUT node is selected. But if the alternative is starvation, a CUT node is selected to be a split node. Once the new split node is selected, `SynthMinWindow()` is used to decide on the search window to use for the search. As discussed earlier, full window searches are only allowed within 7 plies of the leaf nodes. Otherwise, a heuristic minimal window is synthesized.

One important aspect of *ParaChinook*'s implementation of PVFSplit not included in the pseudo-code, to improve its clarity, deals with cutoffs. It is possible for a cutoff to occur at a split node that is closer to the root node than the current split node. Such a condition is detected and PVFSplit immediately backs up to the split node where the cutoff occurred. This phenomenon is an important advantage of PVFSplit over PVSplit because that cutoff would not have been detected until the algorithm, after much more searching, systematically backed up to that higher node.

```
VAR { Changed Global Variable }
    Value     :    array[ 1..MAX_DEPTH ] of  { Results of each subtree  at each ply }
                   array[ 1..MAX_WIDTH ] of integer;
FUNCTION
    PVFSplit( p : position; alpha, beta, ply : integer ) : integer;

VAR
    index, value : integer;                        { Index and value of current move }
    move : moveType;                               { Current move being searched }
    newPly : integer;                              { Ply of new split node with work }
    heurAlpha, heurBeta : integer;                 { Window for new split node }
    pNew : position;                               { Position of new split node }

BEGIN { PVFSplit }

{ Check if at threshold for minimum granularity.  Do not recurse to leaves. }
    if ( ply = ( TreeDepth - MinGranularity ) ) then
        return( PVS( p, alpha, beta, ply ) );   { Yes.  Do sequentially. }

{ Generate legal moves and order them }
    Width[ ply ] := legalMoves( p, Moves[ ply ] );      { Store moves in Moves[] }
    if ( Width[ ply ] = 0 ) then                   { Any legal moves in this position? }
        return( Evaluate( p ) );                   { No.  In checkers, this is a loss }
    orderMoves( Moves[ ply ] );                    { Yes. Do move ordering }

{ Set up global variables }
    Index[ ply ] := 2;                             { Continue with other subtrees, later }
    Alpha[ ply ] := alpha;
    Beta[ ply ] := beta;
    WorkOut[ ply ] := 0;                           { Search results still outstanding }

{ Search first ordered move with full window. Unroll the loop by one }
    Score[ ply ] :=
        -PVFSplit( p.Move[ ply ][ 1 ],
                        -Beta[ ply ], -Alpha[ ply ], ply + 1 );
    Value[ ply ][ 1 ] := Score[ ply ];
    Best[ ply ] := 1;

    if ( Score[ ply ] ≥ Beta[ ply ] ) then   { Cutoff after pv? }
        Index[ ply ] := ∞;                         { Yes.  Will never enter loop }
    Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );       { Improve window? }

{ Search each sibling subtree with minimal window }
{ Split node incomplete if there are local subtrees unsearched or if there are searchers still to return }
    while ( ( Index[ ply ] ≤ Width[ ply ] ) OR ( WorkOut[ ply ] > 0 ) ) do
      begin { while }
        if ( "∃idle searcher" AND Index[ ply ] ≤ Width[ ply ] ) then
          begin { Local work out }                 { Search each local subtree }
            index := Index[ ply ];                 { An unique move leads to the subtree }
            move := Moves[ ply ][ index ];
            Index[ ply ] := Index[ ply ] + 1;  { Advance index }

            if ( isAnALLNode( p ) ) then      { Respect ALL/CUT node structure }
              begin { Local ALL node }
                Value[ ply ][ index ] :=       { Search in parallel }
                    -FarmWork( p.move,         { Negamax value and minimal window }
                        -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );
                WorkOut[ ply ] := WorkOut[ ply ] + 1;    { Keep track of work given }
              end { Local ALL node }
            else
                Value[ ply ][ index ] :=       { Search in sequence }
                    -PVFSplit( p.move,         { Negamax value and minimal window }
                        -Alpha[ ply ] - 1, -Alpha[ ply ], ply + 1 );
          end; { Local work out }
```

{ While-loop continuation }

```
    else if ( "∃idle searcher" ) then   { If no more local work }
      begin { Non-local work out }      { ...give work from other split node }
        newPly := GetOtherSplitNode( pNew );     { Magical function }
        index := Index[ newPly ];       { An unique move leads to the subtree }
        move := Moves[ newPly ][ index ];
        Index[ newPly ] := Index[ newPly ] + 1; { Advance index }

        SynthMinWindow( pNew, heurAlpha, heurBeta ); { Might use min window }
        Value[ newPly ][ index ] :=     { Negamax value and minimal window }
            -FarmWork( pNew.move,       { pNew is the  split node }
                -heurBeta, -heurAlpha, newPly + 1 );
        WorkOut[ newPly ] := WorkOut[ newPly ] + 1;{ Keep track of work given }
      end; { Non-local work out }
```

```
    else if ( index := ReturnedResult() ) then
      begin { Local results in }
        value := Value[ ply ][ index ];
        WorkOut[ ply ] := WorkOut[ ply ] - 1;   { Work received }

        if ( value > Score[ ply ] ) then { Score improved, but did it fail high? }
          begin { Update Best }
            if ( value > Alpha[ ply ] and value < Beta[ ply ] ) then
              begin
                Value[ ply ][ index ] := { Yes.  Re-search  }
                    -FarmWork( p.move, -Beta[ ply ], -value, ply + 1 );
                WorkOut[ ply ] := WorkOut[ ply ] + 1;   { Still not finished }
              end;
            else
                Score[ ply ] := value;    { No }
            Best[ ply ] := index;
          end; { Update Best }

        if ( Score[ ply ] ≥ Beta[ ply ] ) then  { Cutoff? }
          begin
            Index[ ply ] := ∞;            { Yes.  Force exit from loop }
            interruptSearchers();         { Tell searchers to stop and return }
          end;
        Alpha[ ply ] := MAX( Alpha[ ply ], Score[ ply ] );
                                          { Improve window? }
      end; { Local results in }
  end; { while }
  return( Score[ ply ] );                 { Done. Return best score }
END; { PVFSplit }
```

Figure 5.1 - *ParaChinook*'s Principal Variation Frontier Splitting (PVFSplit)
(Shading indicates key changes from `PVSplit()`. Boxes group key functionality.)

5.2 Experimental Results: Principal Variation Frontier Splitting

Table 5.1 shows the average speedups achieved by PVFSplit on the test suite for 2, 4, 8 and 16 searchers. Appendix C contains the detailed experimental data. The best average speedup is 2.91, achieved with 16 searchers for a speedup efficiency of 18%. Although this is a 52% improvement in speedup over PVSplit, it is still low in absolute terms as reflected by the low speedup efficiency. However, a speedup of 2.91 does translate into more than one additional ply of search in the same amount of time [SCT92].

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.48	0.325	8.76	17.4	12.8	4.90
4	2.03	0.546	24.3	26.3	30.8	8.17
8	2.71	0.803	46.4	46.3	48.8	7.74
16	2.91	1.07	83.5	63.5	67.1	7.00

Table 5.1 — Principal Variation Frontier Splitting (PVFSplit) Parallel Performance (speedup, search overhead and idle time)

Figure 5.2 is the speedup graph for PVFSplit, with a vertical bar representing one standard deviation. As a basis for comparison, the speedup curve for PVSplit is reproduced. Although PVFSplit increases the absolute speedup numbers, it does not change the basic shape of the curve. As searchers are added, the speedup curve flattens to reflect the diminishing returns.

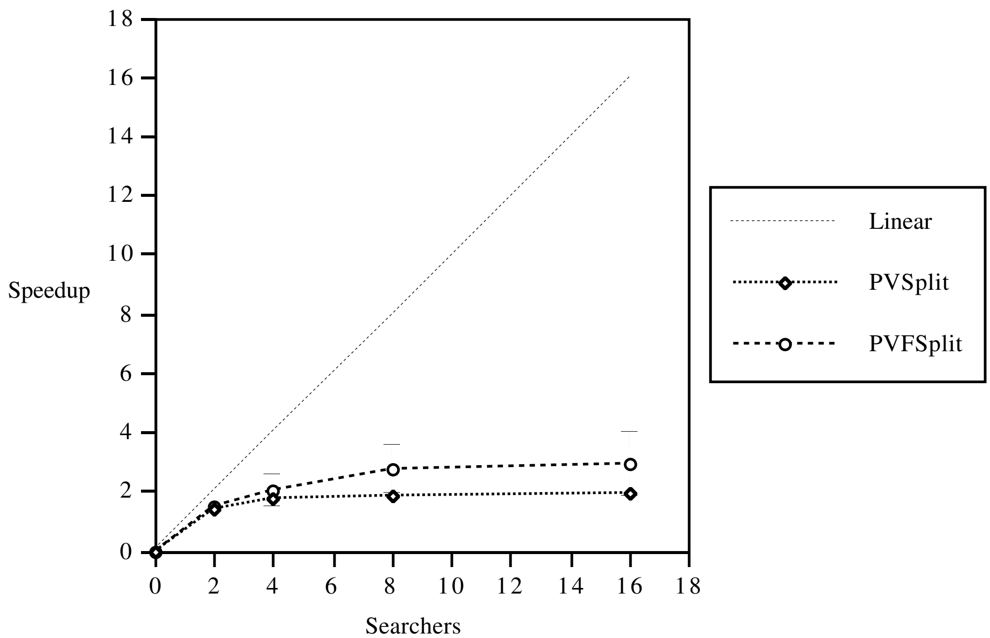


Figure 5.2 — Principal Variation Frontier Splitting (PVFSplit) Speedups
(vertical bar indicates one standard deviation)

With PVSplit, there is a trade-off between having low search overhead and high idle time. An algorithm such as PVFSplit makes the reverse trade-off and increases search overhead in order to reduce idle time.

Inherent in a multiple split node algorithm such as PVFSplit is the risk of higher search overheads due to information deficiency regarding cutoffs and window bounds. In Figure 5.3, we can see that PVFSplit, by being more aggressive than PVSplit in splitting nodes, also suffers from significantly higher search overhead. Of course, the bottom line is that despite the higher search overhead, PVFSplit's speedup is higher.

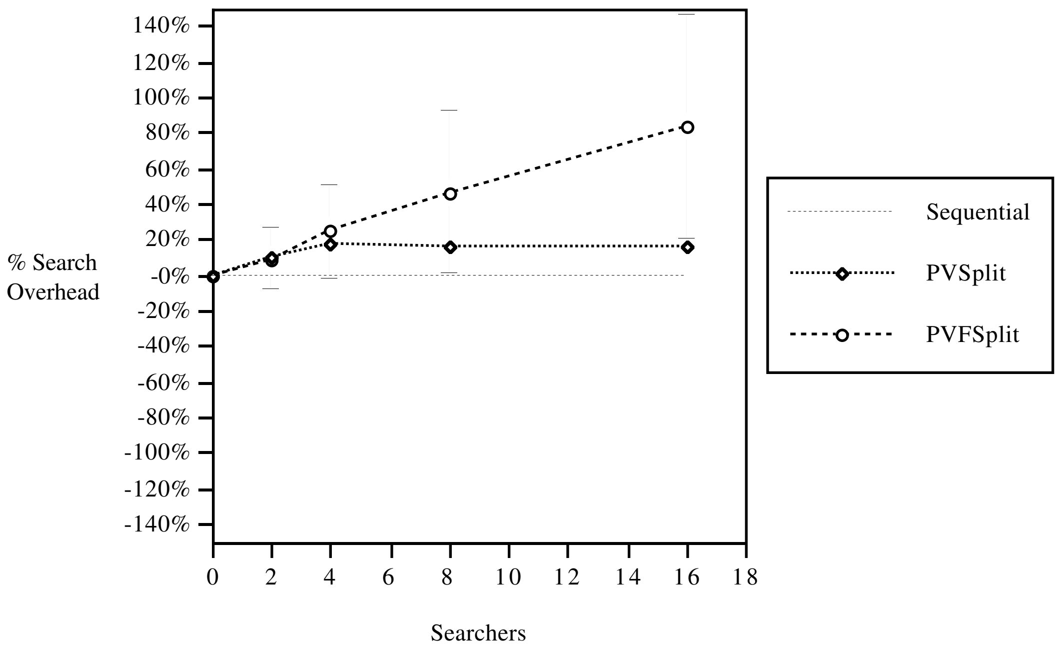


Figure 5.3 — Principal Variation Frontier Splitting (PVFSplit) Search Overhead
(vertical bar indicates one standard deviation)

Interestingly, the search overhead of PVFSplit is dramatically higher than PVSplit, the idle time of PVFSplit is modestly lower, as seen in Figure 5.4, but the speedup is still greater for PVFSplit. This suggests that the trade-off of higher search overhead for reduced idle time is better overall since speedup increases.

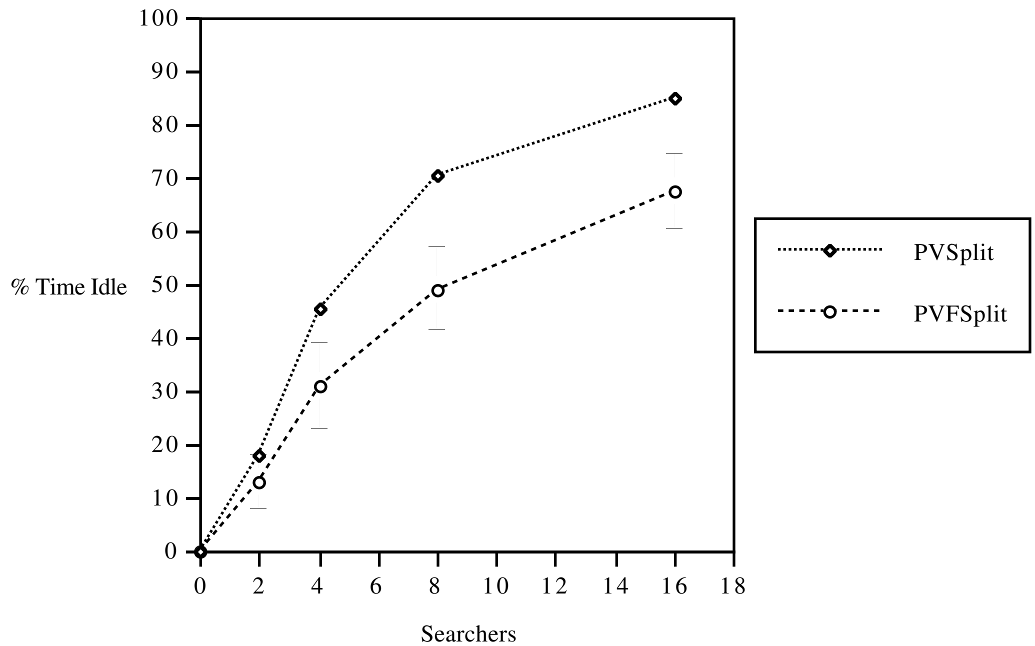


Figure 5.4 — Principal Variation Frontier Splitting (PVFSplit) Percent of Time Idle
(vertical bar indicates one standard deviation)

Despite the success in trading off search overhead for reduced idle time in PVFSplit, the amount of idle time still remains high. Referring to Figures 4.5 and 5.5, the idle time region of the utilization profile has decreased in size while both the speedup efficiency and other overheads regions have grown.

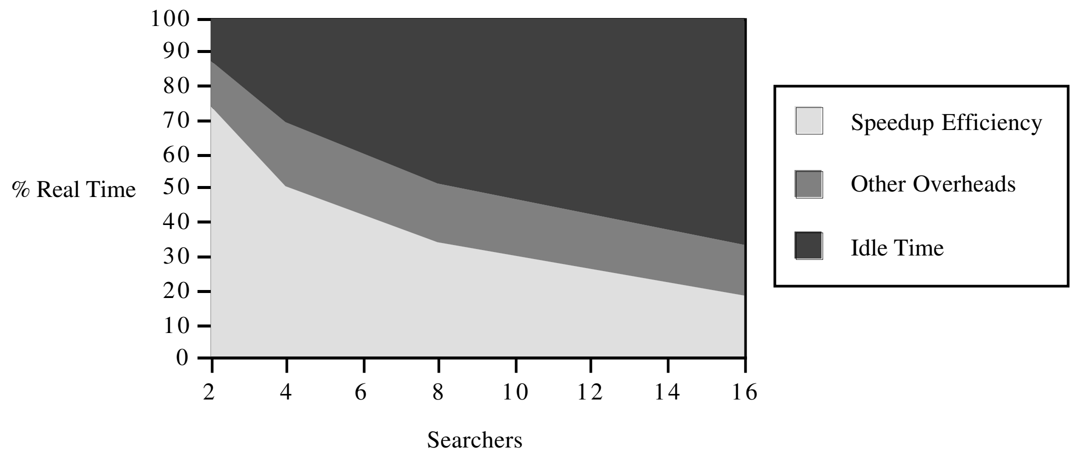


Figure 5.5 — Principal Variation Frontier Splitting (PVFSplit) Normalized Utilization Profile

Reduced starvation through multiple split nodes and reduced synchronization fan-in still leaves a lot of idle time. Although the split node and synchronization policies of PVFSplit can be further enhanced, the fact that search overhead grows as the synchronicity decreases implies that the overall speedup and searcher utilization properties cannot be changed. Even if the need to synchronize at the root position is eliminated, the corresponding growth in search overhead will ensure diminishing returns for additional searchers. Furthermore, the programming and software engineering effort required to debug and maintain a parallel search with so much asynchronous behaviour would be prohibitive.

Despite the reduction in idle time with PVFSplit, it is still rather high at 67.1% with 16 searchers. What is causing the high idle time? First, although frontier splitting allows PVFSplit to avoid synchronizing at each split node, there is still a need to synchronize at the root of the game tree before the next variation an be searched. This synchronization occurs for each subtree at the root position, and for each iteration in the iterative deepening process. One can easily envision a different PVFSplit in which the need to synchronize at the root node is removed, but that enhancement is not implemented yet in *ParaChinook*. Second, the number of candidate split nodes only grows linearly with respect to the depth of the tree and, therefore, there may insufficient nodes on the current variation to keep all of the searchers busy. Recall that with a parallelism overload factor of 5, which is the case with 16 searchers, there needs to be 5 concurrent split nodes at all times in order to prevent starvation. Third, the impact of stragglers has not been addressed yet. When searchers have to wait for a straggler, the result is also idle time.

5.3 Improving Load Balancing: Straggler Preemption

As discussed earlier, the amount of search effort associated with each unit of work varies. Consequently, there exists a load balancing problem when some searchers are given more work than others. Searchers that force other searchers to wait at synchronization points, often because they have been given a larger piece of work, are referred to as *stragglers*. Stragglers cause synchronization overhead that appears as higher idle time.

Clearly, if there is a load balancing problem, the search effort should be re-balanced. The approach taken with Dynamic PVSplit (DPVSplit) [Sch89a] is to assign idle searchers to help the busy searchers. A simpler but more inefficient solution would be to interrupt the straggler and then apply PVFSplit to the subtree that was previously assigned to the slow searcher. PVFSplit would use the idle searchers, which now includes the straggler, to search the subtree in parallel. In essence, the subtree that caused the straggler is preempted and subdivided so that it can be searched in parallel. This is similar to the “shoot-down” procedure of the *Waycool* chess program [FeO88].

The main disadvantage of the interrupt-based approach is that the effort already invested into searching the subtree is largely lost. An interrupted search is incomplete and the returned result is useless. With DPVSplit, the straggler is not interrupted, so the parts of the subtree that have already been searched are not lost. Still, with the help of a transposition table, the entire computation of the straggler is not lost when it is interrupted. The parts of the subtree that have been completed can be found in the transposition table. Still, the interrupt-based scheme is clearly less efficient and less elegant than DPVSplit. Its main strength is that of simplicity of implementation.

Simplicity was also the goal in designing the definition of what constitutes a straggler. Clearly, it would be inefficient to wait until all of the searchers except for one, the obvious straggler, are idle. Therefore, when a quorum of searchers is idle for longer than a threshold length of time, the remaining searchers who are still computing become candidate stragglers. One of them is selected, interrupted and its subtree is searched in parallel. Currently, when more than a quarter of the searchers are idle for longer than 5 seconds, then the searcher who has been searching its assigned subtree for the longest time is selected for preemption. The quorum definition, time threshold and straggler choice have been derived from informal experiments. It has been observed that a small quorum and short time thresholds result in aggressive load balancing, which can be counter-productive if the overhead of interrupting the straggler is not outweighed by the potential benefits.

Although the straggler preemption technique described above is simple, it does appear to be effective in improving the parallel performance of both PVSplit and PVFSplit. More sophisticated load balancing techniques may be able to further improve the speedups, but as the technique grows in sophistication, so does the overhead of executing it. Still, there is room for further exploration of load balancing strategies in the future.

5.4 Experimental Results: Straggler Preemption

Table 5.2 presents the speedup performance of PVSplit with the simple load balancing strategy for stragglers. To distinguish it, PVSplit with the load balancing procedure is referred to as PVSplit-LB.

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.44	0.346	11.3	23.0	13.0	2.86
4	1.89	0.567	22.3	35.7	37.4	3.69
8	2.06	0.539	21.3	34.6	63.6	3.23
16	2.00	0.530	24.2	34.1	81.4	1.66

Table 5.2 — Principal Variation Splitting with Load Balancing (PVSplit-LB) Parallel Performance (speedup, search overhead and idle time)

For each configuration of searchers, PVSplit-LB achieves marginally higher speedups than PVSplit by reducing the idle time due to synchronization overhead. Given the number of idle searchers with PVSplit, the load balancing scheme is likely to be active in preempting the searchers who have work to reduce starvation. Load balancing and starvation prevention are closely related concepts. However, frontier splitting tries to reduce starvation by acting *a priori,* before idle time accumulates, and straggler preemption tries to balance the work *a posteriori*, after already incurring idle time. Consequently, the performance improvement due to frontier splitting is greater than the improvement due to straggler preemption.

Interestingly, the speedup declines (marginally) between 8 and 16 searchers. It is possible that starvation is forcing the load balancing mechanism to frequently interrupt searchers, incurring the overheads already discussed and decreasing overall performance.

Having both starvation prevention and load balancing is better than either strategy by itself. Table 5.3 shows the improved speedup values for Principal Variation Frontier Splitting with load balancing (PVFSplit-LB). PVFSplit-LB achieves the highest performance observed so far, with an average speedup of 3.32 with 16 searchers.

Searchers	Speedup		% Search Overhead		% Time Idle	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
2	1.53	0.402	11.8	23.0	8.41	2.57
4	2.18	0.491	27.0	24.5	23.8	5.61
8	2.80	0.779	59.8	45.0	40.8	6.58
16	3.32	1.01	70.9	41.4	62.5	5.67

Table 5.3 — Principal Variation Frontier Splitting with Load Balancing (PVFSplit-LB) Parallel Performance
(speedup, search overhead and idle time)

The speedup curves for PVSplit-LB and PVFSplit-LB are in Figure 5.5. Note that the average speedup for PVFSplit-LB is always higher than the average speedup for PVSplit-LB.

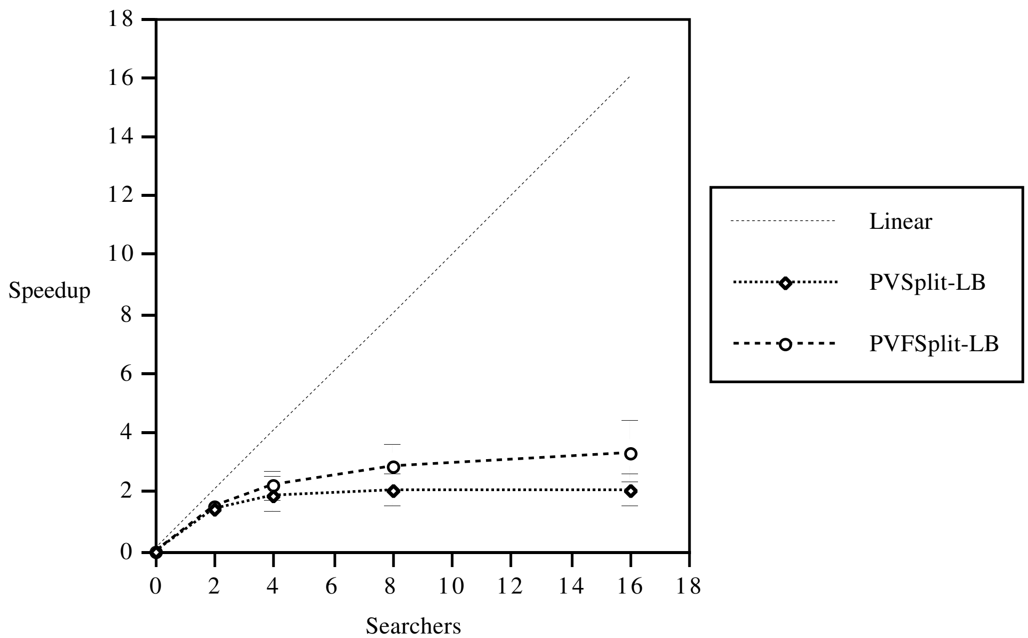


Figure 5.5 — PVSplit and PVFSplit with Load Balancing Speedups
(vertical bar indicates one standard deviation)

As the best performing algorithm of these experiments, PVFSplit-LB's utilization profile, given in Figure 5.6, is worth examining. Although the idle time portion still remains a significant part of the profile, it is substantially smaller than in the case of PVSplit without load balancing (Figure 4.5).

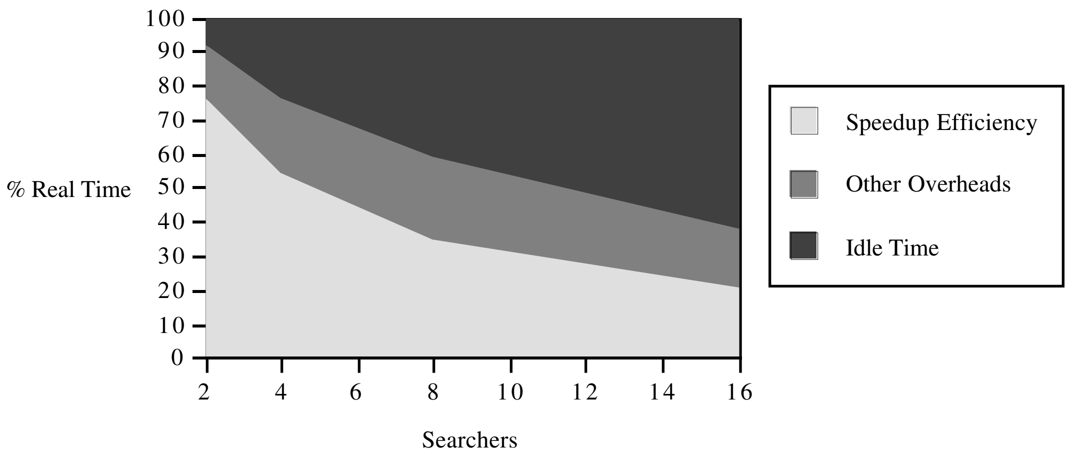


Figure 5.6 — Principal Variation Frontier Splitting (PVFSplit-LB) with Load Balancing Normalized Utilization Profile

Frontier splitting and straggler preemption have been effective in increasing the speedup efficiency of the parallel search by trading off idle time for more speedup and getting higher search overhead as a negative side effect. Since neither search overhead nor idle time can be eliminated completely, PVSplit-based algorithms will always suffer from diminishing returns.

5.5 Summary of Experimental Results: All Algorithms

A total of four related algorithms have been evaluated. Principal Variation Splitting (PVSplit) is the basis of all of the algorithms. Because PVSplit suffers from starvation due to narrow game trees with low branching factors, Principal Variation Frontier Splitting (PVFSplit) was introduced to increase the average degree of parallelism. Because both PVSplit and PVFSplit suffer from load imbalances due to stragglers, a simple but useful straggler preemption strategy was implemented and evaluated.

In Figure 5.7, we can see the speedup curves for all four algorithms. Although frontier splitting and load balancing give PVFSplit-LB the best speedup curve, it remains in the same diminishing returns shape as PVSplit.

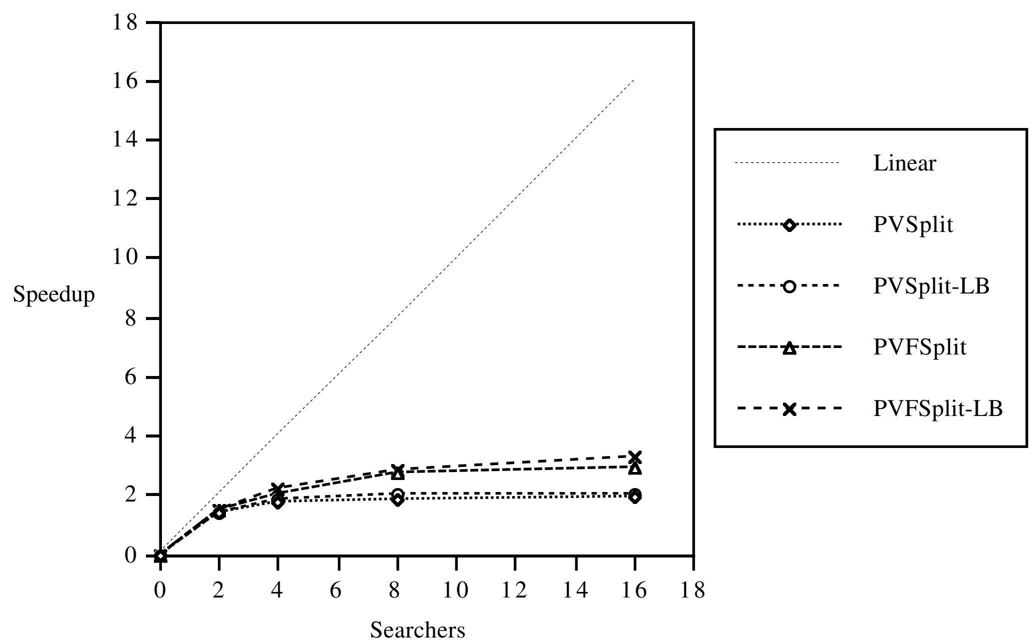


Figure 5.7 — All Algorithms Speedups
(higher curves are better)

Figure 5.8 shows the same speedup information as Figure 5.7, but with unequal axes ranges, in order to show a close-up of the speedup curves.

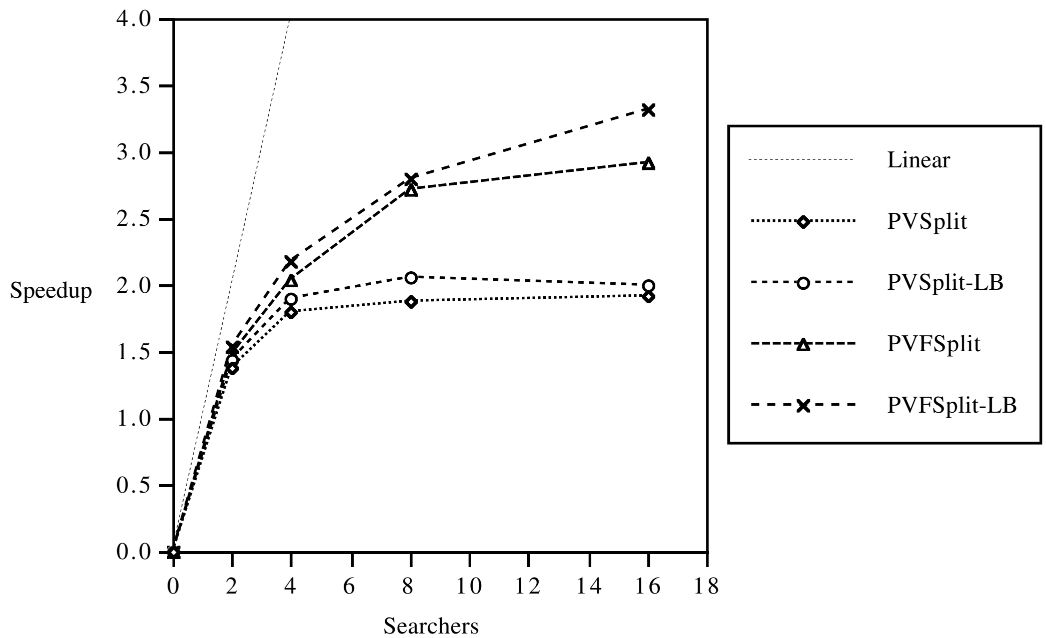


Figure 5.8 — All Algorithms (Close-Up) Speedups
(higher curves are better)

For completeness, Figures 5.9 and 5.10 show the search overhead and idle time graphs of all four algorithms.

The only anomaly occurs when the search overhead curves of PVFSplit and PVFSplit-LB cross over in Figure 5.9. Since interrupting a straggler results in the loss of partial search results, it is not surprising that for certain configurations of searchers, PVFSplit-LB has more search overhead than PVFSplit. However, it is speculated, interrupting a straggler and preempting its subtree typically means that the subtree is searched more quickly and the search result may be used to improve the parameters (i.e. windows) of other searches in progress sooner, thus reducing search overhead through reduced information deficiency.

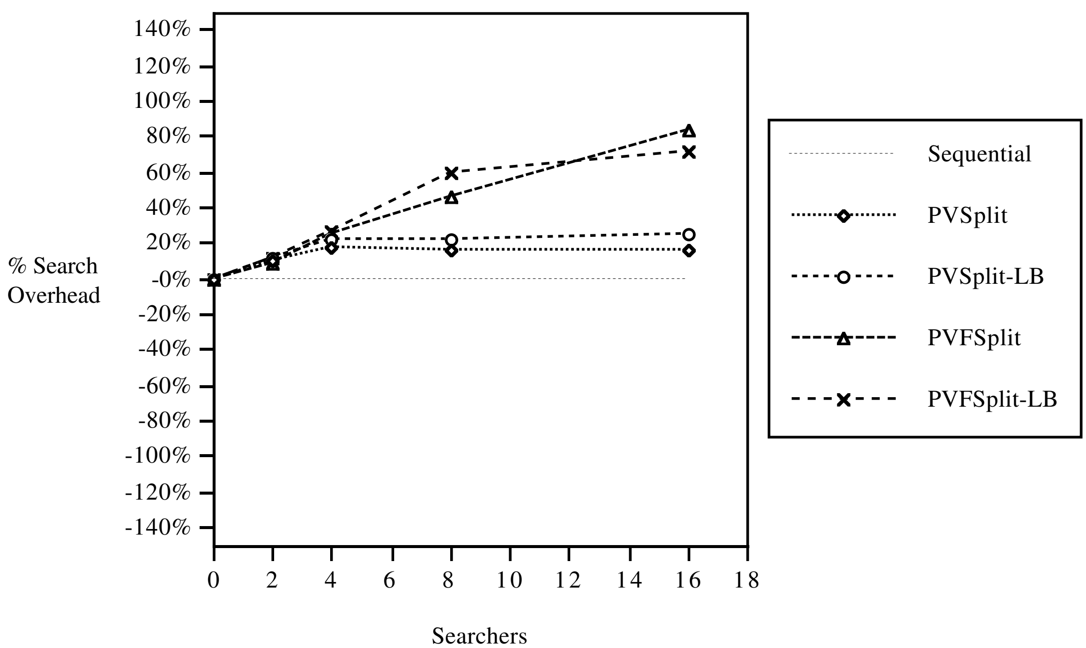


Figure 5.9 — All Algorithms Search Overhead
(lower curves are better)

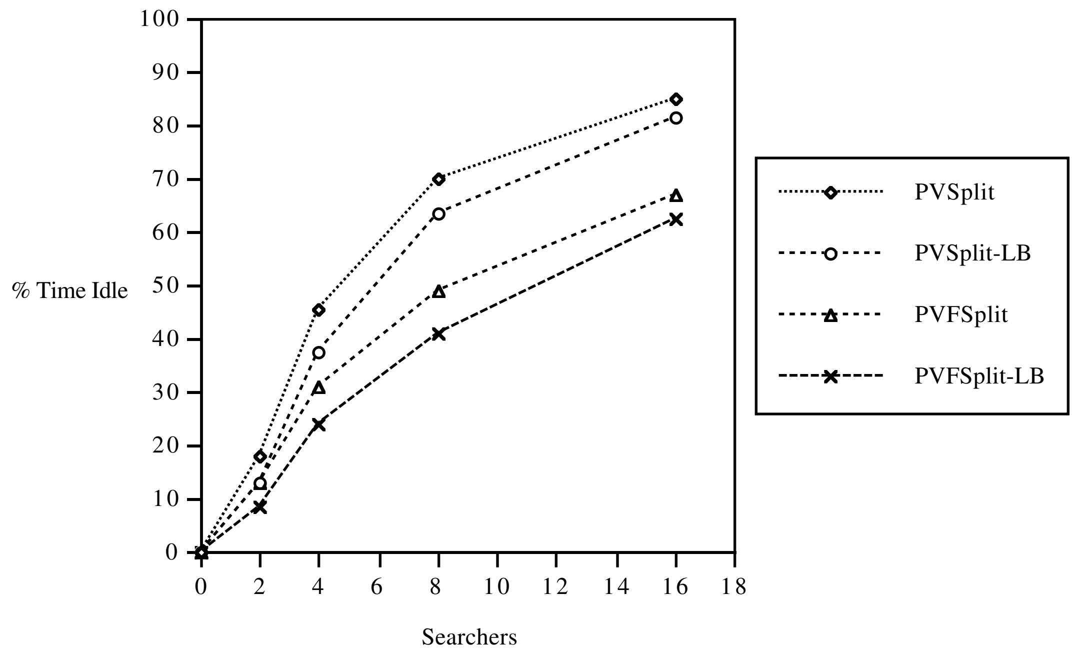


Figure 5.10 — All Algorithms Percent of Time Idle
(lower curves are better)

5.6 Concluding Remarks

From these experimental results, we can conclude that frontier splitting is an useful technique to reduce the starvation of searchers. Furthermore, the need for a load balancing strategy to deal with stragglers is also clear.

In retrospect, arguing that PVFSplit, both with and without load balancing, is better than PVSplit is somewhat unfair. Principal Variation Splitting was originally intended to form the *basis* of parallel search algorithms and does not address application-dependent issues such as game tree branching factor and stragglers. Nonetheless, the poor performance of PVSplit on narrow checkers game trees emphasizes the close relationship between branching factor and the potential speedup. Also, the experimental results point out that some of the successes of computer chess research cannot be directly adopted in other, closely related domains.

Chapter 6

Conclusions

The first implementation of a parallel *Chinook* demonstrated how closely the game tree branching factor is linked with the speedups that can be obtained with tree decomposition algorithms, such as Principal Variation Splitting (PVSplit). The narrow checkers game trees have a branching factor that is over an order of magnitude smaller than in chess game trees. With a branching factor of 2.84, the degree of parallelism offered by checkers game trees is quickly overloaded by even a moderate number of parallel searchers. For 16 searchers, there are over 5 times as many searchers as there are units of parallel work available at a split node. Consequently, PVSplit suffers from starvation.

To reduce the amount of starvation, Principal Variation Frontier Splitting (PVFSplit) was developed. PVFSplit attempts to create more parallel work by allowing each node on the current variation to be a split node. Experiments with PVSplit and PVFSplit on a test suite of positions, using a BBN TC2000 shared memory parallel computer, show that PVFSplit achieves 52% better speedups than PVSplit (with 16 searchers) by lowering the idle time associated with starvation. Additional support for load balancing through straggler preemption further improves the speedup to a high of 3.32 with 16 searchers. In absolute terms, the speedups achieved with PVFSplit are low.

The experience of developing *ParaChinook* has resulted in three main observations. First, a relatively high branching factor in the application domain cannot be taken for granted. One can choose the application, but one usually cannot choose the properties of that domain. Any discussion of parallel performance must be put in the context of the natural degree of parallelism of the application domain. To present speedups without mentioning the branching factor is to present a partial picture. Second, it is important to note how many of the subtrees rooted at a node require a substantial amount of effort to search. Some of the branches may represent poor move choices that lead to quick refutations. These subtrees are considerably smaller than the other more viable subtrees, and thus contain less work. It is speculated that for checkers, there are about 2 viable moves per node on average, even if the actual branching factor is, say, 8. A high branching factor does not guarantee large amounts of parallel work. Third, any PVSplit-based will suffer from the same trade-off between starvation and search overhead due to the branch-and-bound nature of Alpha-Beta search. Furthermore, there are a limited number of split nodes available in a single variation, therefore a future version of *ParaChinook* may have to use a non-depth first search strategy to find enough split nodes to get acceptable speedups.

6.1 A Future *ParaChinook*: Ideas and a Paper Design

Experimental results show that the current version of *ParaChinook* cannot profitably use more than about 8 searchers before the speedup curve begins to level off. Given the trend towards more processors in parallel computers, *ParaChinook* has to be updated too. Even with PVFSplit, the main reason for poor speedups is searcher starvation.

However, PVFSplit should not be abandoned altogether. On the one hand, the depth first search nature of PVFSplit is appealing. Its single controller is easier to debug than a fully distributed control structure and the low control-related storage requirement[7] is an important advantage when the total node count of the game trees is in the millions. On the other hand, a single variation splitting framework cannot provide enough split nodes to eliminate starvation when the trees are narrow. A compromise on pure depth first search *may* be sufficient to achieve acceptable performance.

The storage overhead of best first search and other non-depth first search strategies can be reduced if only the top few plies of the search tree are expanded in this manner. Recall that with the DBTE algorithms, the nodes of top plies of the game tree are queued in memory and are given to a pool of parallel searchers. A similar processor model can be adopted within *ParaChinook*.

The individual searchers can be clustered into groups of 4 or 8 with a single controller per cluster. The main *ParaChinook* program builds the upper plies of the search tree in memory in a non-depth first manner so that many split nodes are generated. A subtree is given to a cluster of searchers who search it to the nominal depth using PVFSplit. Because the subtrees given to the clusters of searchers are near the root of the tree, they are coarse-grained units of work, which will result in better parallel performance. Because the clusters of searchers build

[7] Of course, data structures such as transposition tables do require large amounts of storage. However, they are supporting data structures and are not essential to the control aspects of the tree search.

the last several plies of the tree depth first, the low control-storage requirements of the current *ParaChinook* is largely maintained. Furthermore, clustering the searchers reduces the number of entities contending for a concurrent data structure, such as a work queue. In effect, a cluster of searchers executing PVFSplit becomes a parallel static evaluation function from the point of view of the main *ParaChinook*.

The hybrid design may not be the best choice for a new *ParaChinook*, but it is clear that small changes to the basic PVSplit algorithm, in the same vein as PVFSplit, will be insufficient to improve the speedups on narrow game trees. However, PVFSplit with a small number of searchers can be an useful building block for other parallel search algorithms.

6.2 Final Comment: Quality Versus Quantity

The method of quantifying performance in this thesis has been the parallel speedup. As already touched upon, speedup has both advantages and disadvantages as a performance metric. It is simple to understand, but its simplicity does not capture other important aspects of performance.

For example, although a speedup of 3.32 with 16 searchers is low in absolute terms, it does translate to over 1.5 extra plies of search depth in the same amount of time. Furthermore, since the branching factor is 2.84, the speedup value is greater than the theoretical ideal speedup with PVSplit: speedup equal to the branching factor.

The ideal performance metric would be the improvement in the quality of move choice that comes with parallelism. Since the goal is to improve the game playing program's strength, the decision quality should be the bottom line. Towards that, the potential of solving the game of checkers and creating a perfect checkers oracle is intriguing. Using the endgame databases, a future version of *Chinook* may be able to judge the quality improvement that results from parallelism. However, that remains a future research project.

Bibliography

[ABD82] S. Akl, D. Barnard and R. Doran, "Design, Analysis and Implementation of a Parallel Tree Search Algorithm," *IEEE Trans. on Pattern Anal. and Mach. Intell. 6*(4), 361-371 (1975).

[Bau78] G. Baudet, The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, 1978.

[BBN89] BBN Advanced Computers Inc., Inside the TC2000, Cambridge, MA, (1989).

[BeE89] H.J. Berliner and C. Ebeling, "Pattern Knowledge and Search: The SUPREM Architecture," *Artificial Intelligence 38*(2), 161-198 (1989). A revised version appears as [BeE90].

[BeE90] H.J. Berliner and C. Ebeling, "Hitech" in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 79-109.

[Cam81] M. Campbell, Algorithms for the Parallel Search of Game Trees, M.Sc. thesis, Tech. Rep. TR81-8, Dept. of Computing Science, Univ. of Alberta, Edmonton, 1981.

[CoT82] J.H. Condon and K. Thompson, "Belle Chess Hardware" in M. Clarke, ed., *Advances in Computer Chess 3*, Pergamon Press, Oxford, 1982, 45-54.

[CoT83] J.H. Condon and K. Thompson, "Belle" in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 2nd Edition, 1983, 201-210.

[FMM89] R. Feldmann, B. Monien, P. Mysliwietz and O. Vornberger, Distributed Game Tree Search, Tech. Rep. 61, Dept. of Mathematics and Computer Science, Univ. of Paderborn, Paderborn, June 1989. A version also available as R. Feldmann, B. Monien, P. Mysliwietz and O. Vornberger, "Distributed Game-Tree Search," *Int. Computer Chess Assoc. J. 12*(2), 65-73, (1989).

[FMM91] R. Feldmann, P. Mysliwietz and B. Monien, "A Fully Distributed Chess Program" in D.F. Beal, ed., *Advances in Computer Chess 6,* Ellis Horwood Ltd., Chichester, 1991, 1-27.

[FeO88] E.W. Felten and S.W. Otto, "Chess on a Hypercube" in G. Fox, ed., *The Third Conf. on Hypercube Concurrent Computers and Applications, Vol II-Applications*, 1988, 1329-1341.

[FiF83] R. Finkel and J. Fishburn, "Improved Speedup Bounds for Parallel Alpha-Beta Search," *IEEE Trans. on Pattern Anal. and Mach. Intell. 5*(1), 89-92 (1983).

[FiF80] J. Fishburn and R. Finkel, "Parallel Alpha-Beta Search on Arachne," Tech. Rep. 394, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1981.

[Fis81] J. Fishburn, Analysis of Speedup in Distributed Algorithms, Ph.D. thesis, Computer Sciences Dept., Univ. of Wisconsin, Madison, 1981.

[Gil78] J.J. Gillogly, Performance Analysis of the Technology Chess Program, Tech. Rept. 189, Computer Science, Carnegie-Mellon Univ., Pittsburgh, March 1978.

[Hsu90] F-h. Hsu, Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess, Ph.D. thesis, Tech. Rep. CMU-CS-90-108, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, 1990.

[HAC90] F-h. Hsu, T.S. Anantharaman, M.S. Campbell and A. Nowatzyk, "Deep Thought" in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 55-78.

[HSN89] R.M. Hyatt, B.W. Suter and H.L. Nelson, "A Parallel Alpha/Beta Tree Searching Algorithm," *Parallel Computing 10*(3), 299-308 (1989).

[HGN90] R.M. Hyatt, A.E. Gower and H.L. Nelson, "Cray Blitz" in T.A. Marsland and J. Schaeffer, eds., *Computers, Chess, and Cognition*, Springer-Verlag, New York, 1990, 111-130.

[KnM75] D.E. Knuth and R.W. Moore, "An Analysis of Alpha-beta Pruning," *Artificial Intelligence 6*(4), 293-326 (1975).

[KoB82] D. Kopec and I. Bratko, "The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess" in M. Clarke, ed., *Advances in Computer Chess 3*, Pergamon Press, Oxford, 1982, 57-72.

[LSL93] R. Lake, J. Schaeffer and P. Lu, "Using Retrograde Analysis to Solve Large Combinatorial Search Spaces" in *Advances in Computer Chess 7*, 1993, in press.

[MaC82] T.A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees," *Computing Surveys 14*(4), 433-551 (1982).

[Mar83] T.A. Marsland, "Relative Efficiency of Alpha-Beta Implementations," *Procs. 8th Int. Joint Conf. on Art. Intell.*, (Los Altos: Kaufmann), Karlsruhe, Germany, Aug. 1983, 763-766.

[MaP85] T.A. Marsland and F. Popowich, "Parallel Game-Tree Search," *IEEE Trans. on Pattern Anal. and Mach. Intell. 7*(4), 442-452 (1985).

[MOS85] T.A. Marsland, M. Olafsson and J. Schaeffer, "Multiprocessor Tree-Search Experiments" in D. Beal, ed., *Advances in Computer Chess 4*, Pergamon Press, Elmsford, 1985, 37-51.

[MaS90] T.A. Marsland and J. Schaeffer (eds.), Computers, Chess, and Cognition, Springer-Verlag, New York, 1990.

[Mar92] T.A. Marsland, "Computer Chess and Search" in S. Shapiro, ed., *Encyclopedia of Artificial Intelligence*, John Wiley, 2nd Edition, 1992, 224-241. Also available as Tech. Rep. TR 91-10, Dept. of Computing Science, Univ. of Alberta, Edmonton, 1991.

[New95] M.M. Newborn, "A Parallel Search Chess Program," *Procs. ACM Ann. Conf.*, (New York: ACM), Denver, Oct 1985, 272-277.

[Pea80] J. Pearl, "Asymptotic Properties of Minimax Trees and Game Searching Procedures," *Artificial Intelligence 14*(2), 113-138 (1980).

[Rei83] A. Reinefeld, "An Improvement of the Scout Tree Search Algorithm," *J. of the Intl. Computer Chess Assoc. 6*(4), 4-14, (1983).

[Sam59] A.L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J. of Research and Development 3*(3), 210-229 (1959).

[Sam67] A.L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers II—Recent Progress," *IBM J. of Research and Development 11*(6), 601-617, (1967).

[Sch86] J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Univ. of Waterloo, Waterloo, Canada, 1986. Also available as Tech. Rep. TR86-12, Dept. of Computing Science, Univ. of Alberta, Edmonton, July 1986.

[Sch87] J. Schaeffer, Experiments in Distributed Game-Tree Searching, Tech. Rep. TR87-2, Dept. of Computing Science, Univ. of Alberta, Edmonton, January 1987.

[Sch89a] J. Schaeffer, "Distributed Game-Tree Search," *J. of Parallel and Distributed Computing 6*(2), 90-114 (1989).

[Sch89b] J. Schaeffer, "The History Heuristic and Alpha-Beta Search Enhancements in Practice," *IEEE Trans. on Pattern Anal. and Mach. Intell. 11*(11), 1203-1212 (1989).

[Sch92] J. Schaeffer, Man Versus Machine: The Silicon Graphics World Checkers Championship, Tech. Rep. TR92-19, Dept. of Computing Science, Univ. of Alberta, Edmonton, December 1992.

[SCT92] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu and D. Szafron, "A World Championship Caliber Checkers Program," *Artificial Intelligence 53*(2-3), 273-289 (1992).

[STL93] J. Schaeffer, N. Treloar, P. Lu and R. Lake, "Man Versus Machine for the World Checkers Championship," *AI Magazine 14*(2), 28-35 (1993).

[SlD69] J.R. Slagle and J.K. Dixon, "Experiments With Some Programs That Search Game Trees," *J. of the ACM 16*(2), 189-207 (1969).

[SlA77] D.J. Slate and L.R. Atkin, "CHESS 4.5 — The Northwestern Univ. Chess Program" in P. Frey, ed., *Chess Skill in Man and Machine*, Springer-Verlag, 1977, 82-118.

[SuG91] X-H. Sun and J.L. Gustafson, "Toward a Better Parallel Performance Metric," *Parallel Computing 17*(10-11), 1093-1109 (1991).

[Tho82] K. Thompson, "Computer Chess Strength" in M. Clarke, ed., *Advances in Computer Chess 3*, Pergamon Press, Oxford, 192, 55-56.

[Tho86] K. Thompson, "Retrograde Analysis of Certain Endgames," *Int. Computer Chess Assoc. J. 9*(3), 131-139 (1986).

[Tru78] T. Truscott, The *Duke* Checkers Program, Tech. Rep., Duke Univ., 1978.

Appendix A

Rules of Checkers In Brief[8]

Standard checkers notation assigns each square with a number, as in Figure A.1. Moves are given as from-square/to-square pairs. Black initially occupies squares 1 to 12 inclusive and White occupies squares 21 to 32 inclusive. If an opponent's checker is on an adjacent square, and there is an empty square behind it on the same diagonal, it can be captured (i.e. jumped) and removed from the board. In checkers, captures must be taken. Multiple captures are allowed and if more than one capture is possible, the player can choose. Checkers can only move and capture forwards. When a checker reaches the opponent's back rank (i.e. a White checker reaches one of 1 to 4, a Black checker reaches one of 29 to 32), it is promoted to a king. Kings can move and capture both forwards and backwards. A player wins if the opponent has no more pieces on the board, or if the opponent has no more legal moves. Black moves first.

	1		2		3		4
5		6		7		8	
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
29		30		31		32	

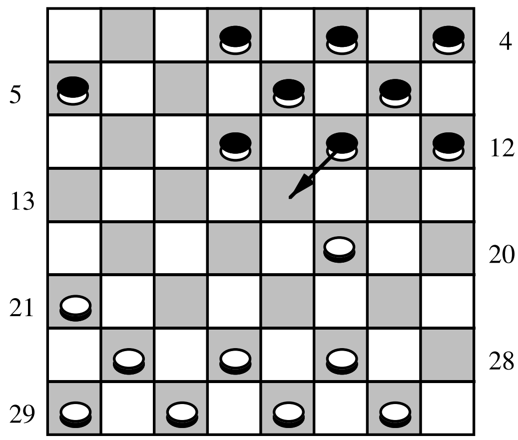
Figure A.1 — Checkers Board Notation

8 Adapted from [STL93]

Appendix B

Tinsley-Chinook 1992 Test Suite[9]

Game 1

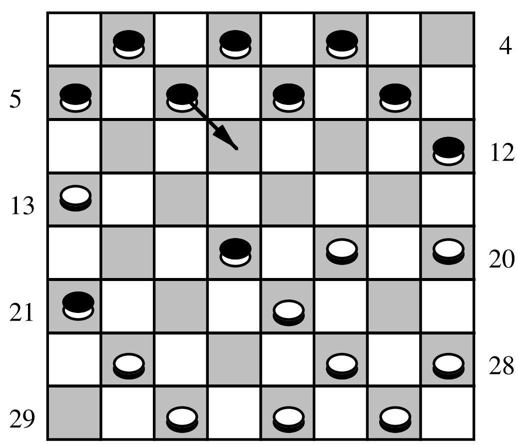


Black: *Chinook*
White: Tinsley

Black to move
Move: **11-15**

Piece count: 18
Legal moves at root: 7

Game 5



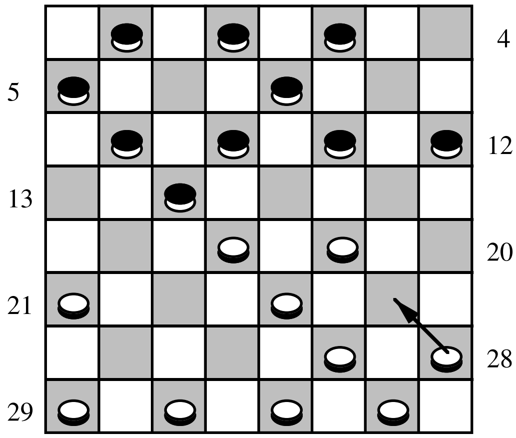
Black: Tinsley
White: *Chinook*

Black to move
Move: **6-10**

Piece count: 20
Legal moves at root: 8

9 See [Sch92] and [STL93] for further details on the match.

Game 7

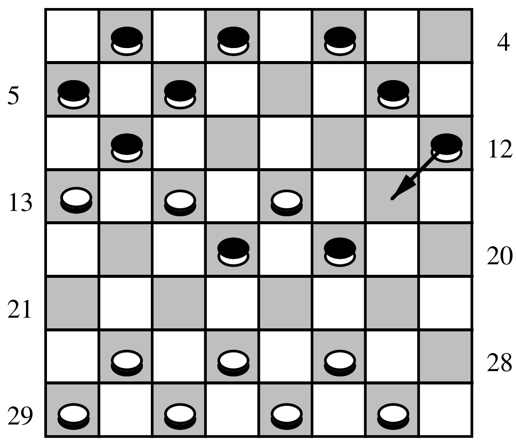


Black: Tinsley
White: *Chinook*

White to move
Move: **28-24**

Piece count: 20
Legal moves at root: 10

Game 8

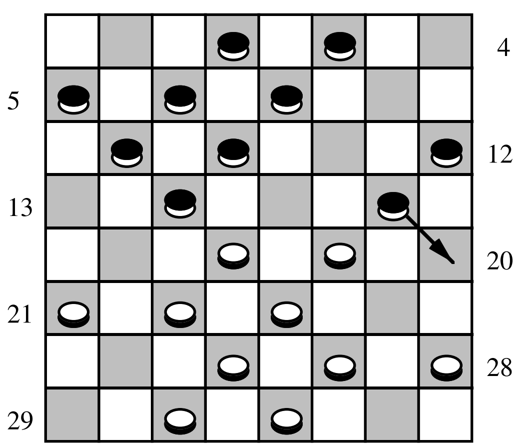


Black: *Chinook*
White: Tinsley

Black to move
Move: **12-16**

Piece count: 20
Legal moves at root: 9

Game 9

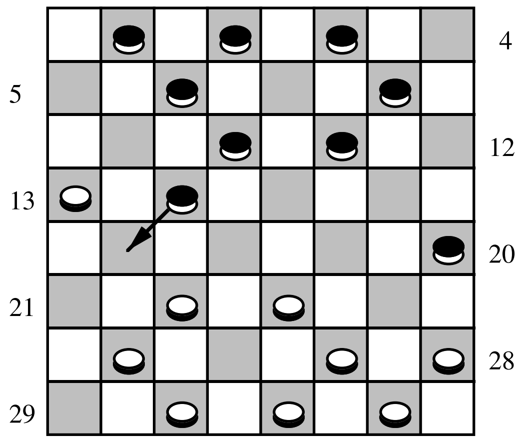


Black: *Chinook*
White: Tinsley

Black to move
Move: **16-20**

Piece count: 20
Legal moves at root: 6

Game 13

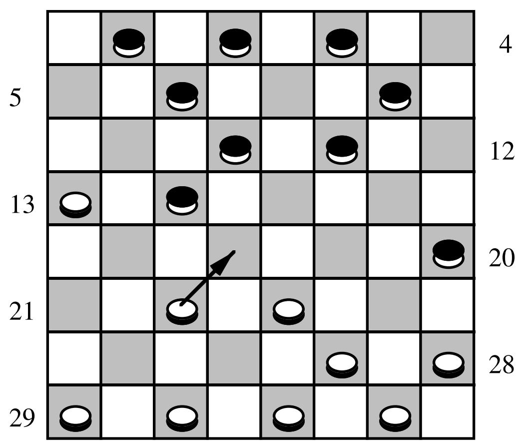


Black: *Chinook*
White: Tinsley

Black to move
Move: **14-17**

Piece count: 18
Legal moves at root: 11

Game 14

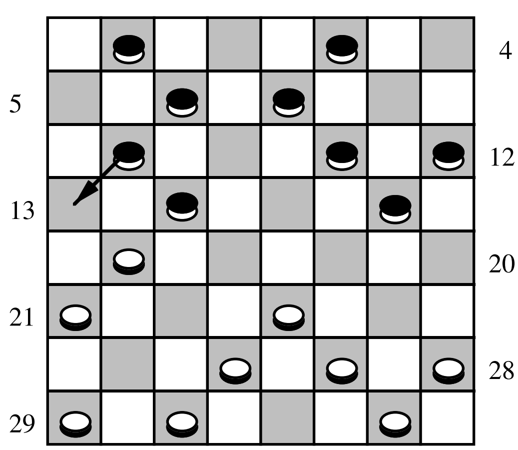


Black: Tinsley
White: *Chinook*

White to move
Move: **22-18**

Piece count: 18
Legal moves at root: 11

Game 15

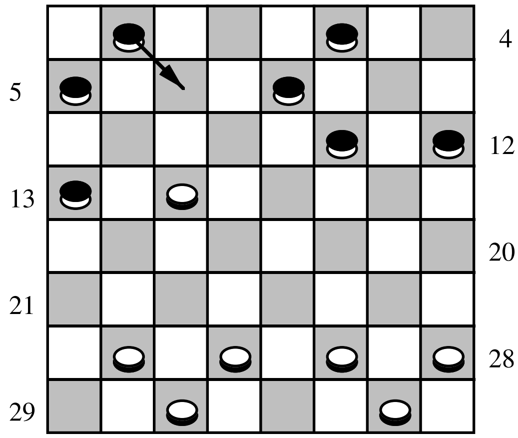


Black: Tinsley
White: *Chinook*

Black to move
Move: **9-13**

Piece count: 18
Legal moves at root: 9

Game 17

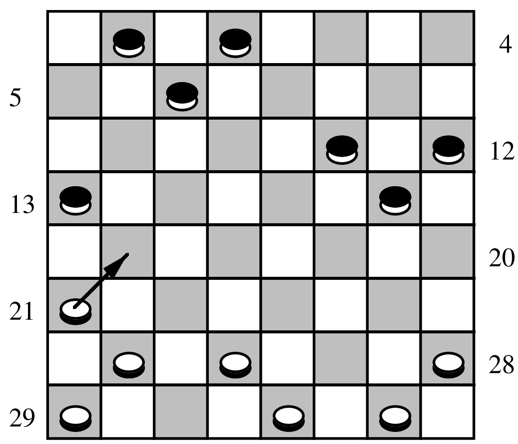


Black: *Chinook*
White: Tinsley

Black to move
Move: **1-6**

Piece count: 14
Legal moves at root: 8

Game 19

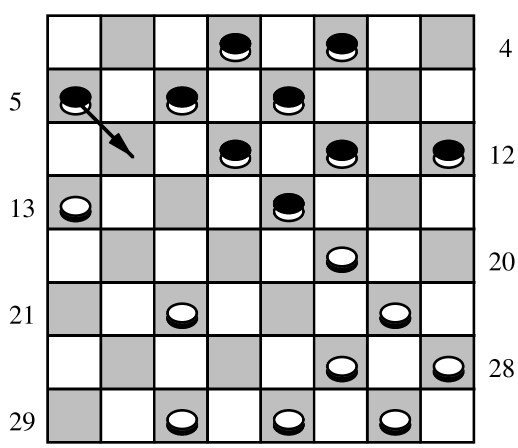


Black: Tinsley
White: *Chinook*

White to move
Move: **21-17**

Piece count: 14
Legal moves at root: 7

Game 22

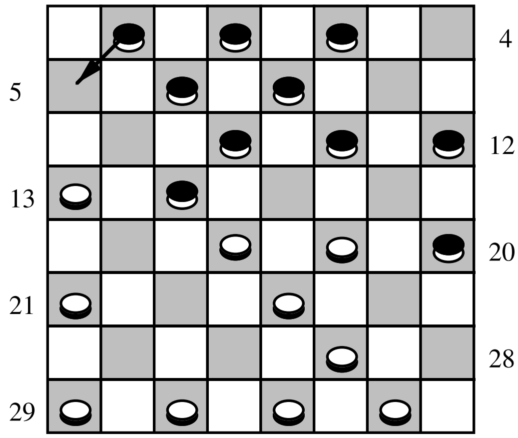


Black: Tinsley
White: *Chinook*

Black to move
Move: **5-9**

Piece count: 18
Legal moves at root: 7

Game 23

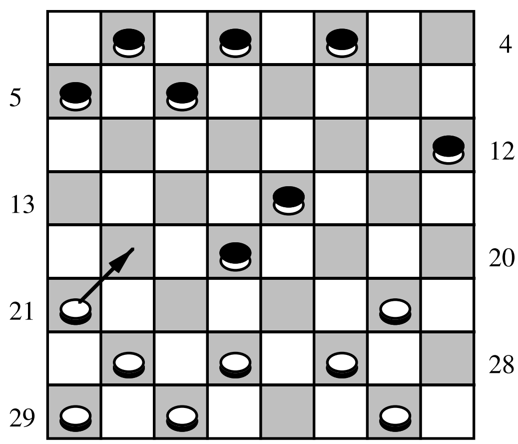


Black: Tinsley
White: *Chinook*

Black to move
Move: **1-5**

Piece count: 20
Legal moves at root: 9

Game 27

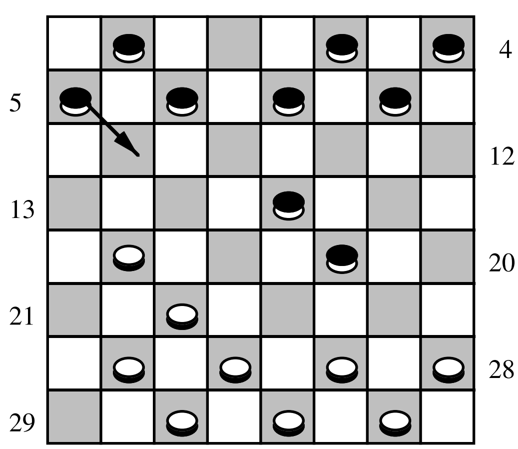


Black: Tinsley
White: *Chinook*

White to move
Move: **21-17**

Piece count: 16
Legal moves at root: 8

Game 29

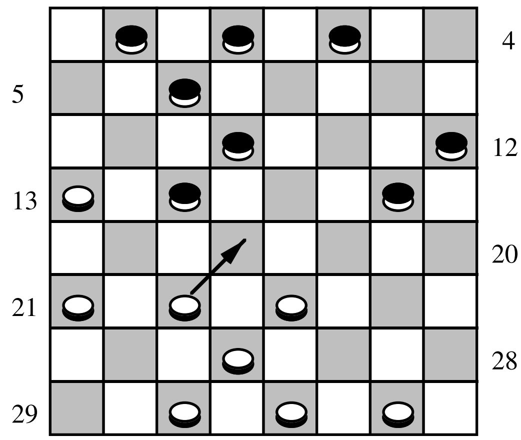


Black: *Chinook*
White: Tinsley

Black to move
Move: **5-9**

Piece count: 18
Legal moves at root: 10

Game 30

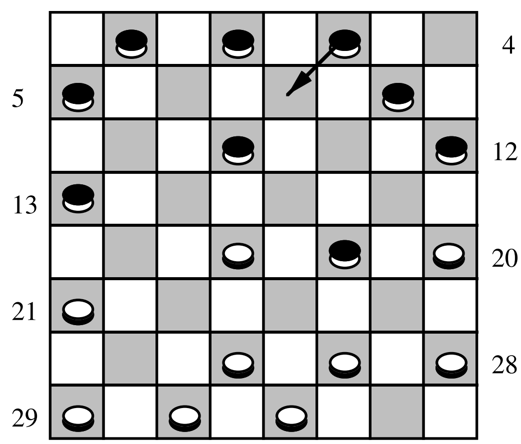


Black: Tinsley
White: *Chinook*

White to move
Move: **22-18**

Piece count: 16
Legal moves at root: 10

Game 31

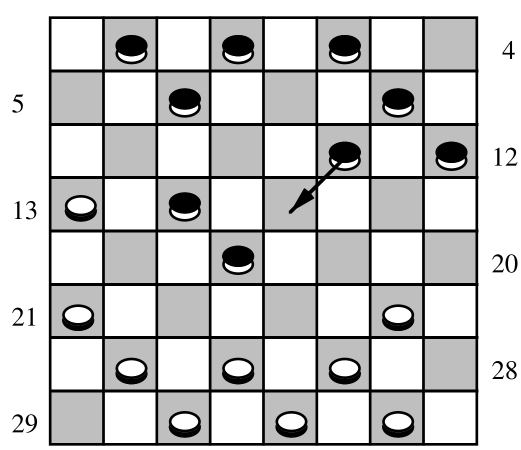


Black: Tinsley
White: *Chinook*

Black to move
Move: **3-7**

Piece count: 18
Legal moves at root: 12

Game 33

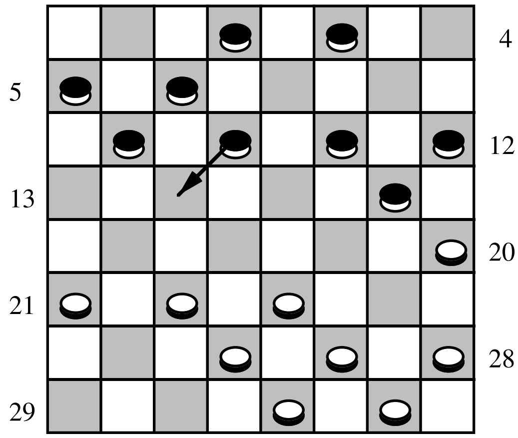


Black: *Chinook*
White: Tinsley

Black to move
Move: **11-15**

Piece count: 18
Legal moves at root: 11

Game 36

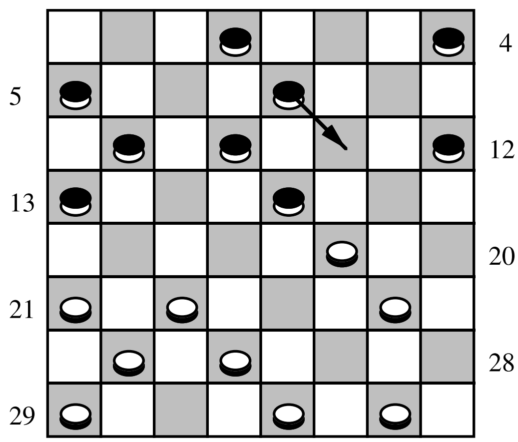


Black: *Chinook*
White: Tinsley

Black to move
Move: **10-14**

Piece count: 18
Legal moves at root: 9

Game 37

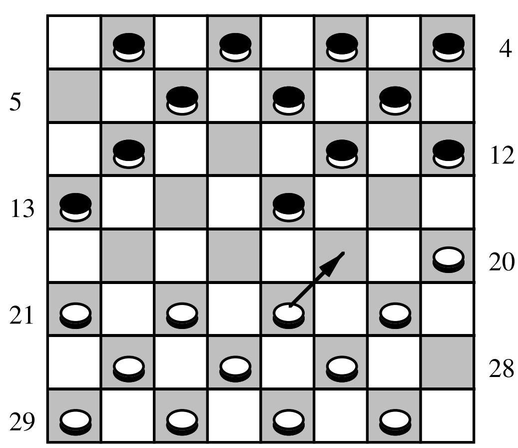


Black: *Chinook*
White: Tinsley

Black to move
Move: **7-11**

Piece count: 18
Legal moves at root: 8

Game 38



Black: Tinsley
White: *Chinook*

White to move
Move: **23-19**

Piece count: 24
Legal moves at root: 8

Appendix C

Experimental Data In Detail

Sequential *Chinook*

Game	Move	Depth	Score	Nodes	Time	N/S	Black	Turn	Pieces	Root BF
1	11–15	17	−14	4,147,002	594	6,981	Chinook	B	18	7
5	6–10	17	−16	1,721,679	231	7,453	Tinsley	B	20	8
7	28–24	15	1	1,697,552	249	6,817	Tinsley	W	20	10
8	12–16	17	13	2,990,826	417	7,172	Chinook	B	20	9
9	16–20	19	6	1,358,169	188	7,224	Chinook	B	20	6
13	14–17	15	−4	3,056,458	431	7,092	Chinook	B	18	11
14	22–18	15	7	3,224,856	462	6,980	Tinsley	W	18	11
15	9–13	17	−9	2,768,981	379	7,306	Tinsley	B	18	9
17	1–6	17	−5	3,635,672	487	7,465	Chinook	B	14	8
19	21–17	15	10	1,771,596	250	7,086	Tinsley	W	14	7
22	5–9	19	13	1,776,809	246	7,223	Tinsley	B	18	7
23	1–5	17	3	2,148,151	300	7,161	Tinsley	B	20	9
27	21–17	15	8	1,440,079	204	7,059	Tinsley	W	16	8
29	5–9	17	2	3,567,708	508	7,023	Chinook	B	18	10
30	22–18	17	−9	3,276,323	447	7,330	Tinsley	W	16	10
31	3–7	17	−12	3,610,286	489	7,383	Tinsley	B	18	12
33	11–15	15	−2	1,807,880	253	7,146	Chinook	B	18	11
36	10–14	19	8	3,331,558	511	6,520	Chinook	B	18	9
37	7–11	17	−8	1,689,039	231	7,312	Chinook	B	18	8
38	23–19	17	4	5,010,806	717	6,989	Tinsley	W	24	8

Table C.1 — Tinsley-*Chinook* 1992 Test Suite, Sequential Search Data (time in seconds)

Referring to Table C.1, in the position from Game 5, the move played by Black (Dr. Tinsley) was 6-10. Sequential *Chinook* searched that position to a nominal depth of 17 plies and selected 6-10 as the best move with a score of -16 points, where a piece (i.e. checker) is worth 100 points and a king is worth an additional 30 points. Sequential *Chinook* searched 1,721,679 total nodes in 231 seconds (7,453 nodes/second). In that position, there were a total of 20 pieces remaining on the board, and there were 8 legal moves in the root position.

Principal Variation Splitting

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,128,252	388	1.53	−0.450	13.6	
5	−17*	1,657,770	156	1.48	−3.71	16.0	
7	1	1,932,830	183	1.36	13.9	15.5	
8	13	4,107,391	382	1.09	37.3	16.1	
9	6	1,046,004	100	1.88	−23.0	22.2	
13	−4	3,839,579	403	1.07	25.6	18.5	
14	7	3,310,092	331	1.40	2.64	21.7	
15	−9	3,042,091	327	1.16	9.86	29.9	
17	−5	4,184,049	369	1.32	15.1	16.6	
19	10	1,872,404	187	1.34	5.69	18.2	
22	8*	2,118,772	208	1.18	19.3	24.5	
23	4*	2,636,876	227	1.32	22.8	10.5	
27	8	1,629,378	146	1.40	13.2	11.2	
29	2	3,772,099	345	1.47	5.73	15.6	
30	−11*	5,369,202	479	0.933	63.9	14.0	Plays 31–27
31	−12	3,518,025	285	1.72	−2.56	8.92	
33	−6*	1,964,771	213	1.19	8.68	18.0	
36	3*	4,129,920	449	1.14	24.0	23.3	
37	−10*	1,498,570	138	1.67	−11.3	18.9	
38	7*	3,795,110	387	1.85	−24.3	24.1	
		Average:		1.38	10.1	17.9	
		Std. Deviation:		0.260	20.1	5.25	

Table C.2 — Principal Variation Splitting (PVSplit), 2 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,863,334	383	1.55	17.3	47.2	
5	−13*	1,389,027	84	2.75	−19.3	39.5	
7	1	2,022,535	143	1.74	19.1	42.9	
8	13	3,724,304	267	1.56	24.5	44.8	
9	6	1,037,548	68	2.76	−23.6	42.9	
13	−4	3,635,724	288	1.50	19.0	42.1	
14	7	3,171,213	277	1.67	−1.66	54.9	
15	−9	3,006,452	267	1.42	8.58	57.3	
17	−5	3,883,726	263	1.85	6.82	44.5	
19	10	2,008,442	175	1.43	13.4	49.9	
22	21*	4,076,931	385	0.639	129	56.0	
23	4*	3,071,762	200	1.50	43.0	40.7	
27	8	1,758,767	114	1.79	22.1	37.4	
29	2	3,717,482	271	1.87	4.20	46.7	
30	−11*	5,795,981	381	1.17	76.9	40.0	Plays 31–27
31	−12	3,522,265	213	2.30	−2.44	38.5	
33	−2	2,322,067	190	1.33	28.4	43.7	
36	8	3,823,024	341	1.50	14.8	54.1	
37	−10*	1,477,428	85	2.72	−12.5	36.4	
38	11*	3,295,441	254	2.82	−34.2	49.7	
		Average:		1.79	16.7	45.5	
		Std. Deviation:		0.591	36.0	6.36	

Table C.3 — Principal Variation Splitting (PVSplit), 4 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	5,069,863	369	1.61	22.3	70.8	
5	−13*	1,388,912	88	2.63	−19.3	71.1	
7	1	1,905,201	132	1.89	12.2	70.1	
8	13	4,182,553	284	1.47	39.9	70.4	
9	6	1,035,137	67	2.81	−23.8	71.0	
13	−4	3,974,516	285	1.51	30.0	64.7	
14	7	3,138,124	262	1.76	−2.69	76.0	
15	−6*	2,843,340	222	1.71	2.69	75.0	
17	−5	3,981,729	253	1.92	9.52	69.7	
19	10	2,081,931	165	1.52	17.5	70.8	
22	21*	4,277,504	406	0.606	141	77.4	
23	4*	2,941,137	173	1.73	36.9	67.0	
27	8	1,618,998	96	2.13	12.4	64.3	
29	2	3,741,091	263	1.93	4.86	71.9	
30	−9	4,262,943	279	1.60	30.1	66.4	
31	−12	3,562,958	198	2.47	−1.31	66.0	
33	−2	2,357,636	167	1.52	30.4	62.3	
36	8	3,735,377	329	1.55	12.1	76.3	
37	−9*	1,627,193	96	2.41	−3.66	67.9	
38	11*	3,373,991	245	2.93	−32.7	73.2	
		Average:		1.88	15.9	70.1	
		Std. Deviation:		0.548	35.4	4.20	

Table C.4 — Principal Variation Splitting (PVSplit), 8 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	5,107,005	367	1.62	23.2	85.1	
5	−13*	1,390,055	88	2.63	−19.3	85.0	
7	1	1,913,578	131	1.90	12.7	84.9	
8	13	4,181,648	283	1.47	39.8	85.1	
9	6	1,035,950	67	2.81	−23.7	85.3	
13	−4	3,279,630	214	2.01	7.30	83.3	
14	7	3,223,037	268	1.72	−0.0564	87.9	
15	−6*	2,873,485	224	1.69	3.77	87.7	
17	−5	4,322,582	250	1.95	18.9	83.5	
19	10	2,128,884	169	1.48	20.2	85.3	
22	21*	4,078,999	371	0.663	130	88.0	
23	4*	2,939,777	174	1.72	36.9	83.3	
27	8	1,641,243	98	2.08	14.0	81.3	
29	2	3,745,228	264	1.92	4.98	85.7	
30	−9	4,250,167	278	1.61	29.7	82.8	
31	−12	3,502,222	196	2.49	−2.99	83.3	
33	−2	2,617,044	187	1.35	44.8	80.4	
36	8	3,730,132	327	1.56	12.0	88.1	
37	−10*	1,458,777	81	2.85	−13.6	83.4	
38	11*	3,379,468	245	2.93	−32.6	86.7	
		Average:		1.92	15.3	84.8	
		Std. Deviation:		0.574	34.1	2.18	

Table C.5 — Principal Variation Splitting (PVSplit), 16 Searchers Results
(* = score is different from sequential search, time in seconds)

Principal Variation Frontier Splitting

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,759,074	423	1.40	14.8	7.85	
5	−16	1,494,056	120	1.93	−13.2	9.15	
7	1	1,889,738	170	1.46	11.3	11.44	
8	13	3,667,551	325	1.28	22.6	10.8	
9	6	946,151	82	2.29	−30.3	15.2	
13	−4	3,468,300	345	1.25	13.5	12.8	
14	7	3,291,823	316	1.46	2.08	19.2	
15	−9	3,916,876	381	0.995	41.5	20.7	
17	−5	3,937,807	311	1.57	8.31	7.91	
19	10	1,923,675	180	1.39	8.58	12.9	
22	13	2,013,199	174	1.41	13.3	14.1	
23	4*	2,657,557	216	1.39	23.7	5.23	
27	8	1,330,856	106	1.92	−7.58	4.74	
29	2	4,055,250	357	1.42	13.7	12.1	
30	−9	3,850,291	401	1.11	17.5	11.3	
31	−13*	4,361,494	365	1.34	20.8	12.3	
33	−6*	1,952,806	188	1.35	8.02	11.4	
36	8	4,426,706	483	1.06	32.9	23.6	
37	−9*	1,590,280	138	1.67	−5.85	15.1	
38	6*	3,995,994	373	1.92	−20.3	18.2	
		Average:		1.48	8.76	12.8	
		Std. Deviation:		0.325	17.4	4.90	

Table C.6 — Principal Variation Frontier Splitting (PVFSplit), 2 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	6,859,264	346	1.72	65.4	18.6	
5	−13*	1,365,501	71	3.25	−20.7	29.9	
7	1	2,143,513	122	2.04	26.3	30.2	
8	13	4,379,309	233	1.79	46.4	26.1	
9	6	1,095,126	65	2.89	−19.4	38.2	
13	−4	3,808,314	260	1.66	24.6	33.6	
14	7	3,850,317	255	1.81	19.4	40.6	
15	−9	3,754,925	256	1.48	35.6	43.0	
17	−7*	4,360,652	197	2.47	19.9	18.5	
19	10	2,476,755	142	1.76	39.8	28.8	
22	8*	2,654,269	181	1.36	49.4	36.7	
23	4*	3,676,823	175	1.71	71.2	15.7	
27	7*	2,018,256	127	1.61	40.2	32.6	
29	2	4,365,675	238	2.13	22.4	26.2	
30	−10*	4,269,145	259	1.73	30.3	30.9	Plays 31–27
31	−12	3,554,156	184	2.66	−1.55	26.3	
33	−3*	2,248,628	147	1.72	24.4	26.2	
36	6*	4,451,511	345	1.48	33.6	45.7	
37	−9*	1,309,180	81	2.85	−22.5	39.8	
38	4	5,061,730	288	2.49	1.02	29.1	
		Average:		2.03	24.3	30.8	
		Std. Deviation:		0.546	26.3	8.17	

Table C.7 — Principal Variation Frontier Splitting (PVFSplit), 4 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−13*	11,003,085	397	1.50	165	41.9	Plays 10–14
5	−13*	1,603,113	55	4.20	−6.89	46.2	
7	1	2,322,187	94	2.65	36.8	50.9	
8	4*	4,575,062	205	2.03	53.0	56.6	
9	6	1,246,268	50	3.76	−8.24	53.5	
13	−4	4,587,382	223	1.93	50.1	51.6	
14	7	4,771,712	173	2.67	48.0	46.9	
15	−5*	4,271,397	175	2.17	54.3	53.6	
17	−5	4,769,071	160	3.04	31.2	44.4	
19	10	3,250,465	113	2.21	83.5	40.6	
22	21*	3,109,622	130	1.89	75.0	54.4	
23	4*	5,069,033	143	2.10	136	31.1	
27	8	2,222,705	67	3.04	54.4	36.5	
29	2	5,600,403	199	2.55	57.0	45.9	
30	−10*	4,639,208	189	2.37	41.6	51.9	Plays 31–27
31	−12	3,909,546	147	3.33	8.29	51.6	
33	−3*	2,243,591	78	3.24	24.1	43.5	
36	5*	5,212,331	246	2.08	56.5	58.3	
37	−10*	1,558,696	81	2.85	−7.72	63.4	
38	11*	3,830,082	158	4.54	−23.6	53.4	
		Average:		2.71	46.4	48.8	
		Std. Deviation:		0.803	46.3	7.74	

Table C.8 — Principal Variation Frontier Splitting (PVFSplit), 8 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−13*	10,297,066	273	2.18	148	61.9	
5	−16	2,298,921	74	3.12	33.5	70.3	
7	1	2,432,525	92	2.71	43.3	74.1	
8	13	6,781,585	235	1.77	127	72.1	
9	6	1,667,907	41	4.59	22.8	62.7	
13	−4	3,597,422	122	3.53	17.7	71.0	
14	7	5,132,661	164	2.82	59.2	69.5	
15	−8*	6,087,879	259	1.46	120	78.1	Plays 11–15
17	−12*	6,824,302	173	2.82	87.7	62.5	
19	10	3,289,290	100	2.50	85.7	68.3	
22	18*	5,082,558	127	1.94	186	62.0	
23	4*	5,691,739	122	2.46	165	55.3	
27	7*	2,409,200	56	3.64	67.3	59.5	
29	3*	6,113,307	141	3.60	71.4	58.5	
30	−13*	6,471,126	202	2.21	97.5	69.7	
31	−12	5,136,121	128	3.82	42.3	62.5	
33	−3*	2,964,894	82	3.09	64.0	63.2	
36	3*	11,006,484	648	0.789	230	82.7	
37	−8	1,242,234	45	5.13	−26.5	74.0	
38	5*	6,434,754	174	4.12	28.4	64.6	
		Average:		2.91	83.5	67.1	
		Std. Deviation:		1.07	63.5	7.00	

Table C.9 — Principal Variation Frontier Splitting (PVFSplit), 16 Searchers Results
(* = score is different from sequential search, time in seconds)

Principal Variation Splitting with Load Balancing

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,153,624	382	1.56	0.160	11.3	
5	−17*	1,560,965	143	1.62	−9.33	14.3	
7	1	2,000,956	180	1.38	17.9	11.3	
8	13	4,167,069	378	1.10	39.3	13.3	
9	6	967,379	84	2.24	−28.8	15.3	
13	−4	3,946,809	412	1.05	29.1	15.7	
14	7	3,515,729	318	1.45	9.02	11.3	
15	−9	3,103,781	286	1.33	12.1	16.6	
17	−5	4,057,776	332	1.47	11.6	9.98	
19	7*	1,976,706	209	1.20	11.6	17.9	
22	8*	2,127,427	195	1.26	19.7	16.5	
23	4*	2,637,240	226	1.33	22.8	9.90	
27	8	1,637,130	146	1.40	13.7	11.5	
29	2	3,785,136	344	1.48	6.09	11.3	
30	−11*	5,716,446	523	0.855	74.5	12.9	Plays 31–27
31	−12	3,546,584	287	1.70	−1.76	8.36	
33	−6*	1,965,308	212	1.19	8.71	17.7	
36	3*	4,313,101	406	1.26	29.5	10.9	
37	−9*	1,291,749	105	2.20	−23.5	9.53	
38	6*	4,191,066	403	1.78	−16.4	14.0	
		Average:		1.44	11.3	13.0	
		Std. Deviation:		0.346	23.0	2.86	

Table C.10 — Principal Variation Splitting with Load Balancing (PVSplit-LB), 2 Searchers Results (* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,897,059	323	1.84	18.1	35.8	
5	−13*	1,410,455	82	2.82	−18.1	37.4	
7	1	2,117,522	145	1.72	24.7	39.4	
8	13	3,946,858	289	1.44	32.0	39.6	
9	6	1,068,764	63	2.98	−21.3	37.7	
13	−4	3,846,023	298	1.45	25.8	38.6	
14	7	3,539,065	242	1.91	9.74	39.3	
15	−6*	2,799,241	184	2.06	1.09	41.5	
17	−5	3,998,435	245	1.99	9.98	38.1	
19	10	2,008,948	160	1.56	13.4	44.6	
22	21*	3,883,701	276	0.891	119	39.8	
23	4*	2,764,950	159	1.89	28.7	32.4	
27	8	1,760,144	111	1.84	22.2	36.4	
29	2	3,875,373	256	1.98	8.62	36.0	
30	−13*	6,251,358	388	1.15	90.8	33.3	Plays 31–27
31	−12	3,733,258	203	2.41	3.41	30.5	
33	−2	2,408,405	196	1.29	33.2	40.6	
36	8	5,576,072	348	1.47	67.4	32.1	
37	−9*	1,838,554	110	2.10	8.85	33.3	
38	11*	3,435,448	245	2.93	−31.4	42.0	
		Average:		1.89	22.3	37.4	
		Std. Deviation:		0.567	35.7	3.69	

Table C.11 — Principal Variation Splitting with Load Balancing (PVSplit-LB), 4 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,784,723	299	1.99	15.4	64.0	
5	−13*	1,418,156	85	2.72	−17.6	66.2	
7	1	2,196,787	133	1.87	29.4	64.3	
8	13	4,529,564	262	1.59	51.5	59.6	
9	6	1,072,509	62	3.03	−21.0	69.0	
13	−4	3,545,173	222	1.94	16.0	63.4	
14	7	3,639,907	235	1.97	12.9	65.7	
15	−6*	2,881,579	167	2.27	4.07	66.5	
17	−5	4,573,658	238	2.05	25.8	59.1	
19	10	2,174,969	159	1.57	22.8	67.7	
22	21*	4,255,805	275	0.895	140	65.0	
23	4*	2,971,705	161	1.86	38.3	63.4	
27	8	1,629,990	95	2.15	13.2	63.3	
29	2	4,033,818	239	2.13	13.1	62.3	
30	−9	4,305,663	268	1.67	31.4	64.5	
31	−12	3,721,975	171	2.86	3.09	55.7	
33	−2	2,653,464	185	1.37	46.8	60.2	
36	8	4,086,296	252	2.03	22.7	66.0	
37	−9*	1,775,127	105	2.20	5.10	60.2	
38	11*	3,698,824	239	3.00	−26.2	64.9	
		Average:		2.06	21.3	63.6	
		Std. Deviation:		0.539	34.6	3.23	

Table C.12 — Principal Variation Splitting with Load Balancing (PVSplit-LB), 8 Searchers Results (* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	5,190,807	310	1.92	25.2	80.9	
5	−13*	1,441,710	87	2.66	−16.3	82.3	
7	1	2,167,872	136	1.83	27.7	82.3	
8	13	4,578,935	265	1.57	53.1	79.3	
9	6	1,079,224	63	2.98	−20.5	85.2	
13	−4	4,434,180	305	1.41	45.1	79.7	
14	7	3,532,129	218	2.12	9.53	82.0	
15	−9	3,397,632	188	2.02	22.7	81.3	
17	−5	4,149,836	236	2.06	14.1	81.9	
19	10	2,177,327	158	1.58	22.9	83.3	
22	21*	4,231,845	276	0.891	138	82.3	
23	4*	2,824,026	151	1.99	31.5	81.8	
27	8	1,660,368	96	2.13	15.3	81.3	
29	2	4,039,350	250	2.03	13.2	81.8	
30	−9	4,313,981	268	1.67	31.7	82.1	
31	−12	3,922,302	186	2.63	8.64	78.5	
33	−2	2,727,977	190	1.33	50.9	79.1	
36	8	40,95,680	251	2.04	22.9	82.8	
37	−9*	1,914,082	106	2.18	13.3	78.7	
38	11*	3,733,014	241	2.98	−25.5	81.8	
		Average:		2.00	24.2	81.4	
		Std. Deviation:		0.530	34.1	1.66	

Table C.13 — Principal Variation Splitting with Load Balancing (PVSplit-LB), 16 Searchers Results (* = score is different from sequential search, time in seconds)

Principal Variation Frontier Splitting with Load Balancing

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	4,734,395	418	1.42	14.2	7.27	
5	−16	1,493,974	119	1.94	−13.2	9.09	
7	1	1,909,130	176	1.41	12.5	11.9	
8	13	3,789,751	338	1.23	26.7	10.0	
9	6	954,879	81	2.32	−29.7	13.3	
13	−4	3,692,904	355	1.21	20.8	9.54	
14	7	3,365,953	296	1.56	4.38	9.56	
15	−9	3,962,077	332	1.14	43.1	8.07	
17	−5	3,974,399	305	1.60	9.32	4.95	
19	10	1,897,205	176	1.42	7.09	11.3	
22	13	1,950,335	162	1.52	9.77	8.24	
23	4*	2,668,012	215	1.40	24.2	4.63	
27	8	1,371,745	108	1.89	−4.75	4.15	
29	2	4,159,271	355	1.43	16.6	6.79	
30	−9	3,858,545	398	1.12	17.8	10.8	
31	−13*	4,352,534	341	1.43	20.6	6.47	
33	−6*	2,823,238	275	0.920	56.2	11.4	
36	8	4,883,847	433	1.18	46.6	5.74	
37	−9*	1,425,349	112	2.06	−15.6	7.18	
38	11*	3,466,980	297	2.41	−30.8	7.85	
		Average:		1.53	11.8	8.41	
		Std. Deviation:		0.402	23.0	2.57	

Table C.14 — Principal Variation Frontier Splitting with Load Balancing (PVFSplit-LB),
2 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	6,959,056	335	1.77	67.8	14.9	
5	−13*	1,366,231	71	3.25	−20.7	29.2	
7	1	2,147,259	118	2.11	26.5	25.8	
8	13	3,745,386	222	1.88	25.2	30.7	
9	6	1,133,069	62	3.03	−16.6	32.4	
13	−4	4,181,356	273	1.58	36.8	29.5	
14	7	3,943,371	209	2.21	22.3	23.7	
15	−9	4,010,342	204	1.86	44.8	22.5	
17	−7*	4,453,223	197	2.47	22.5	16.9	
19	10	2,615,897	142	1.76	47.7	23.2	
22	13	2,530,979	146	1.68	42.5	31.5	
23	4*	3,694,767	167	1.80	72.0	15.0	
27	8	2,111,255	113	1.81	46.6	21.8	
29	2	4,460,449	220	2.31	25.0	17.3	
30	−10*	4,211,776	240	1.86	28.6	28.5	Plays 31–27
31	−12	3,684,349	173	2.83	2.05	19.3	
33	−3*	2,259,988	140	1.81	25.0	25.7	
36	6*	4,538,537	240	2.13	36.2	22.2	
37	−9*	1,620,708	89	2.60	−4.05	28.1	
38	4	5,466,451	260	2.76	9.09	18.0	
		Average:		2.18	27.0	23.8	
		Std. Deviation:		0.491	24.5	5.61	

Table C.15 — Principal Variation Frontier Splitting with Load Balancing (PVFSplit-LB),
4 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	8,584,005	249	2.39	107	28.5	
5	−13*	1,449,089	52	4.44	−15.8	50.1	
7	1	2,347,731	90	2.77	38.3	45.9	
8	14*	6,264,522	224	1.86	109	38.1	
9	6	1,262,043	48	3.92	−7.08	51.6	
13	−4	6,406,134	250	1.72	110	42.0	
14	7	5,020,221	162	2.85	55.7	38.9	
15	−5*	4,146,170	134	2.83	49.7	41.1	
17	−5	5,911,433	188	2.59	62.6	36.3	
19	10	3,046,270	110	2.27	72.0	43.0	
22	13	2,831,192	112	2.20	59.3	51.9	
23	4*	4,985,740	133	2.26	132	28.2	
27	7*	2,315,199	69	2.96	60.8	36.1	
29	2	6,489,585	207	2.45	81.9	36.7	
30	−10*	4,500,069	169	2.65	37.4	47.7	Plays 31–27
31	−12	4,548,592	147	3.33	26.0	37.6	
33	−3*	2,289,609	76	3.33	26.7	41.0	
36	8	8,177,389	270	1.89	145	37.6	
37	−9*	2,560,705	84	2.75	51.6	39.2	
38	11*	4,637,311	160	4.48	−7.45	44.6	
		Average:		2.80	59.8	40.8	
		Std. Deviation:		0.779	45.0	6.58	

Table C.16 — Principal Variation Frontier Splitting with Load Balancing (PVFSplit-LB), 8 Searchers Results
(* = score is different from sequential search, time in seconds)

Game	Score	Nodes	Time	Speedup	% Search Overhead	% Idle Time	Notes
1	−14	9,080,241	196	3.03	119	53.4	
5	−16	2,380,504	77	3.00	38.3	69.1	
7	1	2,888,112	98	2.54	70.1	70.7	
8	14*	5,816,061	152	2.74	94.5	62.0	
9	6	1,384,075	36	5.22	1.91	64.3	
13	−4	3,852,231	121	3.56	26.0	68.3	
14	7	5,821,955	144	3.21	80.5	59.2	
15	−9	5,794,131	152	2.49	109	63.1	
17	−5	6,103,567	138	3.53	67.9	59.9	
19	10	3,070,579	95	2.63	73.3	67.7	
22	21*	4,483,369	106	2.32	152	59.5	
23	4*	4,540,195	90	3.33	111	51.6	
27	8	2,170,753	53	3.85	50.7	61.7	
29	2	6,633,872	154	3.30	85.9	57.5	
30	−10*	4,846,438	153	2.92	47.9	69.9	Plays 31−27
31	−12	6,005,205	135	3.62	66.3	58.4	
33	−3*	3,346,207	119	2.13	85.1	65.6	
36	3*	7,377,430	175	2.92	121	58.8	
37	−8	1,995,842	64	3.61	18.2	70.7	
38	12*	4,900,511	110	6.52	−2.20	57.9	
		Average:		3.32	70.9	62.5	
		Std. Deviation:		1.01	41.4	5.67	

Table C.17 — Principal Variation Frontier Splitting with Load Balancing (PVFSplit-LB),
16 Searchers Results
(* = score is different from sequential search, time in seconds)