1988

A high performance parallel algorithm to search depth-first game trees.

Robert Morgan Hyatt
University of Alabama at Birmingham

Order Number 8909785

A high-performance parallel algorithm to search depth-first game trees

Hyatt, Robert Morgan, Ph.D.

The University of Alabama in Birmingham, 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A HIGH-PERFORMANCE PARALLEL ALGORITHM TO SEARCH DEPTH-FIRST GAME TREES

by

ROBERT MORGAN HYATT

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer and Information Sciences in the Graduate School, The University of Alabama at Birmingham

BIRMINGHAM, ALABAMA

1988

ABSTRACT OF DISSERTATION
GRADUATE SCHOOL, UNIVERSITY OF ALABAMA AT BIRMINGHAM

Degree Ph.D. Major Subject Computer & Information Sciences

Name of Candidate Robert Morgan Hyatt

Title A High-Performance Parallel Algorithm to Search Depth-First Game Trees

The alpha-beta tree search problem is described in mathematical terms to determine the feasibility of searching these trees in parallel. Three algorithms are developed to search depth-first game trees in parallel using a shared-memory multiprocessing computer system. Test results for uniform, non-uniform and alpha-beta depth-first game trees are provided.

The principal variation splitting algorithm (PVS) along with an enhanced version (EPVS) are described. After discussing the performance of these algorithms, a new parallel tree search algorithm, dynamic tree splitting (DTS) is developed and applied to the same trees. The DTS algorithm provides superior performance on the three types of trees discussed above, primarily because DTS does not require that all processors work together on descendants of the same node. This reduces synchronization overhead where a processor is out of work and has to wait on other processors to complete their work before moving to some other part of the tree. The details of this algorithm are given along with an explanation of the particular tree-searching problem each detail addresses.

Test results were obtained using a Sequent Balance 21000 multiprocessor with 30 processors. When searching alpha-beta game trees with sixteen processors, the PVS algorithm provides a speedup of 4.59, the EPVS algorithm provides a speedup of 5.98, and the DTS algorithm provides a speedup of 8.81.

The conclusions identify specific problem areas that must be addressed to improve these results. In particular, the sequential search must become more accurate in the nodes that it examines. Depth–first trees are inherently parallel, but the alpha–beta algorithm adds a sequential property that makes them extremely difficult to search in parallel without adding search overhead that degrades the total performance.

Abstract Approved by: Committee Chairman Bruce W. Suter

Program Director Wayne J. Gore

Date 12/29/88 Dean of Graduate School Anthony Ralston

PS-5744

ACKNOWLEDGEMENTS

The only unfortunate circumstance surrounding the awarding of the Ph.D. degree is that the degree is awarded to only one person, regardless of how many others had a hand in earning the degree. In my case, my family shares equally in earning this degree, without any one of us, things would have certainly fallen apart. My wife, Janie, has always been ready to give me another push at those times when my emotional and mental energies are at a low point. My daughter, Tanya, who has abilities that exceed my own, has taken the changes in our lifestyle in stride with no complaints and has actually enjoyed and grown from this experience. My son Chris, who "wants to graduate from High School, get his Ph.D. and work at UAB" has perhaps been most affected by the past three years. He has seen less of me than normal, but has grown up with the idea that education is everything. I could not have completed this degree without their support and I am extremely proud of the way all three of them have supported me.

The Department of Computer and Information Sciences at UAB has supported my interests in any way that I have asked. Dr. Bruce Suter, my advisor, has become interested in parallel trees (as well as other parallel algorithms) over the course of this research. The department chairman, Dr. Warren Jones, has made a major commitment to parallel processing that provided the machine used for this research. Drs. Kevin Reilly, Steve Harvey and Charles Katholi have provided hours of stimulating conversation and assistance in getting this research done. The department has supported me far beyond any reasonable expectations, and I look forward to continuing my academic career at UAB.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF TABLES (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

ENHANCED PRINCIPLE VARIATION SPLITTING (EPVS)

DYNAMIC TREE SPLITTING (DTS)

CONCLUSIONS

CHAPTER 1

GAME TREE SEARCHING

1.1. Introduction

Tree searching problems occur throughout the field of computer science. Artificial intelligence has game–playing, expert systems and natural language processing trees; compilers use parse trees and symbol tables; data bases use tree structures; and finally, trees play an important role in operations research for such things as the traveling salesman problem.

1.2. The Speed Problem

Since current programs running on the fastest available sequential computer systems cannot search trees fast enough to satisfy every application, developing high–performance tree search algorithms is quite important.

The most recent improvements in high–performance computation have been in parallel processing machines. These include shared memory architectures such as the Cray–XMP supercomputer and distributed architectures such as the hypercube and Sun Microsystems networks. While current examples of parallel supercomputing machines have a small number of processors (four on the Cray XMP), future machines will increase this by more than an order of magnitude.

Most current tree searching algorithms make little use of parallel processing, while those few that do use parallel machines work with small numbers of processors using distributed architecture machines [1, 2, 3, 5, 6, 8, 18, 25,27, 29].

Current research centers on multiprocessing hardware connected via some type of distributed network rather than the more costly (and less available) shared memory architectures. The communication delays caused by the network make search coordination and information interchange among the processors critical when considering the time that is lost waiting on inter-processor communication. Attempting to minimize these costs results in poor performance in terms of the speedup obtained by using parallel processing.

1.3. Parallel Search Requirements

A properly designed parallel search algorithm addresses two important performance-related issues. First, fully utilizing a parallel computer reduces the total time required to complete a search. Second, real-time constraints are sometimes imposed in systems that monitor processes, requiring accurate and frequent control signals. Often, completing a search before the system must make a decision is impossible. In that event, search speed is very important. However, being sure to search the important areas of the tree while leaving the less interesting areas for analysis if time permits is also necessary for making the proper decision.

The real-time requirement can be most readily seen in robotics and manufacturing. Work is now in progress to produce an autonomous vehicle controlled by a computer. The pattern-matching task which steers the vehicle must supply a control signal at the correct time regardless of whether the search is complete or not. Similarly, when controlling some machines, directions must be given at the appropriate time in order to avoid ruining the item being made or to avoid delaying other work requiring the item.

A parallel search algorithm must handle trees of varying complexity, such as those from chess and checker games. Trees produced near the end of the

games have low branching factors; likewise, trees produced when the position is tactical in nature have low branching factors because there are usually few responses to checks and captures. On the other hand, trees produced during the early and middle parts of these games can be extremely "bushy" with a large branching factor. The parallel algorithm must be able to handle this variance and produce adequate load balancing to achieve reasonable performance.

1.4. Purpose of This Research

The purpose of this research is to extend the state–of–the–art in parallel tree searching to take advantage of machines with more than four processors. Rather than accept current algorithms that cannot reach a tenfold performance increase with any number of processors, it seems intuitively possible to achieve a near–linear speed–up for machines with large numbers of processors [12, 21, 22].

1.5. Research Environment

UAB has provided a unique environment to pursue this type of research with the thirty processor Sequent Balance 21000 computer system that is readily available. Its shared memory architecture simplifies data sharing and effectively eliminates the cost of communication that is so significant on other types of machines and architectures. Additionally, the machine provides a sufficient level of parallelism so that testing algorithms targeted for the next generation of supercomputers provides accurate simulations of the parallel performance of the new machines.

CHAPTER 2

GAME TREE CHARACTERISTICS

2.1. Minimax Game Trees

Shannon proposed the minimax game tree search in 1949 as an algorithm to allow a computer to play chess using a technique called the zero sum game tree [29]. In this algorithm, positive scores represent good results for one side, negative scores represent good results for the opposing side, and a score of zero represents an equal state.

Minimax trees belong to the depth first class of tree searching algorithms where the scores are only computed for nodes at the end of branches (sometimes called terminal nodes). That is, scores are only computed when a branch reaches its maximum length.

The term minimax comes from one side favoring higher (MAX) numeric scores and the other side favoring lower (MIN) numeric scores. In this type of tree, MAX moves first and chooses the successor branch (move) that results in the highest numeric score. From each of these resulting positions, MIN moves and chooses the successor branch (move) that results in the lowest numeric score. MAX and MIN alternate turns in this manner until reaching some depth limit.

For a branch that cannot extend deeper, the static evaluation routine generates a numeric evaluation of the terminal node. MAX (or MIN depending on where the node occurs) chooses the branch with the highest (or lowest) numeric evaluation and returns this value to the previous level. There MIN (or

MAX) chooses the lowest (highest) of these backed up values and then returns it to the previous level. This algorithm examines all branches from the root position in this manner.

In these trees there is plenty of opportunity for parallel processing since the branches of the tree are completely independent of each other. There are synchronization points since MAX might give each of n processors a branch to examine and then choose the largest value returned by any of the processors. For common game trees, there are enough branches (millions, billions, or as many as needed!) to keep any number of processors busy. However, using a large number of processors turns out to be much more difficult than expected from an initial analysis.

2.2. Uniform Minimax Game Trees

Uniform game trees provide an excellent theoretical basis for developing a mathematical model for game trees. Chapter 4 uses this type of tree to develop a mathematical description of parallel game tree analysis.

A uniform game tree has a constant branching factor and maximum depth of search (figure 2.1). Each node has the same number of successors as any other node, except for nodes at the maximum allowable depth which have no successors. Also, all branches have the same length (from the root position).

This type of tree is the ideal model for parallel processing since each branch from a given node requires the same amount of work to search as any other branch from the same node. This "uniformity" simplifies the mathematical analysis because stochastic behavior is not present.

Unfortunately, uniform trees do not occur in actual tree searching problems. Even simple game tree searches must detect the occurrence of duplicate branches and take some action which destroys this uniformity. If this

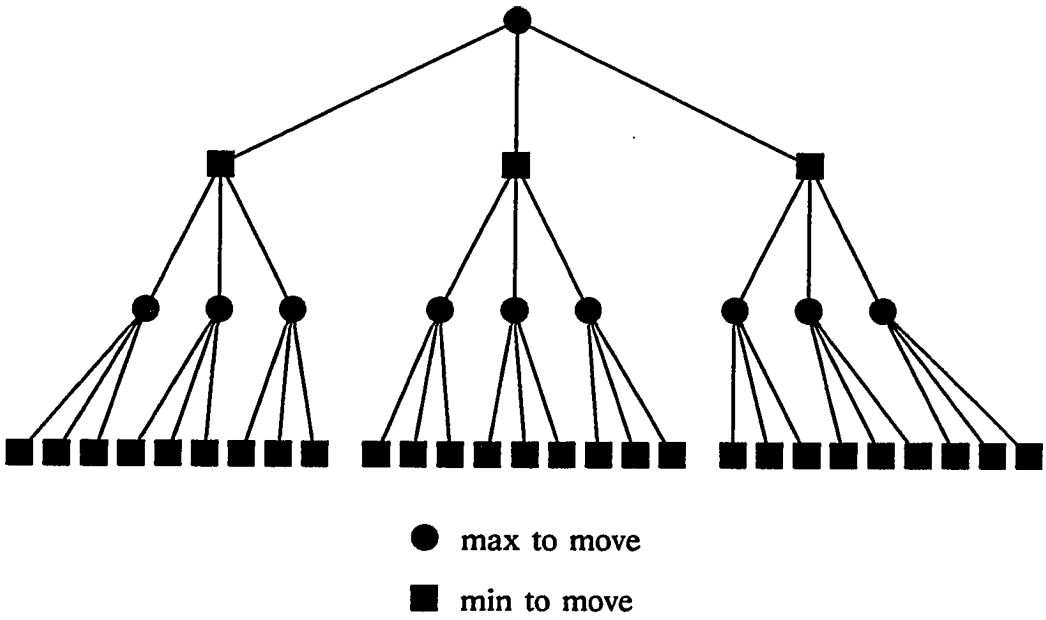

Figure 2.1: Uniform minimax game tree with constant width = 3 and constant depth = 3.

is not done, the game can simply "cycle" between two moves and make no progress toward the search goal. The mathematical analysis of uniform trees is interesting, but the unrealistic assumptions of such trees provide little practical benefit.

In uniform game trees, any two branches at the same depth generate subtrees with identical sizes. This is an important property for parallel processing since giving all processors identical amounts of work to do in parallel is a desirable goal which prevents one processor from waiting on another to complete a larger amount of work. A parallel search of such trees easily produces a near-optimal speedup where n processors complete the entire search in little more than 1/n time units. Knowing the amount of work required to analyze a subtree greatly simplifies the scheduling problems and makes it quite easy to keep all processors busy.

The primary consideration in searching uniform trees in parallel is to choose the depth for parallel division so that the total number of "chunks" of parallel work is evenly divisible by the number of processors available. This ensures that each of the n processors will have the same quantity of work to do, minimizing the amount of time where one processor is waiting on another to finish. A secondary consideration is to avoid giving processors "chunks" of work so small that the multiprocessing overhead of starting, coordinating, and synchronizing the tasks becomes a major portion of the total work done. This results in poor parallel performance since the sequential algorithm does not do this superfluous work.

Developing such an algorithm is non-trivial unless the branching factor is an exact multiple of the number of processors. Since this is unlikely, it is necessary that processors work at different nodes within the tree rather than simply dividing the work at one node among the available processors. This requirement significantly adds to the complexity of the code, but satisfying it produces good results on different computer systems with varying numbers of processors.

This suggested search algorithm reduces the success of the common "doall" or "doacross" type of parallel programming since there is no single loop that can have iterations executed in parallel with synchronization at the end of the loop. Conceptually, the algorithm appears as a sequential outer loop with an inner loop having its iterations executed in parallel; however, to keep all processors busy the next outer loop iteration begins as soon as one processor runs out of work inside the inner loop. Without such a modification to the search algorithm, parallel performance will never approach the maximum speed-up possible.

Neither an algorithm or code is given for this approach since the classic recursive procedure call method does not apply to this type of programming. While recursive procedure calls allow elegant solutions to tree search problems, they cause problems in this algorithm. The difficulty comes from processing a node of the tree by one instance of a recursive call. Until the return from the procedure is made, no further work can be done. An actual code would not use this approach and would therefore not suffer from this problem.

2.3. Non-Uniform Minimax Game Trees

Non-uniform game trees contain nodes with varying numbers of successors or branches that are not constant in length. Common game trees have both of these characteristics, generating trees that are very complex. Figure 2.2 depicts such a tree.

This type of tree is a more natural representation of games since it is unusual for both sides to have the same number of moves on their respective turns at play. For example, in checkers a player must make a jumping move if one is possible, giving only a few possible choices. If no jumps are possible, then there are many possible playing choices. This results in some branches that are quite simple (where jumps are possible at various nodes) while other branches are more complex (where there are no jumps possible) since there are more alternatives. Chess is similar in that when the king is in check, there are few legal alternatives to get out of check, while in other cases there can be over a hundred legal moves.

This type of tree causes significant problems for parallel implementations since obtaining uniform load balancing is not easy. If processor one is given a branch to examine where only one jump is possible, and processor two is given a branch where no jumps are possible, processor two is going to have a more

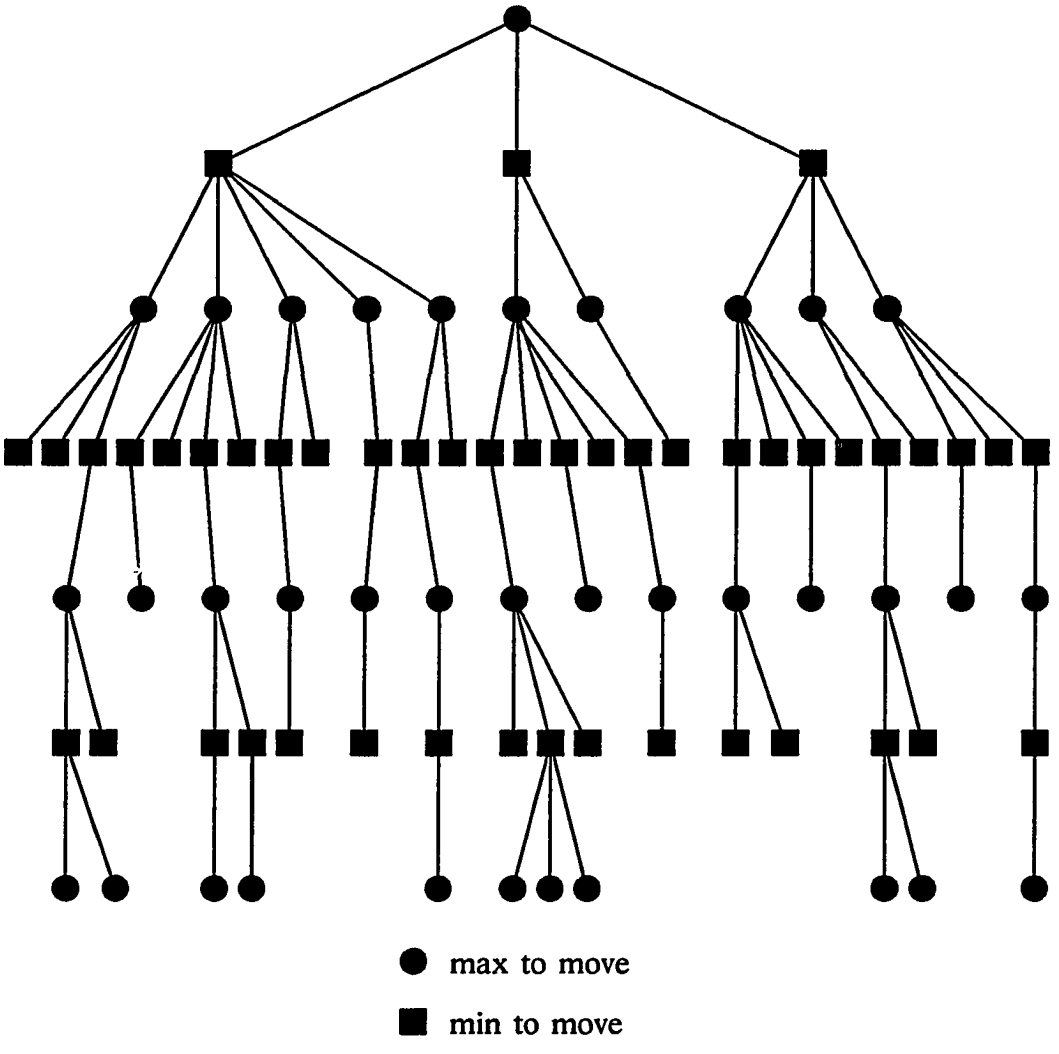

Figure 2.2: Non–Uniform minimax game tree.

complex task to complete. It is quite easy to have one processor waiting for a significant length of time while another processor searches a more difficult branch.

While non–uniformity in width is one type of problem, variable–depth searching is also necessary when trying to play certain types of games. In chess, humans analyze complicated or unclear positions more deeply than simple positions. Good programs must emulate this behavior and continue searching until these positions are properly evaluated.

By varying the depth and width of the typical game tree, the complexity of a given branch becomes impossible to determine without first searching it. This causes significant problems for a parallel search since one branch might have a subtree with few nodes while another branch generates a tremendous subtree with no warning. Measurements of actual chess-playing programs have found subtrees from the same parent that vary in size by several orders of magnitude. Giving such a complex branch to one processor while giving a more normal branch to another guarantees poor parallel performance. Even more important, a processor cannot predict such behavior before analyzing the position. It only discovers the complexity by searching the resulting [large] subtree.

This type of tree presents many different problems for a parallel search algorithm design since achieving load balancing is difficult. The traditional approach of splitting the tree at some node and giving each processor one or more branches at that point fails miserably since invariably one processor receives a branch that takes significantly more time to analyze than anything given to the others. Searching different size trees in parallel reduces the total search performance dramatically by increasing the inter-processor synchronization overhead.

2.4. Alpha-Beta Minimax Game Trees

The alpha-beta algorithm was developed specifically to reduce the size of the normal minimax game tree [13]. It is an intuitive method that refutes a branch without having to search the entire subtree below that branch.

Assume that MAX is to move and that after examining the first branch in the tree the minimax algorithm backs up a score of 1000 to this node. MAX remembers this score and then tries move number two. MIN now has to determine the best response to this move and after searching the first branch,

the minimax algorithm returns a best score of -500 to MIN. At this point, MIN could continue to examine the rest of its possible moves, but it turns out that it is not necessary. Since MIN is going to return the LOWEST value it can generate, the maximum of these values is -500 since MIN would never choose any branch resulting in a score larger than -500. However, MIN also knows that MAX already has a best score of 1000 and will not accept any score lower than that lower bound. MIN therefore determines that additional analysis is not necessary since the second branch tried by MAX is inferior to the first one (comparing -500 to 1000).

The important point here is that a move is worse than another move; exactly how much worse is not important since the move would never be chosen anyway. This algorithm is known as the alpha-beta algorithm where alpha and beta are the lower and upper search bounds. Any score lower than alpha can be immediately discarded as can any score greater than beta. Note that the first move sets the lower bound on the scores that MAX will accept. Whenever MAX finds a better move, MAX replaces the old lower bound with the new score since it is a better choice.

In the preceding example, consider what happens when examining the second move first. MAX obtains a score of -500 (or even a lower value since we do not know exactly how bad the second move is), and then discovers the next move results in a better score (1000). The search must examine both subtrees in their entirety resulting in a tree with more total nodes and more work. Move ordering is therefore very important since examining the best branch first generates the smallest possible tree (a mathematical discussion of this is given in chapter 4). This move order sensitivity is an important property of the

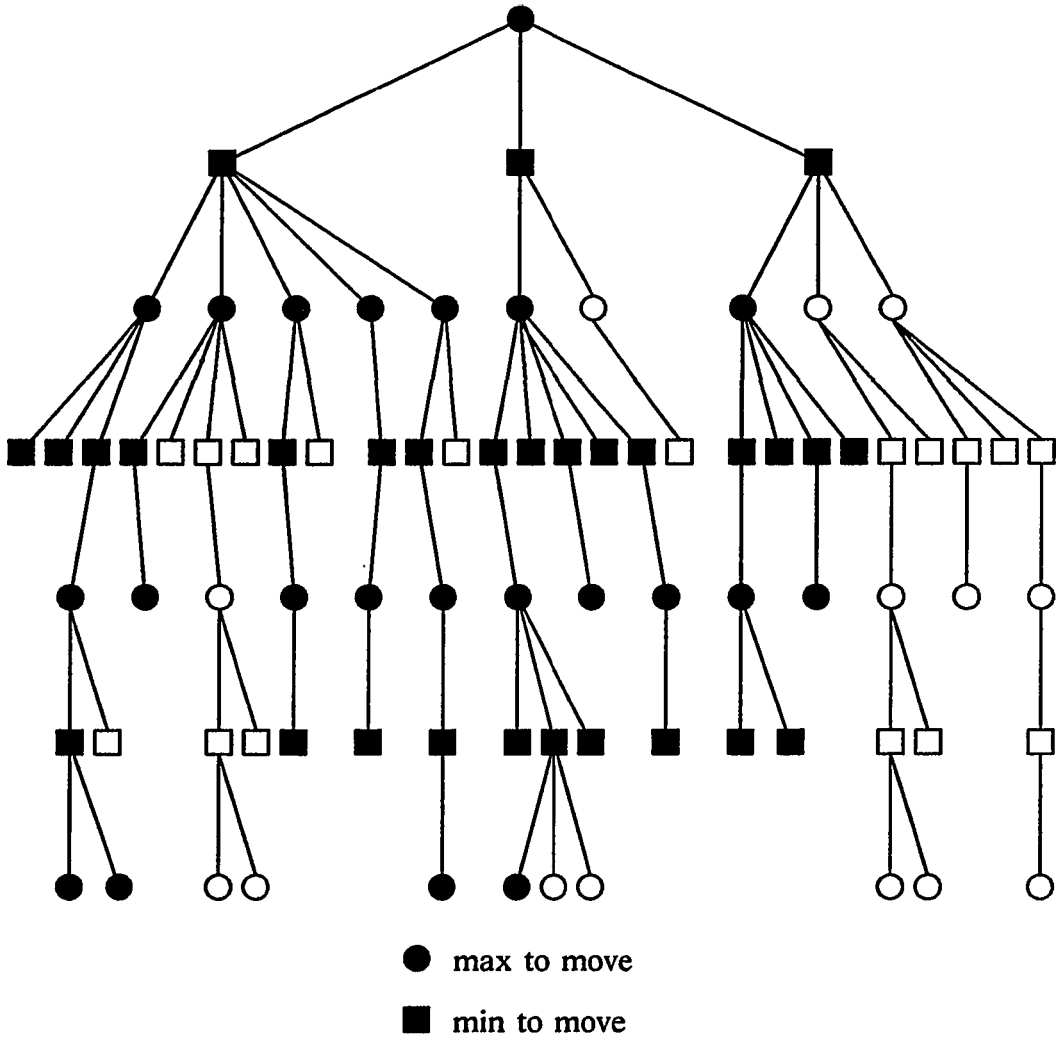

Figure 2.3: Alpha–Beta minimax game tree. Black circles and squares represent nodes examined. White circles and squares represent nodes not searched by the alpha–beta algorithm.

alpha–beta search; that is, the algorithm depends on sequential searching to establish the proper upper and lower search bounds.

Another not–so–obvious problem is that suppose we choose to examine moves one and two in parallel. Since the lower bound (score for the first move of 1000) is not known when examining the second move in parallel with the first, the resulting refutation (called a cutoff) does not happen. Again, the

parallel search examines more nodes to get the same result as the sequential search. In this example, both processors examine the same number of nodes and they do it in parallel, which takes only one-half of the time required to search these nodes sequentially. Unfortunately, the sequential alpha-beta algorithm does not examine ALL of these nodes. These extra nodes, called search overhead, are the nodes searched by a parallel algorithm but not by a sequential program. Chapter 4 mathematically quantifies this difference; suffice it to say for now that on occasion it is large enough to more than offset any benefit obtained by parallel processing.

Any parallel search that occurs when the upper or lower bound is unknown increases the size of the resulting tree beyond that searched by a sequential algorithm. This is the first tree searching example where parallel processing can increase the size of the tree, even though everything is technically done exactly right by the parallel algorithm.

2.5. Summary

The uniform, non-uniform, and alpha-beta trees become progressively more difficult for parallel implementations. Load balancing becomes nearly impossible when assigning the work without being able to determine the complexity of each of the branches. Searching branches in parallel without the knowledge of values for alpha and beta results in subtrees that are larger than their sequential counterparts.

To understand this behavior, chapter 4 mathematically examines the alpha-beta algorithm (for uniform trees) and then develops an approach to searching them in parallel.

CHAPTER 3

PARALLEL TREE SEARCH ISSUES AND PROBLEMS

3.1. Introduction

There are many issues and problems encountered when developing a parallel search algorithm [11, 17, 21, 23]. These problems cause reduced search efficiency at one extreme and cause non-deterministic search behavior at the other extreme.

Before developing parallel search algorithms, understanding how sequential search strategies and enhancements behave in a parallel environment is very important. Some dependable sequential search techniques suddenly fail or cause other detrimental search effects when ported to a parallel machine.

We analyze specific well-known sequential search techniques and enhancements in the following sections, paying particular attention to their behavior in a parallel environment.

3.2. The Transposition Table

The trees searched by game-playing programs are not perfect trees, but are more properly labeled graphs because many different branches can lead to the same node via a transposition of move sequences [17, 24]. In order to eliminate the redundant searches caused by examining the same subtree repeatedly (identical positions reached by different sequences of moves), the transposition table holds results for every subtree examined (or at least holds as many as will fit in available memory). In figure 3.1 the two nodes labeled X_1 and X_2 are identical positions reached by different sequences of moves. The

subtree below node X_2 (omitted from the diagram for clarity) is normally searched twice since this position occurs two times in the search. The transposition table saves the result from the first traversal of this subtree to avoid the second.

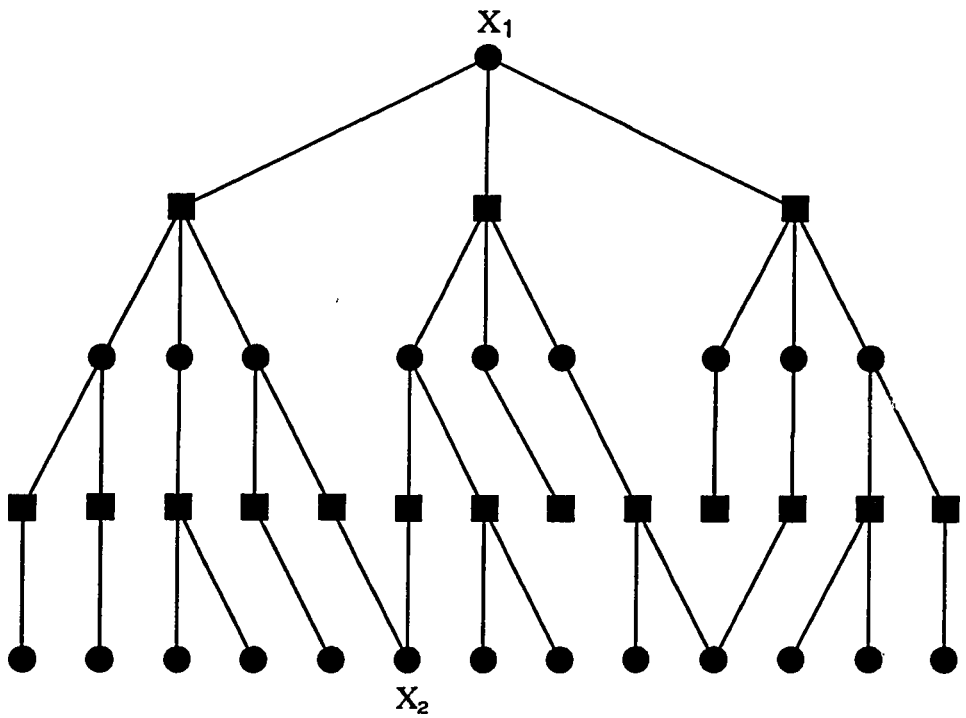

Figure 3.1: Transposition of moves results in identical position.

Since the table is not large enough to store all possible subtrees for searches run on machines like the Cray XMP or Cray YMP (where trees easily contain over 100,000,000 nodes) we require a replacement strategy to overwrite entries based on some measure of expected usefulness. This replacement strategy can be as simple or as complex as desired, and generally pays big dividends in terms of achieving maximum search efficiency. However, the simple idea of replacement begins to fall apart on parallel machines.

When two processors want to update or update/access the same position, a difficult arbitration problem develops. Also, when accessing the table by

hashing, what happens when two different positions hash to the same position in the table?

A secondary issue is the effect caused by replacing table entries when processors search large subtrees in parallel. Processors benefit most from saving positions that occur in positions reasonably similar to each other, such as those that occur in the same subtree. If two processors search two completely different branches (with different tactical and positional considerations) then the transposition information from one branch is not useful when searching the other. The processors then continually overwrite each other's table entries, resulting in a general search performance reduction. If very fine grained parallelism is possible (assuming low overhead from the system and hardware), then processors search subtrees that are more closely related and therefore share useful information through this table rather than sharing useless information and overwriting critical data while doing so.

The first transposition table problem to solve is selecting a replacement strategy. This is very important since finding a position in the table saves a significant amount of work. Storing the entry that will save the most work is a possible plan; alternatively, saving the entry with the highest probability of use is another reasonable option even though this saves less work.

The replacement strategy is not as important as the replacement process since this is where the non-deterministic quality slips in. When all processors are updating data in the table and retrieving data from the table, many timing dependencies exist (see figures 3.2 and 3.3). In figure 3.2, processor A can read position x and save a significant amount of searching since the position is in the table; in figure 3.3, processor B replaces position x with another position before processor A tries the same table retrieval. Now processor A cannot avoid

searching the resulting subtree since the table entry is unavailable; therefore, it searches this subtree and replaces many entries in the table also, possibly affecting other processors. This goes on and on and guarantees that running an eight-ply search ten times yields ten different node counts and ten different search times (sometimes dramatically different!).

time	processor a	processor b
1	read position x	-
2	discontinue searching	write position x
3	start new subtree	-

Figure 3.2: No processor interaction.

time	processor 1	processor b
1	-	write positon x
2	lookup position x and fail	-
3	continue searching this subtree	-

Figure 3.3: Processors interact with each other.

This is not the end of the problems, however. Processors can interact with each other in surprising ways with the transposition table handy to share search results immediately. It is possible for processors to operate in a synergistic manner where processor A barely completes a subtree search and enters it in the table in time for processor B to look the position up and save a lot of time. The next run may alter the timing of this store/retrieve sequence to a retrieve-failure/store sequence with the resulting larger tree since processor B

does not find the entry, forcing it to examine the complete subtree this time. Figures 3.4 and 3.5 illustrate this problem using the same timing diagram as before. In figure 3.4 processor B saves time by finding the information stored by processor A. In figure 3.5 processor B does not find the information in the table and continues searching, even though processor A places it there immediately AFTER the lookup fails.

time	processor 1	processor b
1	store results from subtree x	-
2	-	lookup position x
3	-	discontinue searching this branch

Figure 3.4: Synergism between processors.

time	processor 1	processor b
1	-	lookup position x and fail to find it
2	store results from position x	continue searching

Figure 3.5: No synergism between processors.

Documented cases exist of twenty-ply sequential searches returning search values that are impossible to obtain with a search of twenty plies [24]. The transposition table causes this by grafting a subtree representing ten plies onto another subtree representing fifteen plies. This depends on searching the ten ply subtree before searching the tree that needs this information so that this synergistic interaction is possible. The parallel case is even worse. In the 1984

ACM North American Computer Chess Championship, a parallel version of Cray Blitz made a strange move (that later won the game and tournament). It cannot reproduce this move with any time limit that we set. We suppose that some transposition table interaction similar to that described earlier caused the problem (this was an early parallel version of the chess program.)

3.3. The Killer Heuristic

The killer heuristic is a well-known strategy for alpha-beta tree searches that depends on accurate move ordering [21, 22]. This algorithm simply stores any move that causes a cutoff in the tree or any move backed up to any level in the tree as a new best branch. Programs generally keep such a list of moves for each different level in the tree.

The reasoning behind this algorithm is that when considering full-width (or brute-force) searches that examine all successor branches, there is a reasonably small set of moves that are "good." If a strong move M_1 occurs in position X, then this same move is probably good in most (if not all) positions that occur at this same depth in the tree. This is true because trying all moves at the previous level follows many branches that the same "good" move refutes. Of course there may be many good moves that refute a branch, but finding only one is sufficient for the alpha-beta algorithm.

The killer heuristic "remembers" a small set of good moves at each level and tries them first in the search at those levels. This allows the program to adapt to changing board conditions and remember good moves in the current configurations. The computational cost of the killer heuristic is virtually nil while the savings due to alpha-beta cutoffs caused by good move ordering is significant. Most running programs show a reduction in nodes examined by a factor of three to five when using this ordering strategy.

Parallel searching somewhat confuses this issue as each processor develops its own "killer" list while searching its subtree. The questions quickly follow: What to share with other processors? Which processor has the best killer moves? When should they share them (during the parallel search, at the synchronization point, when one causes a certain number of cutoffs?)

Since the transposition table is such a major contributor of non-deterministic behavior, disturbing anything that affects the number of nodes examined aggravates this problem even further. Sharing the killer moves during the parallel search rather than at a synchronization point introduces the possibility of timing sensitivity. If processor A finds a good killer and shares it with processor B just before processor B needs it, the search proceeds rapidly without including extra nodes and adding additional information in the transposition table. The next test run might let processor B get to the same point before processor A and try another move since the good killer move is not yet available. Trying an inferior move does not generate the quick cutoff obtained before (because the killer move is unknown), therefore the tree contains more nodes than the earlier one and also generates more transposition table entries that overwrite other entries.

It is generally felt that each processor should not share killer moves while searching in parallel, but rather that they merge the most popular killers from the entire pool into one list whenever processors complete their parallel search of a particular node. This tends to reduce the timing difficulties mentioned earlier, but it also gives up a possible reduction in tree search space made possible by sharing the killer list dynamically. This is particularly important at nodes deep into the tree since the processors have no prior killer move information to speed up the search.

3.4. The System Problem

After completing the parallel algorithm development, interface with the operating system (OS) begins. System overhead is critical since every cycle spent doing system tasks detracts from the computation power applied to the search problem. This directly affects the algorithms developed since it is usually too costly to create and discard tasks during the tree search [12, 23].

Instead of a simple 'fork' model we suddenly have to create tasks, keep up with them when there is no work, and then assign them to the correct place when work is available. Using the system FORK procedure to generate tasks and using some type of RETURN to kill the tasks introduces system overhead that can degrade algorithm performance drastically. Later analysis (given in chapter 4) suggests that reasonably fine-grained parallel processing yields the best results; however, if the overhead for creating and destroying processes is too high, the speedup is much less than optimal.

Sharing data is also non-trivial when designing algorithms for the general shared memory multiprocessor model. In some architectures (the Cray XMP for example) all processors can directly address the entire physical memory. This includes the so-called TASK COMMON (process local common) used by other processors. At the other end of this spectrum are machines like the Cray-2 and Cray-3 with real local memory accessible only by the processor connected to it.

Virtual memory machines such as the Sequent Balance 21000 and Symmetry can directly address all physical memory, but since each processor's local memory is in the same virtual address area, processors cannot access each other's local memory. Processor A copies the local data it wants to share with processor B into global (shared) memory where both processors can access it. This forces a processor to 'give' its local data to another, and prevents a

processor from 'taking' data from another processor's private or local memory. While this is a safe mechanism, it causes problems for dynamic algorithms that need to share data with minimal "handshaking."

This memory problem constrains the techniques used to share data. A processor on the Cray XMP can easily pass the address of its local memory to another processor which then freely reads/writes it as necessary (using some type of hardware synchronization to prevent interleaved updating and other types of problems). Running this same program on the Sequent generates incorrect results since passing the address of local common between processors fails because each processor has its own local common mapped into the same virtual address space (using different areas of physical memory).

Developing algorithms for the general shared memory multiprocessor model requires addressing these considerations in the algorithm design phase so that the problems do not 'slip in' and cause algorithm failures when transporting a working code to a new machine.

3.5. Non–Determinism

It should be obvious by now that one of the most important problems facing the designer of a parallel algorithm is debugging. The preceding problems generally result in non–reproducible results; the search usually returns the same suggested move, but the principle variation, score, and node counts vary from test run to test run. The timing variance causes slight differences to creep in during test runs, even on a dedicated machine. If the tests require several consecutive searches from the root position, these differences can cause the search to return completely different results from one run to another [5, 14, 23, 25, 27].

A parallel search to some fixed depth never returns a search result worse than that returned by the sequential search. The parallel search commonly returns results BETTER than the sequential search because of the synergistic behavior discussed earlier. However, when debugging these large and complex algorithms, constant behavior simplifies the testing and debugging problem. Eliminating the non-determinism causes a reduction in the quality of the search results since the synergistic effect is lost. It is not clear that eliminating the non-determinism is possible with the timing variances that regularly occur in parallel machines since the operating system constantly responds to external interrupts (such as network traffic, timer interrupts, I/O interrupts, etc.); however, eliminating this behavior adversely affects the search results due to the excessive synchronization overhead required.

3.6. Summary

Normal sequential algorithms exhibit strange and bizarre behavior when used in a parallel environment. Non-deterministic behavior makes testing and debugging extremely difficult since the timing data and results are often non-reproducible.

Many researchers find that the best sequential algorithms perform poorly in a parallel environment and this seems to carry through to the parallel search of game trees.

With large numbers of processors, it is much more important to keep them all busy than it is to minimize the search space. Search space cannot be ignored, but the computational power lost when using efficient sequential algorithms on parallel machines directly affects the total performance of the system.

Chapter 4 gives precise mathematical analysis of the alpha-beta problem with particular emphasis on the parallel search performance issue. It seems to hint that spectacular parallel performance is possible although previous results fall short of this goal.

CHAPTER 4

MATHEMATICAL ANALYSIS OF ALPHA–BETA

4.1. Minimax Game Tree Search

In a minimax search to depth d with w possible successor branches at each node of the tree, equation (1) defines N, the number of nodes examined.

$$N = W^d \tag{1}$$

4.2. Alpha–Beta Game Tree Search

Knuth and Moore proved that the alpha/beta algorithm reduces this number of nodes significantly[13]. Assuming perfect move ordering, the alpha/beta algorithm reduces N to the value $N_{\alpha\beta}$ defined by:

$$N_{\alpha\beta} = w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1 \tag{2}$$

This reduction in total nodes examined allows searching to a depth that is unreachable without the alpha–beta algorithm.

In defining this formula, Knuth and Moore classified all nodes occurring in an alpha–beta tree into one of three types:

> Type 1: The root position is a type 1 node. The first successor of a type 1 node is itself a type 1 node, while all other successors of a type 1 node are type 2 nodes. Type 1 nodes have all of their successors examined.
>
> Type 2: A type 2 node is a successor of a type 1 node (as given above) or a type 3 node. A type 2 node only has one successor examined (for perfectly ordered game trees).
>
> Type 3: A type 3 node is a successor of a type 2 node and has all of its successor branches examined. The ordering of branches from a type 3 node is not important to the efficiency of the search.

It is important to note that the number of successors given above is correct only when perfect move ordering occurs (except for type 3 nodes). For example, if the first branch chosen from a type 1 node is not the best one, the resulting position is still searched as though it is a type 1 node (by rule 1 above). Later however, when searching the best branch from the type 1 node, it also is a type 1 node (rather than a type 2 node) which results in examining extra nodes. This happens because the lower/upper bound is incorrectly fixed by the poor original successor to the type 1 node. The actual best branch correctly establishes the bound but searches additional branches that are normally pruned. Ordering at a type 2 node is also important since examining the best (or at least a good) successor is necessary. If this is not done, the search examines unnecessary successors before finding a refutation move.

As an example, noticing that the first successor of a type 1 node is itself a type 1 node and that other successors of a type 1 node are type 2 nodes, a parallel algorithm might examine one of the supposedly type 2 nodes before the first node at that level (a type 1 node) is completely examined. The result would be that the lower/upper bound is not properly identified, causing the type 2 node to behave like a type 1 node requiring the examination of all successors. This node typing therefore has a sequential property that can introduce a surprising amount of overhead (that is difficult if not impossible to eliminate) when it is used in a parellel tree searching algorithmn.

4.3. Parallel Alpha–Beta Search

The following sections show that for parallel implementations, $N_{\alpha\beta}$ increases well beyond the limit given in (2) above if care is not taken. First, equation (2) requires perfect move ordering within the tree (for successors of type 1 and type 2 nodes). While this is not always possible, well-known

algorithms exist that provide a reasonable approximation to this ordering. Second, the alpha/beta algorithm uses information obtained from the first branch of type 1 and type 2 nodes to reduce the work done when searching the remaining branches. In particular, examining the first branch (at some level) establishes the lower(upper) search bound for the rest of the branches at that level since the minimax algorithm never chooses a move that is worse (better) than one already examined.

Unfortunately, parallel searching of a minimax tree using the alpha/beta algorithm violates this principle because at some node in the tree, the code examines successor two (or some other successor) in parallel with successor one so that the value (lower/upper bound) for move one is not available. This difficulty with the parallel search tends to increase the value of $N_{a\beta}$. Without proper control, this increase is large enough that the additional time spent examining the extra nodes will offset the time saved by searching multiple tree branches in parallel.

It can be shown from equation (2) that the number of nodes examined on the first branch taken (when no lower search bound is known), denoted N_{fb}, is given by the following equation:

$$N_{fb} = w^{\lfloor (d-1)/2 \rfloor} + w^{\lceil (d-1)/2 \rceil} - 1 \qquad (3)$$

The proof is as follows: equation (2) defines the minimum number of nodes examined from a type 1 node with depth = d. Since the first descendent of a type 1 node is also a type 1 node, reducing d by 1 defines the minimum number of nodes examined for a tree that is one level shallower. From this equation and equation (2), the number of nodes contained in each of the remaining branches, denoted N_{rb}, is:

Subtracting (3) from (2) generates this equation. Since there are w–1 branches

$$N_{rb} = \frac{w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - w^{\lfloor (d-1)/2 \rfloor} - w^{\lceil (d-1)/2 \rceil}}{w - 1} \quad (4)$$

left at the root position after examining the first branch, each of these remaining branches contains $1/(w-1)$ of the total nodes left to examine after the first branch. By comparing equation (3) with equation (4), the first branch examined (with no lower/upper bounds established) will contain significantly more nodes than branches examined after the first one. For an eight ply search of a game tree from chess, assuming $w = 38$, the first branch contains 2,140,007 nodes, and the remaining 37 branches (from the root position) contain 54,872 nodes each.

4.4. Principle Variation Splitting (PVS)

Since the first branch of the tree contains significantly more nodes than any other branch, it seems reasonable that the processors search different parts of this branch in parallel. Avoiding searching more than one root branch without a known lower/upper search bound (alpha/beta) is an obvious benefit of this algorithm.

By applying equations (3) and (4) to the subtree comprising the first branch examined from the root position, the first branch examined in this subtree will also have significantly more nodes than the remainder of the branches that must be examined. This holds true for the first branch of every node examined where the lower/upper bounds of the search are unknown (type 1 nodes).

From the preceding discussion, the search must avoid parallel division of work at nodes where the lower/upper search bounds are unknown. If parallel division does occur at such a node, then each processor searches a subtree containing N_{fb} nodes first, and therefore examines more than the minimum

number of nodes ($N_{\alpha\beta}$). Since violating this principle at some point in the tree is necessary to use all available processors, the division must occur as deeply in the tree as practical in order to minimize the difference between N_{fb} and N_{rb}.

The real purpose of the PVS algorithm [5, 12, 19, 25] is to minimize the difference $N_{fb} - N_{rb}$. By inspection, this occurs when the exponent d is as small as possible when the upper/lower search bounds are unknown. For example, when executing an eight ply search, if the parallel division of the tree is done at depth eight first, then each subtree (ignoring the quiescence search) would only have one node. This would improve the performance of the parallel search since parallel or sequential searches would contain the same number of nodes as discussed earlier.

Next we modify equations (3) and (4) to determine the number of nodes examined by the PVS algorithm. Let *dp* represent the depth where the parallel division occurs. Since d in equations (3) and (4) represents the depth of the tree, the corresponding value for the PVS algorithm, d–dp+1, represents the depth of the subtree from the point of division. The resulting equations are:

$$N_{fb} = w^{\lfloor (dv)/2 \rfloor} + w^{\lceil (dv)/2 \rceil} - 1 \tag{5}$$

$$N_{rb} = w^{\lfloor (dv+1)/2 \rfloor} + w^{\lceil (dv+1)/2 \rceil} - w^{\lfloor dv/2 \rfloor} - w^{\lceil dv/2 \rceil}/(w-1) \tag{6}$$

In these equations, replace dv with d–dp. These are then combined with equation (3) to define the number of nodes in a tree using the PVS algorithm and dividing the tree at depth dp using p processors.

$$N_{pvs} = N_{\alpha\beta} + (p-1) * (N_{fb} - N_{rb}) \tag{7}$$

This can be proven by the following informal reasoning. $N_{\alpha\beta}$ in the equation defines the number of nodes in a perfect alpha/beta tree. The term $(p-1) * N_{fb}$ comes from equation (5) and defines the number of nodes p

processors will look at on the first branch of level *dp*, since the first descendent of a type 1 node is itself a type 1 node. If each processor takes a descendent of a type 1 node to examine, then these will be type 1 nodes themselves, resulting in a larger than optimal tree. The term $(p-1) * (-N_{rb})$ comes from equation (6) and defines the number of nodes remaining after each processor searches exactly one branch at the point of division (*dp*). When searching *p*–1 extra type 1 nodes, it follows that *p*–1 type 2 nodes disappear since these type 2 nodes transform into type 1 nodes because of the parallel division.

In analyzing this equation, as *dp* approaches *d*, the number of extra nodes approaches zero since a type 1 node has the same number of successors at the terminal depth as does any other type of node (none). At the tip of the tree all branches (including the first one) result in terminal positions that require no additional searching.

From this discussion, it follows that *dp* should be as close to *d* as possible in order to maximize the performance improvement made possible by the PVS parallel searching algorithm. Equations (3) and (4) combined with the PVS algorithm shows that if *dp* is chosen correctly (*dp* = depth of search), then (3) + (4) result in the same number of nodes as defined by (2). This implies that a parallel search with NO additional nodes is possible.

An unfortunate side–effect of such a choice for *dp* is that communication and coordination between processors must occur at depth *dp*. At this point in the tree, the processors share a list of branches to examine and access some type of shared data structure to prevent different processors from choosing the same branch. As *dp* increases (reducing the size of the subtree a processor searches) a corresponding increase in coordination and communication traffic at depth *dp* occurs. If the communication cost is high or the coordination is not efficient, the

time lost results in poor processor utilization even though examining few (or even no) extra nodes.

4.5. Summary

For this algorithm to be most useful, *dp* must approach *d*. This requires that the multiprocessing architecture supports efficient sharing of resources and processor synchronization. Additionally, a processor must start processing immediately whenever reaching a node where work is to be done in parallel. The speed of these operations directly influences the overhead introduced as *dp* approaches the value of *d*.

To summarize, the preceding analysis predicts reasonable parallel performance of the alpha-beta algorithm. However, avoiding searching extra nodes is not always easy and straightforward.

Chapters 5 through 7 use this mathematical analysis to develop successively better algorithms for the parallel search of game trees. All of these algorithms struggle to minimize the tree size based on these mathematical premises while at the same time keeping all processors busy.

CHAPTER 5

PRINCIPAL VARIATION SPLITTING (PVS)

5.1. Cray Blitz

Coding and testing algorithms for a parallel search requires a working tree search program of some sort. The availability of Cray Blitz, a computer chess program, eliminated the need for developing such a program first. A secondary motive for using Cray Blitz is the active interest in making it as fast as possible, particularly when considering that Cray Blitz runs on the fastest available supercomputers.

5.2. The PVS Algorithm

The most widely used algorithm for searching game trees in parallel (alpha/beta trees specifically) is the Principal Variation Splitting (PVS) algorithm [1, 2, 5, 6, 7, 8, 11,12, 19, 20, 21, 22, 23, 25, 27]. Developed simultaneously at several different universities, it is a natural result of the preceding analysis.

The name comes from the "principal variation" idea in games where the best sequence of moves from the root position is known as the principal variation. Game programs take great pains to properly order the tree to maximize the alpha-beta cutoffs that occur, generally resulting in nearly optimal move ordering. If these ordering strategies are successful, then the first node at each successive level in the tree is of type 1 (from the mathematical analysis presented earlier.) These nodes represent ideal locations for parallel processing since the search examines all successors of type 1 nodes.

When executing a search to depth eight, one processor traverses the first seven nodes (one at each successive depth) and then divides the work at depth eight among the available processors. Since this node is a type 1 node, examining all successors in parallel occurs with no loss of time. When a type 1 node at depth d is completely examined, all processors back up to depth d–1 and search the remaining branches (at that depth) in parallel. This process continues until the group of processors reach depth one and search the entire list of alternatives at that level.

This algorithm has little risk in terms of search overhead since parallel splitting occurs only at type 1 nodes. The only circumstance that sometimes causes extra search overhead is the improper ordering of branches somewhere in the tree so that a type 1 node does not have all successors examined (it would not be a type 1 node even though it matches the definition because move ordering is not correct.)

The difficulty with this algorithm lies in the non–uniformity of the game tree. Since the processors search successors of the same node in parallel, they must finish before returning to the previous depth to continue the parallel division. If some of the processors return to the previous level too soon, one of the remaining busy processors might return a better bound to the current depth. This causes the processors that returned too soon and started searching without the proper search bounds to possibly (or probably) search extra nodes as a result. The additional nodes searched make this algorithm unattractive; however, eliminating this search overhead by not allowing any of the processors to start work at the previous level until completing the current level causes another type of problem: increasing the synchronization overhead.

Synchronization overhead is a result of one processor (or more than one) waiting on another to complete some type of work. Since the trees are not uniform, this synchronization overhead accumulates very quickly. Technically, it is the primary reason that variations of the PVS algorithm produce less than optimal speedups when compared with the maximum performance obtainable by parallel processing.

This algorithm produces reasonable performance (a speedup of five to eight) with careful tweaking to balance search overhead caused by lack of synchronization with synchronization overhead caused by trying to reduce the search overhead. This is a circular type of problem where improving one area adversely affects the other.

A more general defect in this algorithm is that all processors work at the same node within the tree. For machines with large numbers of processors, it is not unreasonable to expect that cases exist where there are more processors than branches. If this happens, the parallel search starts at that point with processors already idle, rapidly accumulating large amounts of synchronization overhead. Even for current levels of parallelism, as the games progress (chess, checkers, go, etc.) the board becomes simpler with the resulting less complex (lower branching factor) trees. Here, the parallel performance steadily drops lower and lower as the game progresses.

5.3. PVS Results With Uniform Game Trees

An old version of Cray Blitz uses the PVS algorithm just described. These tests were run on a Sequent Balance 21000 computer with thirty NS32032 microprocessors and 16mb of shared memory.

This implementation of the PVS algorithm assigns the current search depth to the variable dp. In these tests, dp was set to cause the first parallel split

along the principal variation to occur at the maximum depth of the search where all nodes are terminal except for those expanded by the quiescence search.

We modified the Cray Blitz tree search in the following ways for these tests. The move generator produces a constant number of moves at each position (this parameter varies between 2 and 64 for these tests). We disabled the quiescence search so that no branch extends deeper than any other branch. These changes generate perfectly uniform trees. Additionally, we disabled the alpha–beta algorithms for these tests making the code examine uniform game trees using the minimax algorithm.

		time in seconds					speedup			
w	d	1	2	4	8	16	2	4	8	16
2	16	575	581	584	590	594	0.99	0.98	0.97	0.97
4	9	795	537	275	278	277	1.48	2.89	2.86	2.87
8	6	523	301	154	78	84	1.74	3.40	6.71	6.23
16	5	1707	924	464	233	141	1.85	3.68	7.33	12.11
32	4	1563	819	410	2028	112	1.91	3.81	7.51	13.96
64	3	357	183	94	46	25	1.95	3.80	7.76	14.28
tot		5520	3345	1981	1433	1233				
avg		920	557	330	238	2205	1.65	3.09	5.52	8.40

Table 5.1: PVS algorithm speedup when searching uniform game trees.

Table 5.1 gives the results obtained from using the PVS algorithm and varying the numbers of processors to execute an exhaustive search on a test position. The branching factor varies from two successors per node up to sixty–four successors per node. This data is graphically summarized in figure 5.1.

Table 5.1 shows that when dividing the work up at only one node at a time, the branching factor serves as an upper bound for the speedup obtained. This obvious conclusion is supported by the results produced by smaller branching factors. As most games progress they become simpler with a steadily

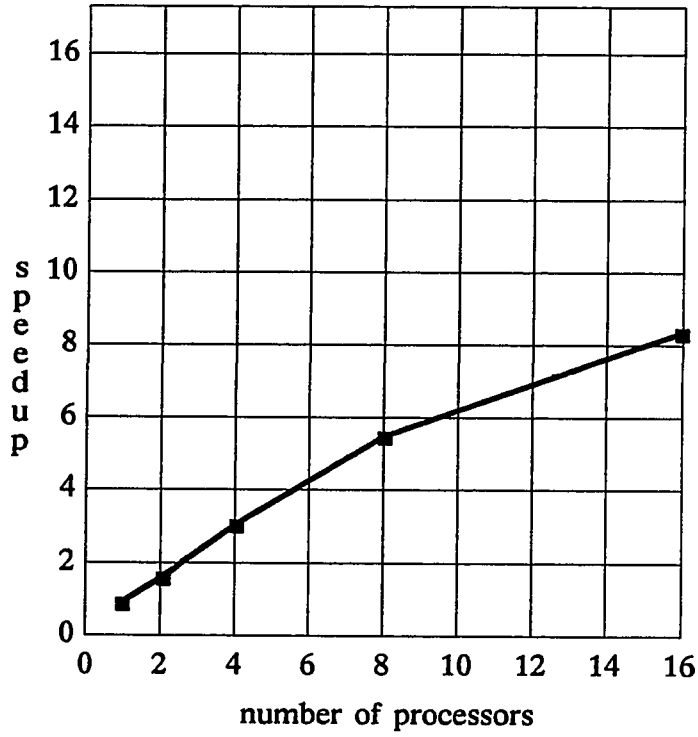

Figure 5.1: PVS algorithm speedup on uniform game trees.

reducing branching factor in their trees. When there are more processors than branches, performance suffers since the extra processors sit idle.

Several conclusions follow from this uniform tree search data. For very small branching factors (two is an example), the parallel processing overhead of the PVS algorithm causes less than optimal performance. Specifically, the PVS algorithm (with more than one processor) provides no performance advantage over the sequential algorithm.

This broad variation in branching factors represents the types of trees encountered in games like chess and checkers. The narrow branching factors represent the trees found early and late in the game or when captures are possible in the game of checkers, and the wide branching factors represent the trees found in the middle of the game when the positions are most complicated in chess or no captures are possible in checkers.

5.4. PVS Results With Non-Uniform Game Trees

Next, we modified Cray Blitz so that it searches non-uniform minimax game trees. Since non-determinism makes testing difficult, Cray Blitz uses the relative position of a branch to determine the branching factor of the node it produces. In this manner, searching the branches in any order still produces the same search tree.

		time in seconds					speedup			
w	d	1	2	4	8	16	2	4	8	16
2	16	351	360	391	453	581	0.98	0.90	0.77	0.60
4	9	294	242	217	229	288	1.21	1.35	1.28	1.02
8	6	503	336	252	224	271	1.50	2.00	2.25	1.86
16	5	1869	1064	696	522	490	1.76	2.69	3.58	3.81
32	4	3838	2071	1170	738	585	1.85	3.28	5.20	6.56
64	3	3462	1850	1013	622	447	1.87	3.42	5.57	7.74
tot		10317	5923	3739	2788	2662				
avg		1719	987	623	464	443	1.53	2.27	3.11	3.60

Table 5.2: PVS algorithm speedup when searching non-uniform game trees with varying depths (d) and branching width (w).

For non-uniformity in depth, the program searches a wildly tactical position with several queens and a king on each side. The many checking variations (along with those that directly lead to checkmates and draws) cause the desired non-uniformity in depth and even affects the non-uniformity in width on occasion. The alpha-beta algorithm was left disabled so that the program searches minimax trees for this test.

Table 5.2 gives the results obtained when using the PVS algorithm and searching non-uniform game trees. The column labeled w gives the average branching factor for this test. For example, if w = 8 and a node produces eight branches, the first branch has one only one successor, the second branch has two successors, and so forth. The actual branching factor averages (w+1)/2. This data is graphically summarized in figure 5.2.

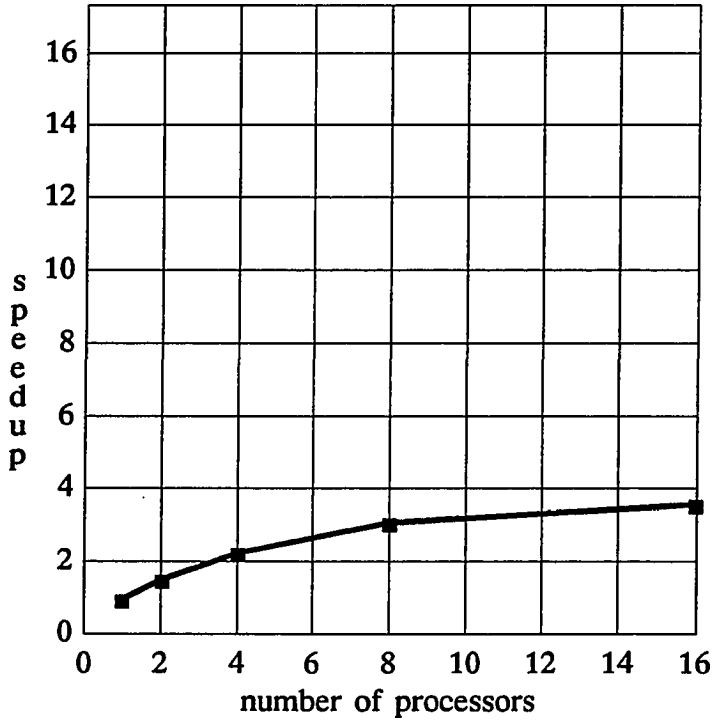

Figure 5.2: PVS algorithm speedup on non-uniform game trees.

The non-uniform tree test illustrates the problem with the PVS algorithm. Whenever the branching factor is less than the number of processors, the speedup cannot exceed the branching factor. Since many branches are quite narrow in trees such as these, the speedup shows even less improvement, particularly with more and more processors available. For future machines with hundreds of processors, these algorithms are totally unsuitable.

5.5. PVS Results With Non-Uniform Alpha-Beta Game Trees

Tables 5.3 and 5.4 give the results obtained when using the PVS algorithm and varying the number of processors to execute an exhaustive 5-ply alpha-beta search on a well known set of positions [23]. Table 5.3 gives the node counts for the one, two, four, eight and sixteen processor test using the Bratko-Kopec problem set. The total and avg rows at the bottom of each table summarize the results and also smooth out the somewhat unusual data values

collected for some of the test cases. Figure 5.3 depicts this data graphically and shows the minimal search space increase when adding additional processors.

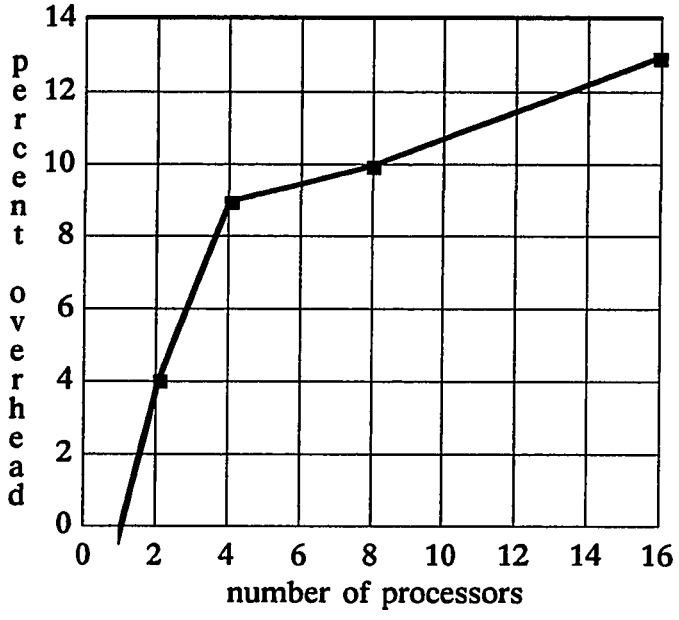

Figure 5.3: Percent increase of search overhead by number of processors.

This data illustrates several interesting attributes of a parallel search. Notice that the node counts do not uniformly increase as the number of processors increases, but that they sometimes decrease by a significant amount. This results from the non-deterministic behavior in the shared transposition where all processors update and use information obtained by other processors in an unsynchronized manner (described earlier).

Notice also that the node counts occasionally increase dramatically when adding processors, sometimes causing the search to take longer with more processors. Repeatedly running the same tests generally eliminates such behavior; therefore, it makes more sense to use a statistical average of several test runs to smooth out these anomalies although in these test cases the unusual totals were left in to illustrate this behavior.

The performance data from table 5.3 shows that additional processors increase the size of the tree by a small amount; this is offset by the benefit gained in terms of search time shown in table 5.4.

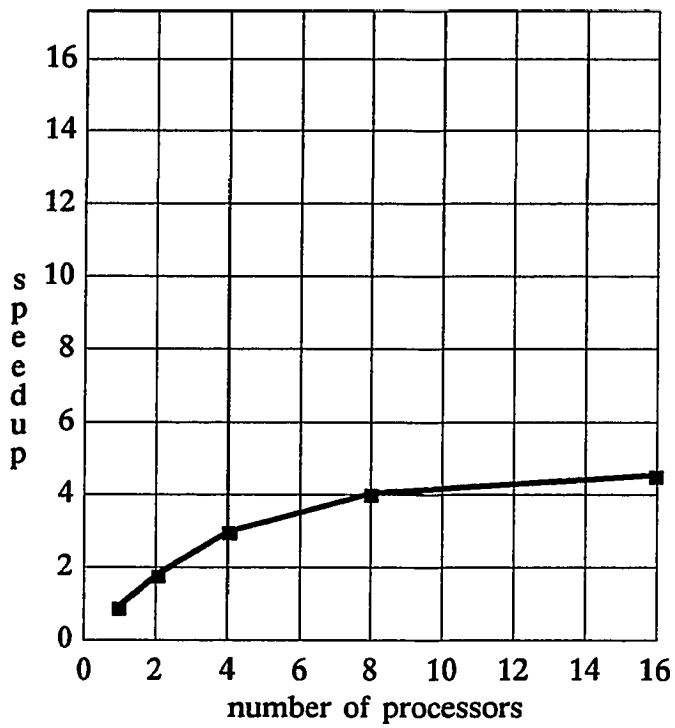

Figure 5.4: Speedup as number of processors is increased.

Figure 5.4 condenses the data from table 5.4 and shows the performance speedup produced by the PVS algorithm. The PVS algorithm is quite efficient because it results in little extra work being done. The average speedup for two, four, eight and sixteen processors is also interesting in that the performance curve flattens rapidly beyond four processors.

Two positions result in anomalies caused by the short amount of time required to complete a five-ply search. In particular, problem one finds the solution at a depth shallow enough that parallel processing is almost useless and problem eight has such a low branching factor that most processors have no work to do. The performance figures for problem one were excluded from the

tables since it is a special case and has been treated similarly by other researchers.

	total nodes				
pos	1cpu	2cpu	4cpu	8cpu	16cpu
1	---	---	---	---	---
2	90337	90395	93968	91340	96479
3	29529	29528	29834	30215	30287
4	87411	82994	86938	82943	86214
5	153900	163048	166317	174911	171384
6	13379	13516	13904	14157	13962
7	50675	51405	54675	56167	48441
8	6530	6665	6699	6833	7447
9	75555	79260	77098	86639	55863
10	96369	98214	112360	96749	127496
11	56960	57261	59275	59759	60873
12	197945	206202	213085	214776	219764
13	66899	74519	84702	74484	94951
14	56660	57816	57547	59512	56364
15	53384	53913	54267	55665	56329
16	75483	82509	86909	96321	97722
17	58624	76723	89125	88307	69466
18	120988	124374	137831	138693	146334
19	62071	58784	61657	60079	61091
20	97537	98425	106257	111466	151428
21	141686	158172	157725	159302	162938
22	67291	66150	68198	69649	63782
23	50081	50552	52070	52925	55568
24	73479	73749	73676	74221	74497
tot	1782773	1854174	1944117	1955113	2008680
avg	77512	80616	84527	85005	87334

Table 5.3: PVS algorithm node counts.

The table row labeled "mse" is the mean standard error and represents the variability of the timing data that is caused by the non-deterministic behavior of the parallel algorithm. This table (and similar tables in later chapters) represent average timing over many test runs. This data provides an estimate of how much the timing varies from run to run with absolutely no changes to the algorithm, test data or hardware.

pos	time in seconds 1cpu	2cpu	4cpu	8cpu	16cpu	speedup 2cpu	4cpu	8cpu	16cpu
1	--	--	--	--	--	---	---	---	---
2	970	508	278	210	201	1.91	3.49	4.62	4.82
3	375	205	124	93	87	1.83	3.02	4.03	4.31
4	1950	972	674	487	477	2.01	2.89	4.00	4.08
5	3193	1927	1035	986	971	1.66	3.09	3.24	3.28
6	141	77	50	41	41	1.83	2.82	3.44	3.44
7	584	312	190	134	111	1.87	3.07	4.36	4.36
8	75	46	34	42	42	1.63	2.21	1.79	1.79
9	958	554	315	263	122	1.73	3.04	3.64	3.64
10	1174	606	385	229	317	1.94	3.05	5.13	5.13
11	697	364	208	136	132	1.91	3.35	5.13	5.28
12	3763	2079	1131	675	439	1.81	3.33	5.57	8.57
13	652	374	239	148	175	1.74	2.73	4.41	3.73
14	1068	569	323	252	234	1.88	3.31	4.24	4.56
15	939	484	264	241	243	1.94	3.56	3.90	3.86
16	1283	732	526	436	417	1.75	2.44	2.94	3.08
17	737	514	442	366	128	1.43	1.67	2.01	4.66
18	1977	1042	659	460	461	1.90	3.00	4.30	4.29
19	741	419	316	255	249	1.77	2.34	2.91	2.98
20	2009	1060	596	393	371	1.90	3.37	5.11	5.42
21	2706	1458	811	554	549	1.86	3.34	4.88	4.93
22	840	426	280	236	120	1.97	3.00	3.56	7.00
23	600	325	188	124	113	1.85	3.19	4.84	5.31
24	758	391	217	146	140	1.94	3.49	5.19	5.41
tot	28190	15444	9285	6907	6143	1.83	3.04	4.08	4.59
mse	0	3	10	35	44	.00	.01	.05	.11

Table 5.4: PVS algorithm speedup.

5.6. PVS Performance Summary

The PVS algorithm suffers from the defect of dividing one node at a time among the various processors. If a chosen node does not have enough branches, processors sit idle causing performance degradation.

To remedy this, we describe the enhanced PVS (EPVS) algorithm in chapter 6. This algorithm attempts to address this problem and reduce this idle time.

CHAPTER 6

ENHANCED PRINCIPAL VARIATION SPLITTING (EPVS)

6.1. The Enhanced PVS Algorithm (EPVS)

As described, the PVS algorithm is efficient for trees that are perfectly ordered (for reasonably small numbers of processors). The PVS algorithm has one particular failing that greatly reduces the performance obtained on a parallel machine. If the branches given to various processors for parallel examination are not roughly equivalent in the amount of work required to analyze them, then some processors sit idle waiting on others to finish examination of the last few branches [5, 12, 19, 20, 23].

Even though move ordering cannot be perfect, there are several algorithms that generate good ordering without requiring an unacceptable amount of execution time. One of these, the iterated or staged search [2, 5, 10, 11, 18] provides good move ordering by first doing a shallow search and then using information from this shallow search to improve the move ordering for a deeper search. However, since the ordering is not perfect, at some point when using the move ordering information from a depth d search to execute a depth d+1 search, the depth d+1 search will discover that a different move is best.

We expect this "change of heart" since the reason for doing the depth d+1 search is to improve upon the information gained from the depth d search. Every time the search finds a new best move for the same depth, it examines additional (extra) nodes ($N_{fb} - N_{rb}$ where dp = 1 to be exact). Finding a better move at depth d+1 is the only justification for spending the time necessary to

complete the search. However, an unfortunate performance penalty occurs since only one processor is searching this soon to be best move (recall that all processors eventually split up the moves at ply=1 and then synchronize before starting the next search one ply deeper.)

If all processors work on the same node (as in PVS), a potential problem of load balancing occurs when giving one processor a task that is significantly more complex to do than those given to others. Whenever a processor runs out of work while other processor(s) are busy, that portion of the total computational power of the computer system is being lost.

In order to minimize this problem, we now modify the PVS algorithm to address this case. The enhanced PVS (EPVS) algorithm detects the condition where a processor becomes idle at any node in the tree. When a processor can find no additional work to do at the current divide node, it assumes that the remaining busy processors are searching complex branches and sends a stop signal to them. The sequential search then advances two plies deeper (following one of the complex branches deeper into the tree) before restarting the parallel search. This allows all processors to work on the same subtree. The transposition table [24] keeps the set of processors from re-examining branches within this subtree already examined by a single processor before it honored the resplit request.

This algorithm is recursive in that if the case again arises where one processor becomes idle, the group will go two plies deeper into the remaining subtree to re-divide the work up again. Each time this occurs, the division point (dp) moves closer to the maximum depth of the search, resulting in smaller and smaller subtrees for each processor to examine in parallel.

This has proven to be quite effective as will be seen in the results section. This enhancement makes parallel processing with more than four processors yield reasonable performance whereas the regular PVS algorithm shows little benefit with more than four processors.

With this modification, there is a risk associated with the tree search. If the move ordering is less than perfect, then the point chosen to re-divide the tree search might not be a type 1 or type 3 node. If this occurs, extra work will be done causing the corresponding loss of performance. This small loss of efficiency is offset by the additional computational speed gained by keeping the processors busy.

6.2. EPVS Results With Uniform Game Trees

For this test, we use the same modified tree search from section 5.3 which produces a uniform game tree. The program searches the same position used in section 5.3 and the corresponding results appear in table 6.1. This data is not summarized in a figure because of the cases where the speedup was nearly zero when there were more processors than branches at most node.

		time in seconds					speedup			
w	d	1	2	4	8	16	2	4	8	16
2	16	575	475	–	–	–	1.21	--	--	--
4	9	795	439	400	–	–	1.81	1.99	--	--
8	6	523	268	150	–	–	1.95	3.49	--	--
16	5	1707	861	433	538	–	1.98	3.94	3.17	--
32	4	1563	780	395	201	686	2.00	3.96	7.78	2.28
64	3	357	183	92	42	49	1.95	3.88	8.50	7.29

Table 6.1: EPVS algorithm results when searching uniform minimax game trees. -- entries indicate that the search takes an excessive amount of time compared to the sequential search.

In table 6.1, the columns labeled "--" indicate that the corresponding speedup is less than one. For this reason, totals and averages are not given since

the results are meaningless. These columns represent the cases where the search "blows up" and requires excessive time when compared to the sequential algorithm.

The EPVS algorithm is very sensitive to the ratio of the number of processors to the number of moves (branching factor). As this ratio grows beyond one, the overhead increases tremendously. Idle processors (which occur at EVERY node) constantly request resplits. As the busy processors stop and honor these requests, the next resplit operation immediately results in additional resplit requests since idle processors report in at every node.

For current supercomputers like the Cray XMP which has four processors, the search never "blows up" as it does in this test. As new machines with more processors arrive however, we expect significant problems with this algorithm when searching simple trees like those that occur in endgame positions.

EPVS shows little improvement over PVS for uniform trees. In fact, due to its propensity for "blowing up" when the number of processors exceeds the number of branches, it produces results worse than the basic PVS algorithm. The next section applies the EPVS to the more difficult non-uniform tree search problem where the results are more favorable.

6.3. EPVS Results With Non-Uniform Game Trees

Table 6.2 shows the performance for the same non-uniform test given in section 5.4. The EPVS algorithm does better on this test since the non-uniform character of the search produces both complex and simple branches from the same parent node.

Once again, the EPVS algorithm shows a propensity for "blowing up" when the number of processors exceeds the number of branches at a node. The

non-uniformity smoothes this out here since the resplitting operation usually finds a complex node to split.

In this test case, the algorithm shows that it can offer significant performance improvement over the PVS algorithm, but only when the number of processors is less than the branching factor. Violating this requirement sometimes results in search times an order of magnitude larger than the sequential search time.

		time in seconds					speedup			
w	d	1	2	4	8	16	2	4	8	16
2	16	351	321	401	–	–	1.09	0.88	—	—
4	9	294	231	198	–	–	1.27	1.48	—	—
8	6	503	307	212	227	–	1.67	2.37	2.22	—
16	5	1869	1013	599	421	467	1.85	3.12	4.44	4.00
32	4	3838	2020	1106	643	701	1.90	3.47	5.97	5.48
64	3	3462	1795	940	565	376	1.93	3.68	6.13	9.21

Table 6.2: EPVS algorithm results when searching non-uniform minimax game trees with varying depths (d) and branching width (w). — entries indicate that the search time takes an excessive amount of time.

In chess, except for very simple endgame positions, this excessive search overhead problem does not occur often enough to render this algorithm unusable. While very narrow branches occur during the course of the tree search, they occur infrequently, or at least they occur rarely enough that the search does not "blow up" in actual play as it does in these examples. However, the potential for disaster does exist with this algorithm as these tables show. If the program spends most of its time executing overhead code, it can surely make a tactical mistake due to the reduced search depth it obtains when wasting so much time, resulting in a significant risk that the program might make a tactical blunder due to the reduced search depth caused by this overhead. In such a case, the parallel algorithm might actually lose where the sequential algorithm would not.

6.4. EPVS Results With Non-Uniform Alpha-Beta Game Trees

This section tests the EPVS algorithm on the most difficult test tree of all, the alpha-beta tree. The increased search depth made possible by this algorithm greatly increases the effectiveness of the EPVS algorithm and seems to avoid the conditions that cause the search to "blow up."

Tables 6.3 and 6.4 provide the same information for the EPVS algorithm as that presented in chapter 5 for the PVS algorithm. It is important to note that even though EPVS is faster than PVS, the performance gain comes from reducing processor idle time and adds virtually no additional nodes to the tree. The data in tables 6.3 and 6.4 is graphically summarized in figures 6.1 and 6.2.

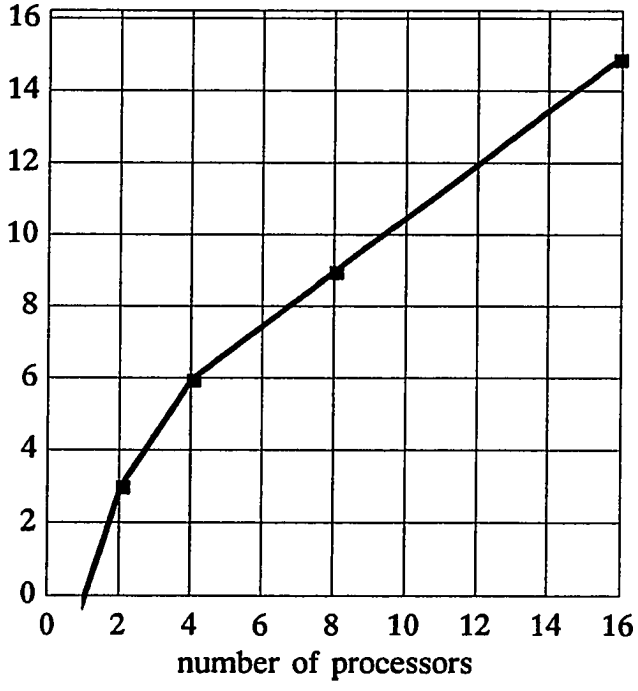

Figure 6.1: Percent increase of search overhead by number of processors.

Test results are not given for more than sixteen processors even though the Sequent Balance 21000 has 30 processors in our configuration. Using more than sixteen processors causes the slope of the speedup curve to go negative, showing that additional processors sometimes hurt performance. This is caused

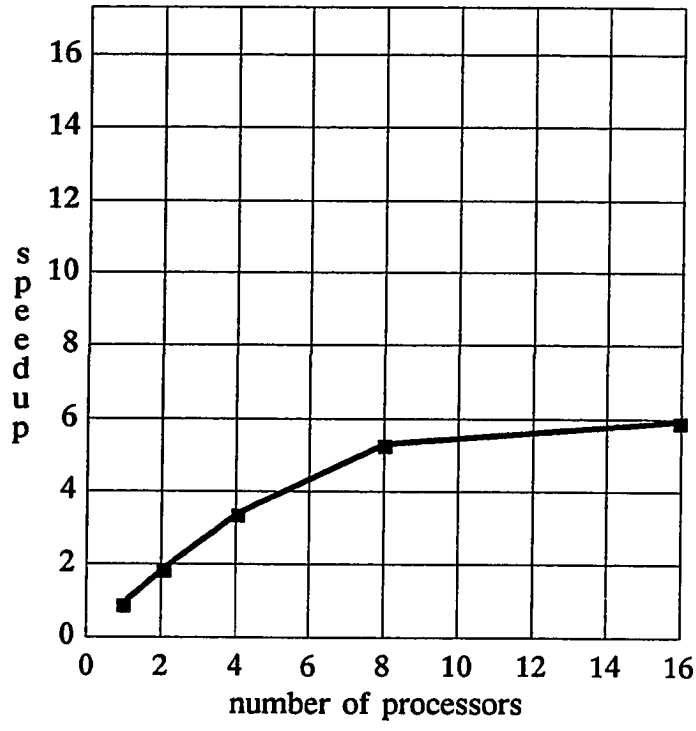

Figure 6.2: EPVS algorithm speedup on alpha-beta game trees.

by the fact that the local memory used by each process created by the chess program is sufficient to exhaust the real memory available on the Sequent when using sixteen or more processors (a maximum of sixteen megabytes in the current hardware configuration). Going beyond this point causes process swapping (drastically increasing the I/O traffic) which causes severe performance degradation.

It is worrisome that some branches are more complex than others, even when the subtree examined occurs near the maximum search depth. Better game-tree programs (chess in particular) must have a sophisticated quiescence search to avoid tactical/positional blunders. This always results in narrow trees that are unsuitable for large numbers of processors since the branching factor is is extremely low in these cases. Tests with the quiescence search disabled provide much more impressive results by eliminating the majority of these

narrow branches, but they only do so by side-stepping the real issue of improved performance without reducing the quality of the search.

Notice also that the variability of the search times increases when using this algorithm. The resplit operations occur at unpredictable times and sometimes occur at correct nodes. On other occasions, a resplit is done at a node that doesn't require a full search which introduces search overhead and the time variability.

6.5. EPVS Performance Summary

The EPVS algorithm offers improved performance over the PVS algorithm in most search problems. It also offers the potential for disastrous search overhead when the number of processors gets large or the branching factor becomes small.

Dividing the tree at one node at a time increases this risk when using large numbers of processors. Chapter 7 describes a much-improved algorithm that circumvents this problem completely.

pos	total nodes 1cpu	2cpu	4cpu	8cpu	16cpu
1	---	---	---	---	---
2	90337	75265	74507	96624	89464
3	29529	29645	30081	30319	30480
4	87411	84079	81469	103207	96552
5	153900	151460	161367	163222	167350
6	13379	13642	14004	14281	14631
7	50675	51450	51040	56138	58008
8	6530	6943	6989	7203	7772
9	75555	78509	81540	87161	85114
10	96369	93839	105983	104126	101158
11	56960	58373	59438	60717	61486
12	197945	207219	207426	216477	228925
13	66899	74318	75645	74664	88700
14	56660	58725	56561	58178	56636
15	53384	54096	54437	56888	56973
16	75483	82603	87025	95520	99733
17	58624	78252	84698	63189	94945
18	120988	131156	137621	138282	147557
19	62071	61086	61161	63067	66150
20	97537	97864	106242	114871	110267
21	141686	153738	157788	146496	177198
22	67291	66383	72557	76496	68029
23	50081	50082	52499	47160	56929
24	73479	73749	73457	75634	83936
tot	1782773	1832476	1893535	1949920	2047963
avg	77512	79673	82328	84779	89042

Table 6.3: EPVS algorithm node counts.

pos	time in seconds 1cpu	2cpu	4cpu	8cpu	16cpu	speedup 2cpu	4cpu	8cpu	16cpu
1	--	--	--	--	--	---	---	---	---
2	970	419	220	198	184	2.32	4.41	4.90	5.27
3	375	198	114	79	77	1.89	3.29	4.75	4.87
4	1950	925	480	382	380	2.11	4.06	5.10	5.13
5	3193	1569	874	586	546	2.04	3.65	5.45	5.85
6	141	76	45	35	35	1.86	3.13	4.03	4.02
7	584	308	171	120	112	1.90	3.42	4.87	5.21
8	75	46	35	35	34	1.63	2.14	2.14	2.21
9	958	525	324	224	161	1.82	2.96	4.28	5.95
10	1174	576	346	218	187	2.04	3.39	5.39	6.27
11	697	371	206	131	118	1.88	3.38	5.32	5.91
12	3763	2076	1069	589	438	1.81	3.52	6.39	8.58
13	652	373	205	134	143	1.75	3.18	4.87	4.56
14	1068	562	268	190	192	1.90	3.99	5.62	5.56
15	939	482	261	175	154	1.95	3.60	5.37	6.10
16	1283	729	461	324	282	1.76	2.78	3.96	4.55
17	737	509	294	139	119	1.45	2.51	5.30	6.19
18	1977	1095	625	367	301	1.81	3.16	5.39	6.57
19	741	372	207	158	160	1.99	3.58	4.69	4.63
20	2009	1143	596	361	334	1.76	3.37	5.57	6.02
21	2706	1478	780	413	412	1.83	3.47	6.55	6.57
22	840	424	249	175	136	1.98	3.37	4.80	6.18
23	600	315	178	96	87	1.90	3.37	6.25	6.90
24	758	392	216	137	131	1.93	3.51	5.53	5.79
tot	28190	14963	8224	5266	4723	1.88	3.43	5.35	5.97
mse	0	3	13	47	61	.00	.01	.07	.16

Table 6.4: EPVS algorithm speedup.

CHAPTER 7

DYNAMIC TREE SPLITTING (DTS)

7.1. Introduction

Chapter 5 outlined the difficulties of parallel tree splitting algorithms that are too static when managing parallel tasks. When watching ants devour crumbs at a picnic, they do not gang up on one crumb and then stop and wait when the crumb becomes too small for the entire group; rather, the ants that get crowded out move to the next crumb immediately without waiting on the rest of the group.

This "ants at a picnic" idea provides the basis for the correct approach to solving the parallel search problem. Processors cannot wait for each other to finish work, but rather they must immediately proceed to find another branch and begin searching it. This autonomy is a nice concept, but it turns out to be surprisingly difficult when implementing it on a parallel machine. The interactions that naturally occur make coordination both necessary and at the same time complex.

The correct approach to this problem is obvious but the solution is both difficult to design and nearly impossible to debug. The processors search the tree differently each time they try the same test, producing unpredictable behavior that introduces a significant amount of non-determinism.

The following sections describe the basic operation of the DTS algorithm, the various requests used to coordinate the parallel search, the data structures required to implement the DTS algorithm and how they are used, and then

presents the DTS algorithm in detail along with actual performance data produced by it for the same types of trees considered in chapters 5 and 6.

The mathematical analysis presented in chapter 4 is somewhat more difficult to apply to trees examined by the DTS algorithm. Since the tree is sometimes split before a search bound is established, equation 7 must be applied to these subtrees individually and the resulting node counts must then be added in to the total. In the optimal case, equation 7 can be applied to the root node, but in practice non-optimal splits are sometimes done to keep processors busy. The cost of the non-optimal splits is less than the performance advantage gained by keeping the processors busy.

7.2. An Overview of the DTS Algorithm

Cray Blitz creates the parallel tasks when the operator requests parallel processing. All processors (except for processor one, the original processor) post help requests and wait for work.

When starting the tree search, the single processor quickly detects the help requests from the idle processors, but takes no action until it searches down to the search depth requested (just like the PVS and EPVS algorithms). At this point, the processor executes a split operation which immediately attaches the idle processors to some node and starts the parallel search.

As the search progresses, processors run out of work to do at an active split node and then generate a help request. A processor accepting the help request executes a split operation and starts a parallel search at some node. As other processors run out of work, they also join at this split node and help with the parallel search.

From this point on, idle processors (those "crowded"out) move to the next reasonable split node (find the next available "crumb") and search it in parallel.

This continues until all processors run out of work; then, processor one returns the search value computed by the group to the main program.

The only special–purpose code present is in the routine that selects a split node since it does not split type one nodes unless the search depth is deep enough to avoid excess overhead (recall the $N_{\alpha\beta}$ discussion from chapter 4.) The search treats this as a special case, otherwise the many help requests demand a node for splitting from some processor (only one is busy here).

7.3. The Help Request

When a processor needs work, it first checks the shared search data structure and finds all active split points (ignoring the node just completed since it contains no more work.) If any split points exist, the processor identifies the split point with the most work remaining and immediately joins the parallel search at that node. This eliminates any additional overhead whatever and also keeps the processors together in the same area of the tree maximizing the usefulness of the transposition table.

If the processor finds no split point, it uses the help request to ask one of the other processors for work. The processor queried by this request examines the work left at each of its nodes and determines if there is an appropriate place to split the work with another processor. If there is, it executes a split operation (discussed below) to let the idle processor help out.

If this processor finds no reasonable node to split, it passes the help request to the next busy processor. This requests passes from processor to processor unless one finds a good node to split (defined as a node almost certainly needing all its successors examined.) This first cycle through the busy processors finds the first processor with a node requiring examination of all its branches.

If no processor finds such a node, the help request cycles back to the first processor, but this time it relaxes the certainty requirement so that a processor chooses a node less certain to need complete examination. This algorithm finds the first good node to split without polling every processor for a response unless necessary; usually the first processor polled finds a node to split.

Since searching a node not needing examination leads to search overhead, this algorithm invests a small amount of computation overhead to minimize the size of the tree. It cycles through the pool of busy processors an arbitrary number of times when trying to select the best possible new split point, each time lowering the full–width confidence requirement by some amount. The help request handler introduces virtually no overhead when processing these confidence interval searches, so that the primary issue is locating the best possible node choice for parallel splitting.

7.4. The Split Operation

Splitting should take place only at type 1 and type 3 nodes within the tree. When asked (via a help request) to split a node and share it with another processor, split uses a function 'typenode' to classify each node in the current tree being examined. Typenode uses the definitions from chapter four and classifies each node from ply one through the current ply as 1, 2, or 3. After a type 1 node has a value backed up to it (which signals that it is then safe to split the node since the bound is properly established), typenode changes that node type to 3.

Split then locates the type 3 node at the lowest depth and assigns processors to it (assuming it has work left to be done). If there are more idle processors than there are nodes remaining at this split point, only enough processors are assigned to examine each of the remaining nodes. Split then

once again finds the type 3 node at the lowest depth (again, with work remaining) and repeats the previous procedure until either all idle processors are busy or all type 3 nodes have been split.

If split finds that processors are still idle after the above procedure, it then forwards the help request to the next processor so that it can split and assign idle processors to help it. This represents a "worst case" condition and is the first symptom of the "feeding frenzy" that occurs as more and more processors become idle and request work.

It should be noted here that split never selects a type 2 node for splitting (except as described below) since such a node only requires that one successor be examined. Likewise, split never selects a type 1 node for splitting since the corresponding lower or upper search bound is unknown, possibly causing extra nodes (search overhead) to be examined.

The only time a type 2 node is selected for examination is when all processors have tried to split based on the above criteria (phase 1 split operation). If idle processors still exist after the above procedure is tried on all processors, a phase 2 split is attempted. Here the processors split the deepest type 1 node available. Share will inform these processors as soon as the bound becomes known, meanwhile we risk search overhead rather than letting a processor sit idle, since it would be doing nothing useful anyway.

If the phase 2 split operation still does not exhaust the idle processor list, then a phase 3 split is tried. Here, even type 2 nodes may be split, but only if more than one successor has already been examined. The reasoning here is that if more than one successor has already been examined, then possibly this node is a type 3 node due to a breakdown in move ordering. It might be that the best move from the type 2 node has not been tried, and will cause a cutoff as

soon as it is found. If so, then this will probably introduce additional search overhead, but at a reduced cost since the processor searching these extra nodes would be idle anyway. It should be noted that it is not reasonable to split "just anywhere" rather than "nowhere" since once a split is done, additional overhead is incurred due to the merge operation, share operations, and the synchronization that occurs when processors update the shared memory block.

During the first 99.9% of a search iteration, this split overhead is extremely low since phase 1 and phase 2 splits easily find work to do. However, as the search nears the end, there are more and more idle processors demanding work and there is less and less work to give them. Eventually, these processors behave like a school of fish on a "feeding frenzy" and disturb the search enough that the split overhead increases significantly. When only one or two branches are left, this traffic can become nearly unbearable and has on occasion resulted in tremendous increases in the time to complete a search even though the best move was found in record time.

Whenever processors split work at any node, they share a block of global memory that contains the entire tree search data structure up to that point in the tree.

The split operation (illustrated in figure 7.1) copies the tree search data structure (in the processor's local memory) into global memory so the idle processor can access it. This shared memory area contains approximately 10,000 words of memory (the same size as the local memory used by each processor).

Since split operations occur frequently, the split code is very selective in the data that it copies. To this end, the algorithm recognizes three distinct areas of the shared common. The algorithm copies area one completely; it copies each

array of area two from element one to the depth of the split node; it copies none of area three since these contain purely local data passed between procedures when processing one node. This usually copies only a few hundred words.

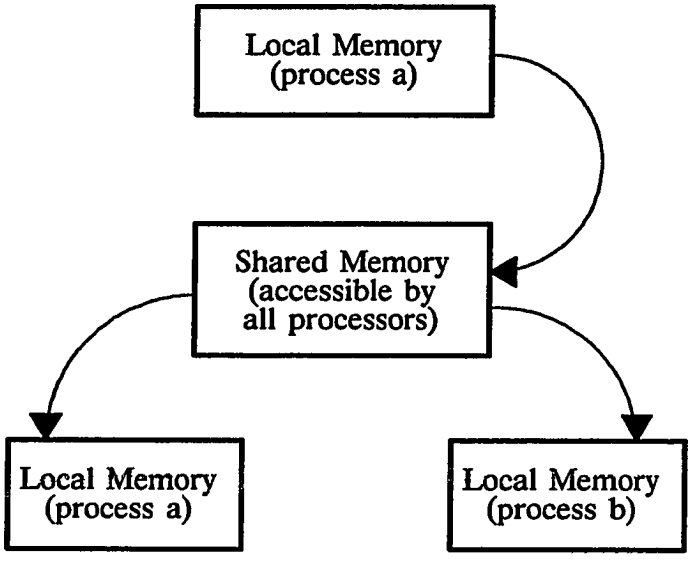

Figure 7.1: Memory copy operation performed as a result of the split request so that processors can share their local memory areas.

Except for branches selected from the split node, the parallel code behaves exactly like the sequential code. At the split node, the processor selecting a branch to follow first locks the shared data structure with a semaphore operation, selects the next branch to follow and removes it from the list of available branches to search, and then unlocks the shared data. This avoids multiple processors following the same branch and duplicating the same search tree.

The data structures that implement this split operation are described later. However, since absolutely no distinction is made among the processors while they execute the parallel search, the data structures become necessarily complex. It is possible that processor 1 splits a node with processors 2 and 3 and then runs out of work at the split point. It will then enter the idle loop and

eventually, either processor 2 or 3 will back up through the split point, even though they did not originate the data above that level. Avoiding the so-called master/slave relationship between processors is necessary to avoid those cases where the master is busy waiting on one of its slaves to complete a search. This synchronization overhead is highly undesirable, but is also quite complicated to eliminate.

7.5. The Unsplit Operation

The next problem arises when all but one processor run out of work (at the same split node). Each idle processor sends a help request as before, but there is a memory usage problem associated with the remaining busy processor. Notice that there is a shared memory block in use, even though only one processor is now using it.

When the next-to-last processor at a split node becomes idle, it sends an unsplit request to the remaining processor. The unsplit request conditionally copies the shared memory back over the busy processors local memory if appropriate. Specifically, the processor compares its value at the split node with the value in the shared common (produced by the other processor or processors that split the work up). If the shared common represents a better result than what is in the local common, its contents replace the local common. Figure 7.2 illustrates this.

The last step is marking the shared common block as "not-in-use" so that another split operation can use it. Without this unsplit operation, the shared memory requirement of this algorithm is astronomical since processors may execute the split operation thousands of times in a single search (depending on the machine used.)

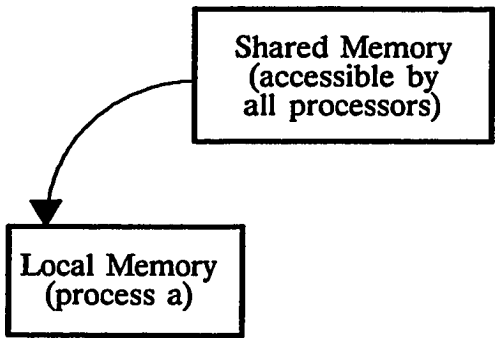

Figure 7.2: Memory operation done when an unsplit request occurs.

The unsplit request effectively eliminates any sign of parallel processing at the node in question, including shared memory usage. After processing the unsplit request, the processor still searching a branch at that node behaves as though the split was never done.

7.6. The Merge Operation

When processors split a node into parallel tasks, they copy the shared data (which originally came from a sequentially processed branch) to their own local memory. After completing the examination of whatever branches they search, the processors finish with n different results from searching n different subtrees.

To reconcile this, the first processor finishing its work copies its data back over the shared data unconditionally. As each additional processor finishes, they merge their data into the shared memory as needed. This operation compares the results obtained by that processor's search with the best results returned so far. If the processor's results are better, these results replace the ones in the shared memory area.

The criteria for the merge operation is identical to those for the unsplit requests. The best result returned from the parallel search replaces the global memory data and eventually get merged back into the branch they originated from.

7.7. Processor Inter-Communication

As the parallel search progresses, processors encounter conditions that affect other processors. This requires that certain types of messages (or requests) pass between processors as rapidly as possible without the need for handshaking during the communication.

An example of this occurs when move ordering breaks down at a node within the tree (generally at a type 2 node) and successors of the resulting position refute the branch when they should not. This requires searching another branch from the type 2 node with the resulting search overhead. When this happens, each of the processors starts searching at the split node and one quickly finds a score which causes a cutoff. This processor quickly backs up to the split point, but the other processors continue to search since they have not detected the condition causing the cutoff. Rather than waiting until they also find a move causing a cutoff, the processor finding the cutoff move immediately informs all processors working at this split node that they should stop searching and find work to do at some other node. Even though they searched extra nodes, they stop as soon as one detects the cutoff and avoid even more search overhead.

Another example occurs when processors split the tree at the root node. If one of them then encounters a move (branch) better than the current best move, a beta cutoff occurs. This processor informs the other processors of this so that they split the subtrees of this new best branch and complete the analysis of it more quickly, possibly avoiding running out of time without having the analysis completed. Recall that the principle variation is very important as it guides the next search iteration by providing good move ordering to minimize the search tree size and also provides the search with an estimated move for the opponent.

The search then uses this estimated move to continue searching while waiting on the opponent.

Another reason for stopping processors when a new best move is found is that the new best move raises the lower search bound, resulting in a tree with fewer nodes than one searched without this bound. Letting processors continue to search the tree without knowledge of this better lower bound increases search overhead (nodes examined) and adversely affects overall performance by allowing a processor to search nodes that are not necessary. An interesting point here is that with small numbers of processors this does not happen frequently. However, when large numbers of processors are available, split takes more risk in choosing split nodes, causing more communication.

7.8. The Share Operation

When processors split the tree at so-called split points, it sometimes happens (particularly when searching type-1 nodes) that a new best score is backed up to the level of a split point. This processor now has a better search bound for searching the remainder of its subtrees more efficiently. Unfortunately, the other processors know nothing of this new value and search branches that are probably unnecessary.

To prevent this, whenever backing up a new best value to a level identified as a split point, the processor sends a "share" request to all processors sharing this split point. These processors then compare the shared scores with their own and copy those that are better. A processor must check all split points it is using when such a request comes in as it is possible for several share requests to be generated simultaneously from each of these split points.

An added bonus of this share operation is that after doing it, it is not necessary to do anything when an unsplit is done, since the score and related

information is already current in the last process at a split point. For timing reasons, the unsplit checks are still made as a share request can come in as a processor is finishing the last bit of work at a split point. When this happens, the share operation might not be caught, but the unsplit operation correctly handles the new best value anyway.

7.9. Data Structures

The data structures used to maintain order and integrity during the parallel tree search are illustrated in figures 7.3 and 7.4. The major elements are explained below to show how they are used during the course of the parallel search.

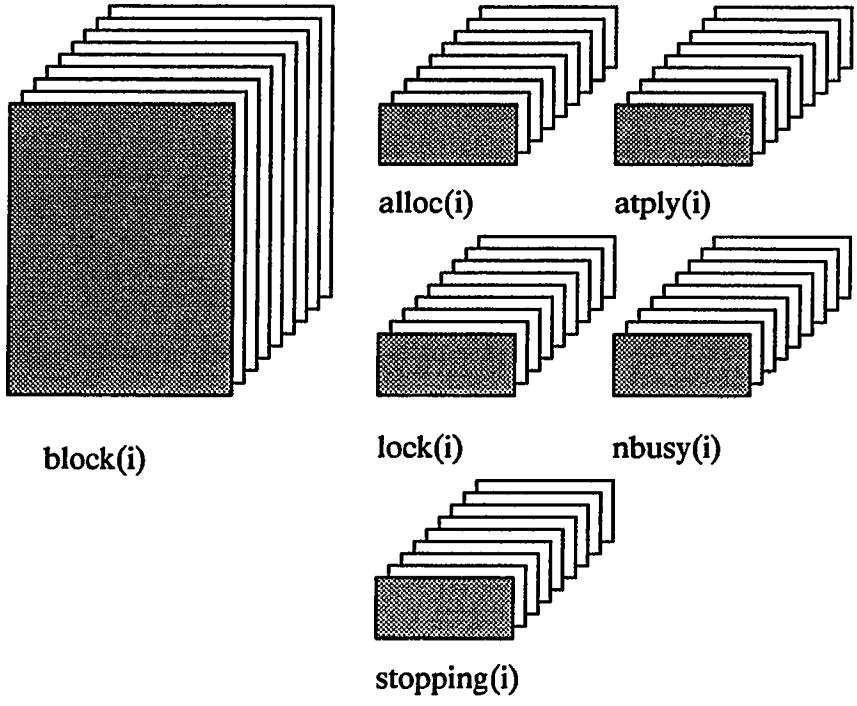

Figure 7.3: Data structures shared at a split ply and indexed by a block number.

BLOCK(i) is the shared memory area used to split the tree among several processors. It is essentially a duplicate of the local common each separate processor used to perform a tree search.

If ALLOC(i) is non-zero, then BLOCK(i) is in use by some processor(s) and is not free to be used at the next split point. If ALLOC(i) is zero, then the corresponding BLOCK(i) is unused and available. To handle narrow trees, it is necessary that i be at least the maximum number of processors plus one. Otherwise, a split operation might have to wait for an unsplit to free up a BLOCK(i), wasting time.

LOCK(i) is the semaphore array used to protect BLOCK(i) when it is being accessed by one of several processors. Before BLOCK(i) is used in any way (including reading and writing), the corresponding LOCK(i) must be set by a semaphore test and set instruction. Even reading is not permissible without protection since another processor might change part of the data while it is being read, resulting in part of the copied data being old and part being new.

ATPLY(i) indicates the depth at which the corresponding BLOCK(i) was produced by a split operation. This is used by debugging tools to dynamically display what is going on during the parallel search without interfering with the search in progress.

NBUSY(i) is a counter used to determine how many processors are busy at the corresponding BLOCK(i) split point. As processors are assigned to BLOCK(i), NBUSY(i) is incremented. As processors finish at a split point and call merge to possibly copy their results back to BLOCK(i), merge decrements NBUSY(i). When merge notices that NBUSY(i) is exactly one (after it decrements it), it then sends an unsplit request to the last busy processor working on BLOCK(i) so that BLOCK(i) can be freed up for re-use.

STOPPING(i) is used to indicate that a split point is bad, but has not been cleaned up yet. No other processors should join with this split point even though it appears that plenty of work is still available here. Essentially, this node is not a type 3 node, even though it appeared to be when the split was originally done. When some processor finds a refutation that makes the remainder of the work left at this node unnecessary, it then sends a stop requests to all processors busy at this split point, and uses STOPPING(i) to make sure that no additional processors join in and miss the stop request which was sent before they joined in.

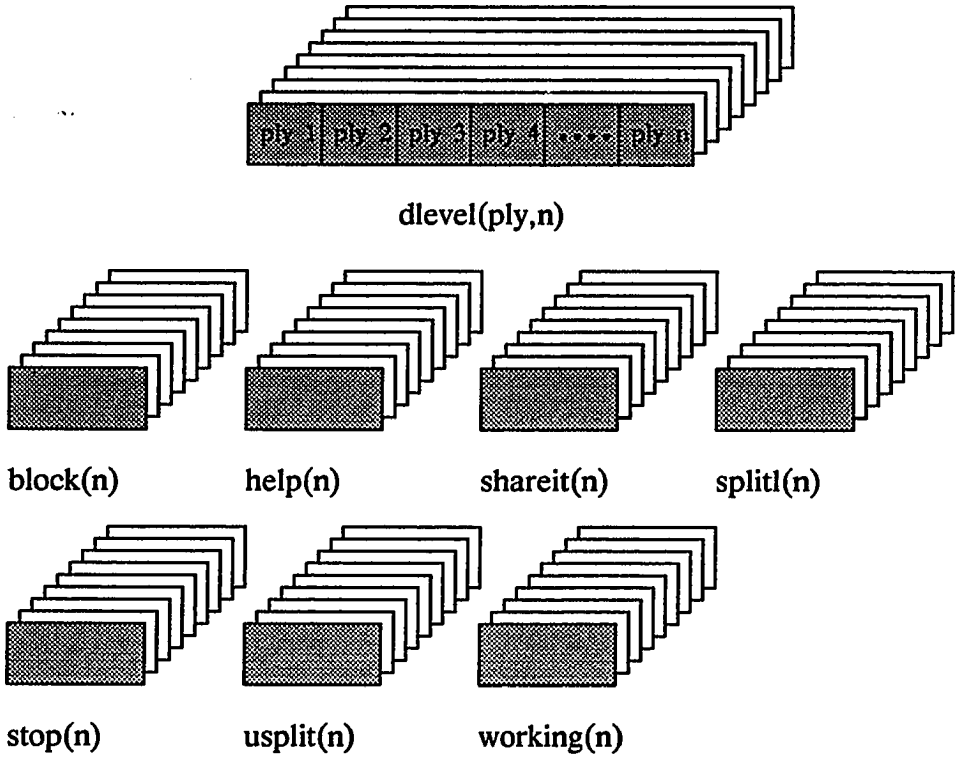

Figure 7.4: data structures for each processor indexed by processor id.

DLEVEL(ply,n), where n is a unique processor id, is the principle data structure used by DTS to keep up with the split operations done. If processor n

has split at ply 5 using block 3, then DLEVEL(5,n) is 3. This identifies each ply at which processor n is splitting, and also identifies the associated shared memory block that contains the tree data for that split point.

BLOCK(n) indicates the deepest split point for processor n. When a merge is done, this pointer identifies the block without having to search through DLEVEL(ply,n) to find the block number. Also, when a processor is idle and waiting on work, BLOCK(n) is zero (note that if BLOCK(n) is zero, it does not imply that the processor is idle, because it could be in unsplit and fixing to change BLOCK(n) to a new block number). Split first finds a split point, allocates a block and copies its local tree search data to this block and then sets BLOCK(n) to point to the block. At this point, the idle processor exits the wait loop (where it is waiting for BLOCK(n) to become non-zero) and starts the parallel search.

HELP(n) is the method used to inform busy processors that an idle processor is available. When processor n notices that HELP(n) is non-zero, it calls split to attempt parallel division if any reasonable split node can be found. A value of 1 indicates a phase 1 split attempt should be tried; if the help requests cycles through all processors, it is incremented to 2. A value of 2 indicates that a phase 2 split attempt should be tried, which incurs more search overhead than phase 1 attempts. If this help request cycles through all processors, it is incremented to 3 indicating that a phase 3 split attempt should be tried. It should be noted that only one HELP(n) will be non-zero at any instant in time, since no broadcasting is currently done, but rather sequential polling is used.

SHAREIT(n) indicates that some other processor is sharing a split point with processor n and has posted a new best value at that depth. Processor n

must copy this new best value if it is better than what it already has in order to minimize the tree search overhead. Since a processor may have several active split nodes, it must check them all when the share request is made because several simultaneous share requests can arrive with no indication of which split point to which they refer.

SPLITL(n) is used by the search routine as a quick check for if the current ply is a split point. If SPLITL(n) = current ply, then special action is required when choosing the next branch. Also, when completing such a ply, backing up is not possible, rather a merge must be done with only the last processor at a split point being allowed to back up beyond this ply.

STOP(n) is used to stop processors when their work is found to be unnecessary. If a cutoff occurs that would makes complete examination of a split point unnecessary, STOP(n) for each processor working at this split point is set to the depth of the split point. When this is non–zero, each processor must immediately stop, clean up the data structures and enter the idle loop after posting a help request. The last processor busy at the split point returns through it and continues the search, possibly splitting with the idle processors that just quit, but at some other split point.

USPLIT(n) is used to inform a processor that it is the last one working at a split point. It must call unsplit to clean up and release any BLOCK(i) shared data area with NBUSY(i) = 1 so that these blocks can be used again.

WORKING(n) is used by the debugging display and is always set to 1 if a processor is busy, and is set to 0 when a processor is idle. This is used to determine if a processor is really busy, even though BLOCK(i) is zero.

7.10. Task Granularity

Task granularity is a major consideration when developing a high-performance algorithm such as this. Holding search overhead to a minimum requires a code that can divide even the smallest trees into parallel tasks. The analysis in chapter 4 proved that parallel division must be done primarily at type 3 nodes (since there are only n type 1 nodes in a depth n search). In a parallel search, choosing the best node for parallel analysis might select a node deep in the tree, generating very small subtrees. If the time required for starting or coordinating parallel tasks is high in relation to the time required by the parallel search, then selecting such nodes causes unacceptable overhead with poor performance.

The current code wastes less time when an idle processor joins in with a busy processor than it uses when expanding a single node (depending on the machine being used because much of the overhead is vectorizable on Cray computers where the tree search code is not). A processor is rarely idle for more than four or five node cycles, even on the Sequent, except near the end of the search when little work is available. This means that this code works for arbitrarily small trees (or for nodes arbitrarily deep in the tree). This small time requirement allows parallel division in the quiescence search where small subtrees and low branching factor nodes regularly occur.

This completely removes search depth and branching factor considerations from the decision when choosing a node for parallel analysis. This algorithm selects the best candidate for parallel processing without weighing any other features (such as the probable size of the subtrees, the probable number of subtrees, or any other related aspect). If this were not possible, then the code would find circumstances where the only work available is too fine-grained.

7.11. Synchronization Overhead

Synchronization overhead occurs at many points during the parallel search, even with the current algorithm. Whenever updating a shared data structure, synchronization prevents problems such as interleaved updating or traversing a broken linked list.

To reduce this synchronization overhead to an absolute minimum, these timing windows are as small as possible. This minimizes the period of time between loading a shared value and storing an updated value back possible. Often this requires new code (that is somewhat less readable) to improve the sequential update processing.

After minimizing these windows with optimized coding practices, semaphores prevent the previously mentioned problems. However, since this algorithm updates many different data structures during the course of the search, it uses different semaphores for each different data structure. This allows updating different shared data structures in parallel while avoiding updating the same data structure in parallel.

This idea is particularly important when splitting and merging operations occur on different nodes. Even though different processors update the same shared data structure, they always access different parts of the shared tree data. This lets different processors find split points in parallel except when they try to join with the same processor where sequential processing occurs due to the semaphore locking mechanism.

7.12. Processor Clustering

The primary reason for processors first trying to work at an existing split node before creating a new one is that this keeps processors together in the same area of the tree. This both maximizes the usefulness of the transposition

table and also eliminates the small overhead required in creating a new split node.

As a processor "peels off" from the cluster of busy processors (because there are no additional branches), it creates a new split node. As other processors peel off later, they immediately join with the first at the new split node.

The normal parallel tree has two split nodes active at any time, one that has no unsearched branches (except those branches in progress) and one that is newly created. For narrow trees there may be many split nodes with no unsearched branches (again, except for those in progress), but there is only one split node with work available because idle processors will automatically join at such a split node without ever sending a help request to another processor.

7.13. DTS Performance on Uniform Game Trees

The DTS design addresses uniform and non-uniform minimax game trees in addition to alpha-beta trees. DTS searches uniform game trees exceptionally well and produces near-optimal speedups when compared to the speed of the sequential algorithm.

We modified Cray Blitz so that the move generator produces a fixed number of moves when called. We also disabled the quiescence analysis and the alpha-beta algorithm so that the program searches perfectly uniform trees.

Table 7.1 shows the results obtained with the DTS algorithm searching uniform game trees. These test cases vary the branching factor and also vary the number of processors searching each tree to show that this algorithm works equally well for narrow or wide trees. Compare this with table 5.1 and table 6.1 to see the advantage DTS has over either PVS or EPVS. For these tests, DTS exhibits no leveling off in the performance curve, showing that it produces a

w	d	time in seconds 1	2	4	8	16	speedup 2	4	8	16
2	16	351	176	89	46	24	1.99	3.94	7.63	14.63
4	9	294	147	75	38	19	2.00	3.92	7.74	15.47
8	6	503	252	128	64	33	2.00	3.93	7.86	15.24
16	5	1869	935	474	239	119	2.00	3.94	7.82	15.71
32	4	3838	1919	965	484	244	2.00	3.98	7.93	15.73
64	3	3462	1731	870	441	220	2.00	3.98	7.85	15.74
tot		10317	5160	2601	1312	659				
avg		1719	860	434	219	110	2.00	3.95	7.81	15.42

Table 7.1: DTS algorithm speedup when searching uniform game trees.

linear speedup as the number of processors is increased (meaning that a straight line fits the speedup plot, even though it might not be optimal). This data is graphically summarized in figure 7.5.

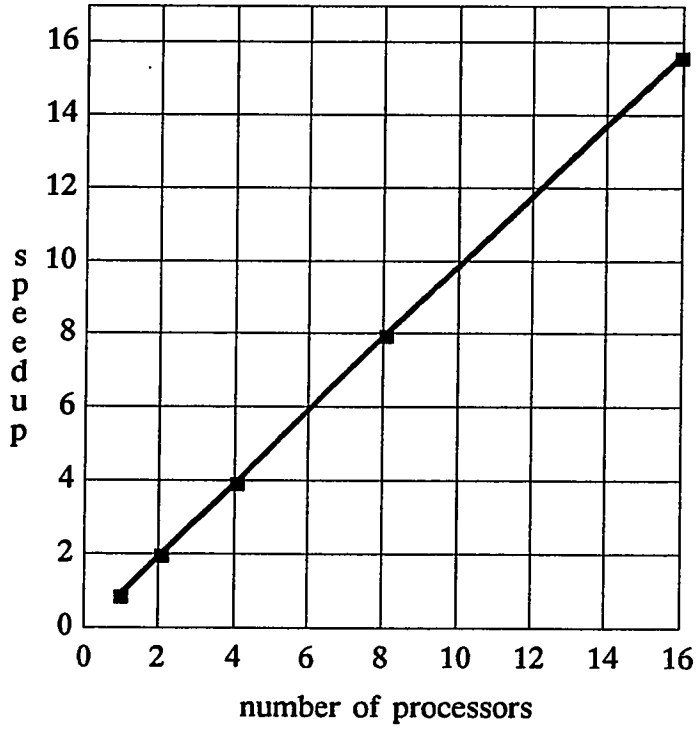

Figure 7.5: DTS algorithm speedup when searching uniform game trees.

The columns labeled "w" and "d" reflect the branching factor (w) and the search depth (d) used for that particular test. These parameters generate trees

requiring approximately twenty minutes for one processor (except for w=8 as the next depth generates a tree requiring over thirty minutes).

DTS provides almost linear speedups for any reasonable tree as table 7.1 shows. Note that the lowest branching factor trees (w=2) generated the poorest speedup because there is little to do at each node (only two branches). Note also that even the worse case shows no tapering off or asymptotic behavior; however, it does show the loss of performance caused by the split/unsplit/merge overhead (even though this overhead represents a fraction of the work spent in analyzing one node).

Another consideration when testing an algorithm such as this is that using large numbers of processors causes the search to slip inside the timing accuracy of the hardware. For example, in the w=8 test, the code generates the sixteen processor result in less than twenty seconds. A fraction of a second is very significant when it potentially represents five percent of the total search time. This greatly affects the speedup curve. Increasing the complexity of the problem only slows down the single processor tests, making them very difficult to run since they require long periods of dedicated test time.

7.14. DTS Performance on Non-Uniform Game Trees

We again modified Cray Blitz so that the search generates non-uniform trees and left the alpha-beta algorithm disabled. The move generator produces non-uniform trees where each branch from the root position produces a predetermined number of successors. Each branch from these successors also produces a predetermined number of successors. This provides a constant tree that does not change when the parallel search examines branches in different orders.

A second modification to the search varies the depth of search according to the position of the branch within the search tree. This also results in trees with a constant shape regardless of the order in which the search examines the various branches.

With these modifications, the program executes searches using the DTS algorithm and produces the results in table 7.2. The search parameters cause the search to vary the tree and produce six different test cases. For the test set, the average branching factor w varies from 2 to 64 while the depth remains constant. This data is graphically presented in figure 7.6.

		time in seconds					speedup			
w	d	1	2	4	8	16	2	4	8	16
2	16	351	176	90	47	25	1.99	3.90	7.47	14.04
4	9	294	147	76	39	19	2.00	3.87	7.54	15.47
8	6	503	252	128	64	33	2.00	3.93	7.86	15.24
16	5	1869	935	475	240	120	2.00	3.93	7.79	15.58
32	4	3838	1919	965	484	244	2.00	3.98	7.93	15.73
64	3	3462	1731	870	441	220	2.00	3.98	7.85	15.74
tot		10317	5160	2604	1315	661				
avg		1719	860	434	219	110	2.00	3.93	7.74	15.30

Table 7.2: DTS algorithm speedup when searching non–uniform minimax game trees with varying depths (d) and branching width (w).

Notice that once again, the DTS algorithm has no problem in producing results nearly identical to those from the uniform search tests. We expected this since it performed so well on the uniform tree test with w=2. The only difficulty the search faces in these tests is that after the basic full width search, many branches have exactly one successor. This produces very narrow subtrees and increases the search overhead when a help request cycles through all processors more than once before finding a good split node, since in these cases some nodes (and entire subtrees) present no opportunity for parallelism.

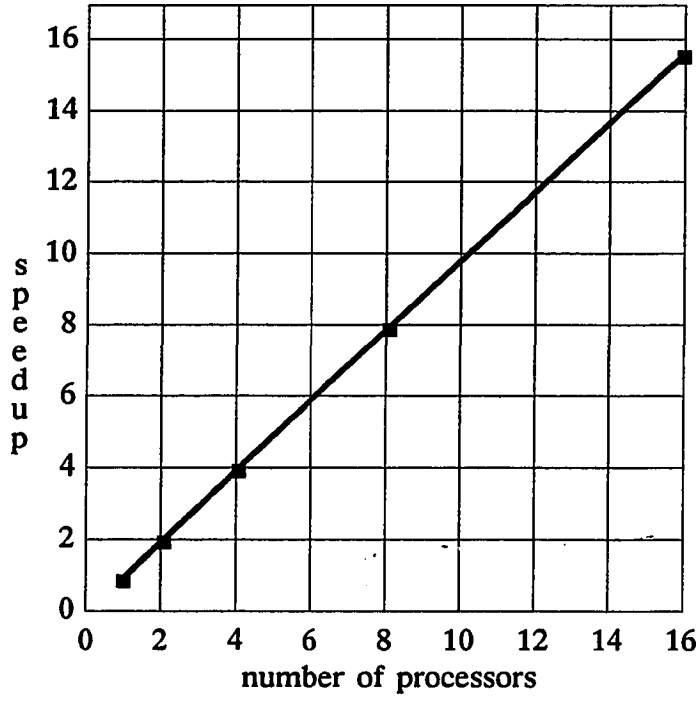

Figure 7.6: DTS algorithm speedup when searching non-uniform minimax game trees

As table 7.2 shows, these trees cause no problems for the DTS algorithm. The results are nearly identical with the results using uniform trees. The search copes with the narrow branching factor and variable length branches with little difficulty. Some very slight increase in the overhead results from the help requests that cycle through several processors until reaching one with a node that can be split.

The narrow trees are especially significant since the algorithm handles them with the same efficiency it displays on wide trees. This implies that the endgame and tactical tree problems discussed earlier disappear with this algorithm.

7.15. DTS Performance on Non-Uniform Alpha-Beta Game Trees

This section tests the DTS algorithm on the alpha-beta tree problem. The test results clearly show that the DTS algorithm outclasses either PVS or EPVS

for all problem types. For small numbers of processors, the advantage is quite narrow, but increases sharply as the number of processors increases.

Tables 7.3 and 7.4 provide the same information for the DTS algorithm as that presented in chapters 5 and 6 for the PVS and EPVS algorithms. Before comparing the single-processor times between PVS, EPVS and DTS, note that the chess program used with PVS and EPVS were identical, but that improvements have been made to the current chess program, changing the single processor node counts, sometimes significantly. However, the speedup data can be directly compared between the three algorithms.

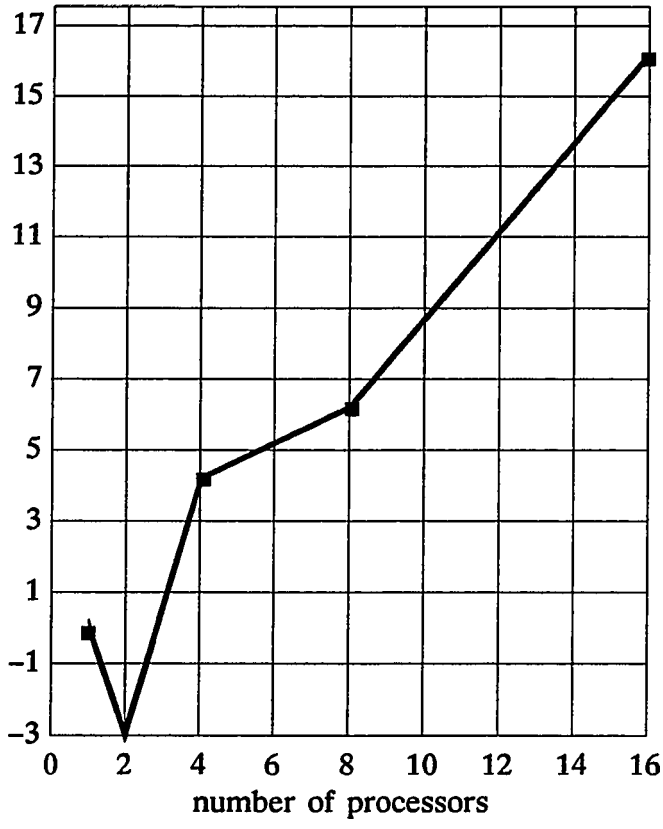

Figure 7.7: DTS algorithm percent search overhead on alpha-beta game trees

In analyzing this data, figure 7.7 graphically presents the search overhead introduced as the number of processors increases. Notice that this number is still quite low, suggesting a minimal cost increase when adding processors. The

two processor data is particularly interesting in that the search actually examines fewer nodes with two processors than it does when only using one. This is an anomaly caused by poor move ordering in the sequential search that the parallel search happens to improve since it alters the order in which moves are considered.

Figure 7.8 presents the speedup curve for the DTS algorithm. It is interesting to note that not only is it significantly better than either PVS or EPVS, but that for up to eight processors, the speedup curve is almost a straight line, the first time this behavior has been seen.

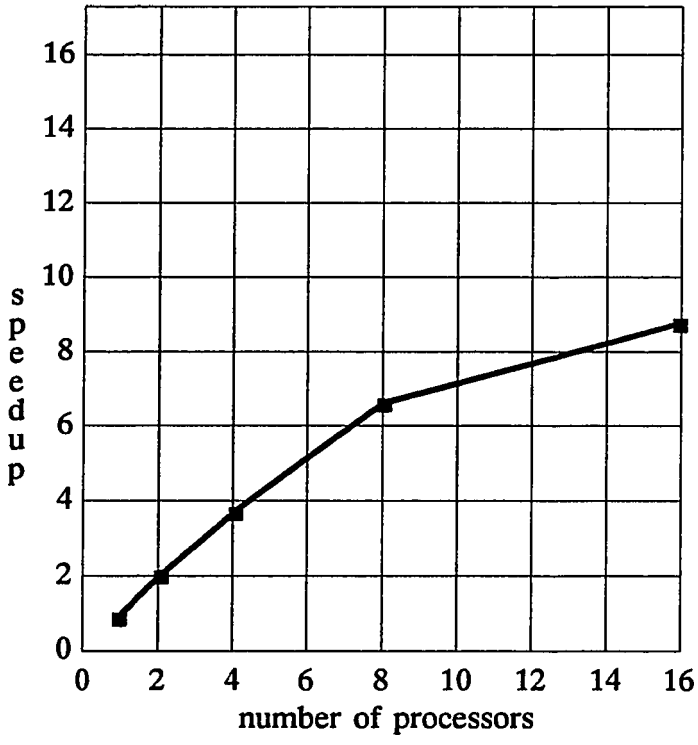

Figure 7.8: DTS algorithm speedup on alpha–beta game trees

Looking at the raw data is much more revealing in that this algorithm actually provides significant speedups for more than eight processors. There are several examples where a speedup of more than ten is obtained with sixteen processors, something that neither PVS or EPVS came close to producing.

The mean standard error reveals another interesting characteristic of this algorithm. The variability of the search time is directly caused by the unpredictability of where splits are done and the unrepeatability of doing them the same way more than once. The PVS algorithm provides the most repeatable timing data while DTS shows significant variance.

7.16. DTS Performance Summary

The DTS algorithm offers significantly improved performance over the PVS algorithm in all search problems. It also avoids the potential for search "thrashing" experienced by EPVS since DTS does not require that all processors work at the same node, nor does it require that processors stop searching tree branches just to allow idle processors to join in at some point in the tree.

The only multiprocessing problem exhibited with the DTS algorithm occurs at the end of the search examining the first branch, and at the end of the complete search for some fixed depth.

Since it is undesirable to examine a type 1 node until all of its successors are completely examined, there are periods of time when the remaining work is scarce, since the work division must be done *below* the type 1 node. As this work is completed, idle processors increase their generation of help requests, causing the busy processor(s) to continually try to split, only to find that there is little or no work to split. In these peak periods, properly called a "feeding frenzy," the overhead climbs significantly, sometimes beyond the point where a processor could complete the work faster if left completely alone.

This same condition arises at the end of a complete search for the same reason. Again, the idle processors demand work and the busy processors have no luck in giving it to them. Again, the overhead increases as busy processors are continually interrupted for nothing.

Several ideas to reduce this overhead have been suggested. When the number of idle processors passes some lower limit, the code could assume that the search is winding down and is susceptible to the "feeding frenzy" condition. Under such circumstances, it might be more efficient to defer help requests until a branch is encountered with enough work to warrant the overhead caused by trying to split the tree at some point. This would occur when all nodes below a type 1 node are completed, or when the next search starts after the current one is finished.

Another suggestion would be to circulate the help request when the number of idle processors is small, but to broadcast it to all processors when the number of idle processors is large. This would allow the split to be done on the absolutely best possible node, rather than on the best possible node of a random processor, which might not provide much work for idle processors.

After all is said and done, however, the alpha–beta tree search problem is difficult for parallel analysis, and the above ideas will not come anywhere close to providing near–optimal speedups for most problems.

Chapter 8 details other problems with this type of tree search in general, and addresses possible future work to solve them.

pos	total nodes 1cpu	2cpu	4cpu	8cpu	16cpu
1	---	---	---	---	---
2	96650	100694	96995	102753	121404
3	29500	29453	29562	29947	30059
4	87595	82822	87813	98382	91589
5	153571	151051	158544	154068	179707
6	13947	14034	13631	14099	13945
7	50709	51370	56909	57443	64321
8	6489	6763	8099	10203	7107
9	59488	56653	66493	75426	87156
10	97245	92149	97364	92733	100098
11	57070	58584	58835	61792	66122
12	198013	206831	215585	194690	188861
13	68961	70462	71949	75772	75002
14	59163	55089	52536	54102	54941
15	55082	55347	55353	55504	55202
16	133471	92350	98389	103750	122738
17	57479	64563	90445	87234	125545
18	121129	115684	133067	145535	169720
19	62117	64857	78068	90097	122089
20	122176	104371	107507	114070	121396
21	127709	132681	143856	149375	145584
22	69994	66914	74262	65411	74309
23	50142	52099	52419	52192	52648
24	73590	73659	73908	74631	76500
tot	1851240	1800480	1921589	1967209	2146043
avg	80489	78282	83547	85531	93306

Table 7.3 DTS algorithm node counts.

pos	time in seconds 1cpu	2cpu	4cpu	8cpu	16cpu	speedup 2cpu	4cpu	8cpu	16cpu
1	—	—	—	—	—	---	---	---	---
2	1058	552	264	165	144	1.92	4.01	6.41	7.35
3	383	201	113	65	47	1.91	3.39	5.89	8.15
4	2086	1006	528	289	215	2.07	3.95	7.22	9.70
5	3224	1618	841	461	318	1.99	3.83	6.99	9.11
6	144	80	42	24	13	1.80	3.43	6.00	11.08
7	597	315	185	116	109	1.90	3.23	5.15	5.48
8	75	43	27	21	14	1.74	2.78	3.57	5.36
9	747	369	194	141	98	2.02	3.85	5.30	7.62
10	1210	591	328	188	155	2.05	3.69	6.44	7.81
11	699	373	205	129	123	1.87	3.41	5.42	5.59
12	3821	2077	1095	451	287	1.84	3.49	8.04	13.31
13	680	360	197	109	53	1.89	3.45	6.24	12.83
14	1142	503	247	151	122	2.27	4.62	7.56	9.36
15	960	499	258	145	112	1.92	3.72	6.62	8.57
16	2480	849	523	321	250	2.92	4.74	7.73	9.88
17	742	386	239	127	67	1.92	3.10	5.84	11.07
18	1996	973	603	354	308	2.05	3.31	5.64	6.48
19	749	379	208	131	115	1.98	3.60	5.72	6.51
20	2689	1148	590	364	301	2.34	4.56	7.39	8.48
21	2627	1313	801	459	295	2.00	3.28	5.72	8.82
22	890	427	248	137	123	2.08	3.59	6.50	7.24
23	620	320	185	111	97	1.94	3.35	5.59	6.39
24	769	397	218	133	94	1.94	3.53	5.78	8.18
tot	30388	14779	8139	4589	3460	2.02	3.73	6.64	8.81
mse	0	3	18	61	73	.00	.01	.09	.19

Table 7.4: DTS algorithm speedup.

CHAPTER 8

CONCLUSIONS

8.1. Summary

The PVS, EPVS, and DTS algorithms represent an organized algorithm evolution for searching alpha-beta trees on parallel machines. The DTS algorithm sets a new "high water mark" for parallel performance, but still falls far short of optimal performance for large numbers of processors. [28]

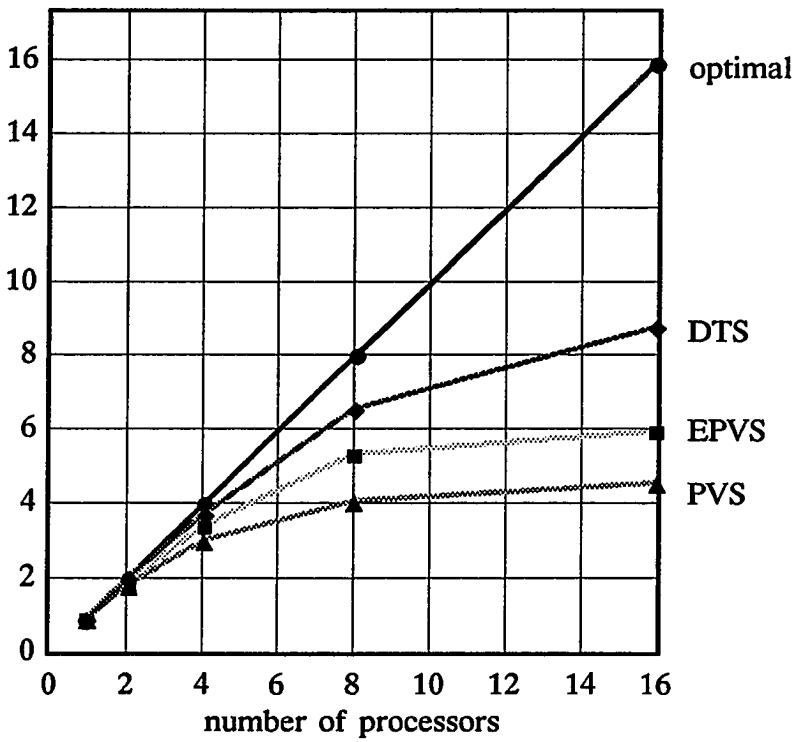

Figure 8.1: speedups for all algorithms on alpha-beta game trees

Figure 8.1 depicts the speedups for all three algorithms (PVS, EPVS and DTS) as well as the optimal speedup line. Note the closeness of the algorithms for four processors, but that by eight the advantages of each improvement over

its predecessor becomes more apparent. Also note that DTS produces significant improvement when going from eight to sixteen processors while the other algorithms hardly produce any. The final point in figure 8.1 is how far all of the algorithms diverge from optimal as the number of processors increases. This problem is discussed later and is not caused solely by a poor parallel algorithm, but is influenced by shortcomings in the sequential algorithm as well.

That move ordering is important for the sequential alpha–beta algorithm was clearly shown in chapter 4. What might not have been gleaned from that mathematical presentation is that poor move ordering reduces the performance of a parallel algorithm even more. Since this code is more interested in the "time to best move" rather than the "time to completely search the alpha–beta tree," move ordering at ply one is very important. If the code must change its mind many times at ply one, the search overhead climbs rapidly. As a processor finds a move that is better than the best so far (but does not know how much better without searching it again with relaxed search bounds), it stops all busy processors so that they can help with the new best branch. The portion of the tree search done by these interrupted processors is lost when they stop. If this happens many times (as it does when the program changes its mind repeatedly), the overhead grows and there is nothing that can be done about it.

If move ordering is wrong farther down into the tree, a type three node becomes a type 2 node. When processors split at such a node before realizing what type of node it is, search overhead increases. In the worst case, no speedup whatever is obtained since all of the work done in parallel is useless because it is not searched by the sequential algorithm.

Testing done on the Sequent exhibits both of these problems due to the relatively shallow searches that the Sequent can perform. Increasing the depth

of the test problems by one ply significantly improves the performance of the search, as the deeper searches change their mind less frequently (a fact observed by playing many similar positions using more powerful Cray XMP and Cray YMP computers.) In fact, the tabular data obtained from the Sequent is significantly worse than experiments run on the Crays in years past. We anticipate better results when extensive test time on a Cray YMP becomes available (eight processors) and then anticipate even better parallel performance when a Cray–3 (32 processors with significantly faster processors) becomes available.

It is important to note, however, that every performance gain obtained on the Sequent directly carries over to the Cray, but is multiplied by some factor (>1) due to the improved performance obtained with deeper searches.

Interpretation of the performance data in chapters 5, 6, and 7 requires a certain level of caution. Due to the timing variance when running the same problem many times with the same number of processors, each data point represents an *average*. As the number of processors was increased, the number of test runs for each problem was also increased so that the average of these runs stabilized. The times for the two processor tests hardly varied at all, but the times for the sixteen processor tests sometimes varied by a factor of two. This averaging tended to smooth this out.

Note also that the speedup averages represent the average time to search the entire problem set with a given number of processors. This lets the longer problems "outweigh" the short problems and affect the speedup more. While a case can be made for computing the speedup either way, we view the set of problems as a single entity that must be searched as quickly as possible, rather than as a set of twenty–four problems that are unrelated. It turns out that

computing the time for each problem and averaging the sum produces a slightly higher performance figure, but when the total time is measured it does not generate the same result. The reader can decide on which method gives a more useful result; however, all results reported herein are consistently computed as explained.

8.2. Conclusions

While this algorithm represents current state-of-the-art to search alpha-beta trees in parallel, it is still far from optimal in terms of the speedup obtained for large numbers of processors (>8).

One significant research topic deserving more attention is the mathematical analysis of alpha-beta trees relative to the parallel search performance issue. In particular, by quantitatively measuring the move ordering ability of this code, it should be possible to precisely define the upper bound for the performance of a parallel algorithm. It is difficult to find ways to improve the performance without knowing how well the current code compares to the theoretical optimum. While we would like for sixteen processors to run sixteen times as fast, if that is impossible, the upper bound would be useful information when evaluating changes and making (or trying to make) improvements in the parallel search.

It is interesting to note that DTS provides excellent performance on all tests cases regardless of the branching factor of the tree searched. Even more importantly, for those test cases where the program produces reasonable move ordering, the performance of the parallel search is really outstanding, as was seen in chapter 7. Those problems with large speedups were directly aided by good move ordering, so that the search overhead did not increase significantly as the number of processors was increased.

8.3 Future Work

The overall efficiency of the DTS algorithm is quite high. However, as the number of processors increases, the interactions between the processors increases (help requests, unsplit requests, share requests, stop requests, etc.) and serves to limit the ultimate performance of the algorithm. Some method for separating the processors into groups is probably necessary so that the volume of traffic between groups is at least an order of magnitude lower than the volume of traffic between processors in a group. How this will work out remains to be determined.

The search also needs to understand the concept of "futility" and not choke the system with help requests and attempted split operations when only a few nodes are left to examine. Even though the overhead of this code is very low, it is still non-zero and enough excitation can make it much more than merely measurable.

Splitting in the quiescence search is risky since it is very difficult to identify type 2 and type 3 nodes there. It is desirable to be able to split the quiescence search when it is complicated, or when it is the only work left. Unfortunately, it easily drives the search overhead upward because splits are done at type 2 nodes. A solution to this problem must exist, but the current implementation does not contain it.

Finally, the split process itself must become more dynamic when finding a split point. It doesn't need to be as careful when only one processor is waiting on work, but when many are waiting, the idle time accumulated while the help request ripples from processor to processor quickly adds up. A threshold is needed to trigger a help broadcast rather than a help circulate type of operation when the code notices that much computational power is idle. It must also

recognize the "feeding frenzy" and reconcile itself to the fact that at some point in the tree, processors are going to be idle and there is nothing it can do to prevent it. Some type of "futility indicator" is needed that defers split attempts as long as it is active to reduce the thrashing at the end of each search.

REFERENCES

[1] S. Akl, D. Barnard, and R. Doran, "The Design, Analysis and Implementation of a Parallel Alpha-Beta Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4 (1982), 192-203.

[2] S. Akl and R. Doran, "A Comparison of Parallel Implementations of the Alpha-Beta and Scout Tree Search Algorithms Using the Game of Checkers," Technical Report TR 81-121, Computing and Information Science Department, Queen's University, Kingston (1981).

[3] B. Awerbuch, "A New Distributed Depth-First Search Algorithm," *Information Processing Letters,* 20 (1985), 147-150.

[4] G. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Ph.D. Dissertation, Carnegie-Mellon University, Pittsburg, Pa. (1978).

[5] M. Campbell, "Algorithms for the Parallel Search of Game Trees," M.Sc. Thesis, Technical Report TR 81-8, Computer Science Department, University of Alberta, Edmonton (1981).

[6] R. Finkel and J. Fishburn, "Parallelism in Alpha-Beta Search," *Artificial Intelligence,* (1982), 89-106.

[7] R. Finkel and J. Fishburn, "Improved Speedup Bounds for Parallel Alpha-Beta Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5, (1983), 89-92.

[8] J. Fishburn and R. Finkel, "Parallel Alpha-Beta Search on Arachne," Technical Report 394, Computer Science Department, University of Wisconsin, Madison (1980).

[9] J. Fishburn, "Analysis of Speedup in Distributed Algorithms," Ph.D. Dissertation, University of Wisconsin, Madison (1981).

[10] R. Hyatt, A. Gower, and H. Nelson, "Cray Blitz," *Advances in Computer Chess 4*, Pergammon Press (1986), 8-18.

[11] R. Hyatt, H. Nelson, A. Gower, "Cray Blitz – 1984 Chess Champion," *Telematics and Informatics*, 2 (1986), 299–305.

[12] R. Hyatt, H. Nelson, and B. Suter, "A Parallel Alpha/Beta Tree Searching Algorithm," accepted by *Parallel Computing*, to be published Spring, 1989.

[13] D. Knuth and R. Moore, "An Analysis of Alpha–Beta Pruning," *Artificial Intelligence*, 6 (1975), 293–326.

[14] G. Li and B Wah, "Coping with Anomalies in Parallel Branch–and–Bound Algorithms," *IEEE Transactions on Computers*, C–35 (1986), 568–573.

[15] T. Lai and S. Sahni, "Anomalies in Parallel Branch–and–Bound Algorithms," *Communications of the ACM*, 27 (1984), 594–602.

[16] G. Lindstrom, "The Key Node Method: A Highly Parallel Alpha–Beta Algorithm," Technical Report UUCS 83–101, Department of Computer Science, University of Utah, (1983).

[17] T. Marsland, "A Review of Game–Tree Pruning," *ICCA Journal*, 9 (1986), 3–19.

[18] T. Marsland and F. Popowich, "A Multiprocessor Tree–searching System Design," Technical Report TR83–6, Department of Computing Science, University of Alberta (1983).

[19] T. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees," *ACM Computing Surveys*, 4 (1982), 533–551.

[20] T. A. Marsland and F. Popowich, "Parallel Game–Tree Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–7 (1985), 442–452.

[21] T. Marsland, M. Campbell, and A. Rivera, "Parallel Search of Game Trees," Technical Report TR 80–7, Computing Science Department, University of Alberta, Edmonton (1980).

[22] T. Marsland and M. Campbell, "Methods for Parallel Search of Game Trees," Proceedings of the 1981 International Joint Conference on Artificial Intelligence.

[23] T. Marsland, M. Olafsson, and J. Schaeffer, "Multiprocessor Tree–Search Experiments," *Advances in Computer Chess 4*, Pergammon Press (1986), 37–51.

[24] H. Nelson and R. Hyatt, "Hash Tables in Cray Blitz," *ICCA Journal*, 8 *(1985) 3–13.*

[25] M. Newborn, "A Parallel Search Chess Program," Proceedings, ACM Annual Conference, (1985), 272–277.

[26] J. Pearl, "Scout: A Simple Game–Searching Algorithm with Proven Optimal Properties," Proceedings of the First Annual National Conference on Artificial Intelligence, Stanford, (1980).

[27] F. Popowich and T. Marsland, "Parabelle: Experiments with a Parallel Chess Program," Technical Report TR83–7, Computing Science Department, University of Alberta, Edmonton (1983).

[28] J. Schaeffer, "Experiments in Distributed Game–Tree Searching," Technical Report TR87–2, Computing Science Department, University of Alberta, Edmonton (1987).

[29] C.E. Shannon, "Programming a Digital Computer for Playing Chess," *Philosophical Magazine*, 41 *(1950)*, *256–275.*

[30] B. Wah, G. Li, and C. Yu, "Multiprocessing of Combinatorial Search Problems, *Computer*, 18 (1985), 93–108.

GRADUATE SCHOOL
UNIVERSITY OF ALABAMA AT BIRMINGHAM
DISSERTATION APPROVAL FORM

Name of Candidate Robert Morgan Hyatt

Major Subject Computer and Information Sciences

Title of Dissertation A High-Performance Parallel Algorithm to Search Depth-First Game Trees

Dissertation Committee:

Bruce W. Suter, Chairman

Kevin D. Reilly

Charles R. Kalmbach

Stephen C. Harvey

Warren D. Jones

Director of Graduate Program Warren D. Jones

Dean, UAB Graduate School Anthony Ramond

Date 12/29/88

PS-1428