

# Exploiting Parallelism in Game-Playing Programs

by

Philip H. Chu

Submitted to the Department of Electrical Engineering and  
Computer Science

in partial fulfillment of the requirements for the degree of

Bachelor of Science

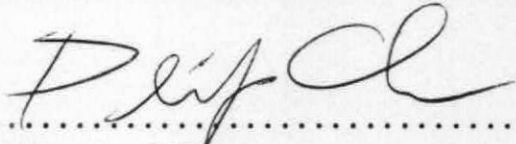
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1988

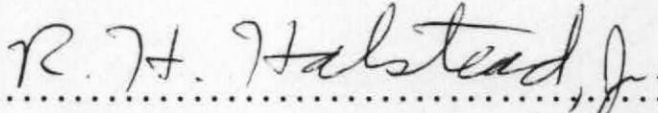
© Philip H. Chu, 1988

The author hereby grants to MIT permission to reproduce and  
to distribute copies of this thesis document in whole or in part.

Signature of Author .....  .....

Department of Electrical Engineering and Computer Science

May 16, 1988

Certified by .....  .....

Robert H. Halstead, Jr.

Associate Professor, Laboratory of Computer Science

Thesis Supervisor

Accepted by .....

Leonard A. Gould

Chairman, EECS Departmental Committee

# Exploiting Parallelism in Game-Playing Programs

by

Philip H. Chu

Submitted to the Department of Electrical Engineering and Computer Science  
on May 16, 1988, in partial fulfillment of the  
requirements for the degree of  
Bachelor of Science

## Abstract

The Von Neumann bottleneck present in traditional computers has prompted the development of various parallel architectures and programming languages. Among the latter is Multilisp, a version of Scheme that includes several parallel-task spawning constructs.

Multello, an Othello-playing program written in Multilisp, is used to demonstrate several parallel alpha-beta algorithms. Subsequently, attempts at finer grain parallelization are made. Runtime results are generated on Concert, an experimental multiprocessor. Conclusions about Multilisp, Concert, and the effectiveness of the parallel searches tested are then drawn.

Thesis Supervisor: Robert H. Halstead, Jr.

Title: Associate Professor, Laboratory of Computer Science

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Summary . . . . .	9
1.2	Organization . . . . .	9
1.3	Multilisp . . . . .	10
1.3.1	Constructs for Parallelism . . . . .	10
1.3.2	Using Futures . . . . .	11
1.3.3	Concert . . . . .	12
1.4	Objectives . . . . .	13
1.4.1	Evaluating Multilisp . . . . .	13
1.4.2	Comparing Search Techniques . . . . .	13
<b>2</b>	<b>Game-Playing</b>	<b>15</b>
2.1	The Computer as Player . . . . .	15
2.1.1	History . . . . .	15
2.1.2	Concepts . . . . .	16
2.1.3	Othello . . . . .	17
2.2	Adversary Search . . . . .	18
2.2.1	Minimax . . . . .	18
2.2.2	Alpha-Beta . . . . .	21
2.3	Details . . . . .	25
2.3.1	Static Evaluation . . . . .	25

<i>CONTENTS</i>	4
2.3.2 Making a Move . . . . .	26
2.3.3 Finding Legal Moves . . . . .	29
<b>3 Adding Parallelism</b>	<b>32</b>
3.1 Search . . . . .	32
3.1.1 Parallel Exhaustive Search . . . . .	32
3.1.2 Mandatory Work First . . . . .	33
3.1.3 Speculative Search . . . . .	37
3.1.4 Parallel-Aspiration Search . . . . .	50
3.2 Finer Grain Parallelism . . . . .	52
3.2.1 Static Evaluation . . . . .	52
3.2.2 Flipping Pieces . . . . .	55
3.2.3 Move Generation . . . . .	56
<b>4 Results</b>	<b>60</b>
4.1 Gathering Data . . . . .	60
4.1.1 A Test Case . . . . .	60
4.1.2 Parallelism Profiles . . . . .	60
4.1.3 Sequential Search . . . . .	61
4.1.4 Parallel Exhaustive Search . . . . .	62
4.1.5 Using Mandatory Work-First . . . . .	64
4.1.6 Speculative Search . . . . .	66
4.1.7 Using Parallel-Aspiration Search . . . . .	69
<b>5 Concluding Remarks</b>	<b>73</b>
5.1 Conclusions . . . . .	73
5.1.1 Parallel Search . . . . .	73
5.1.2 Multilisp . . . . .	74
5.2 Future Work . . . . .	75
5.2.1 Enhancing Multilisp . . . . .	75



*CONTENTS*

5

5.2.2 More on Searching . . . . . 76

## List of Figures

2-1	The opening position of an Othello game . . . . .	18
2-2	Exhaustive minimax search . . . . .	20
2-3	Alpha-beta search . . . . .	24
2-4	Values given by Multello to board positions for, (left) mid-game, and right) end-game. . . . .	25
2-5	Procedure for static evaluation during mid-game. . . . .	27
2-6	Procedure for static evaluation at end of the game. . . . .	27
2-7	Procedures to make a move. . . . .	28
2-8	Procedure to update list of potential moves. . . . .	29
2-9	Procedure to generate list of legal moves. . . . .	30
2-10	Procedure to check if a move is legal. . . . .	31
3-1	Exhaustive search with futures added. . . . .	34
3-2	mandatory-work first procedure . . . . .	38
3-3	Operations on shared alpha-beta window for speculative search . . .	41
3-4	An <i>ad hoc</i> task-killing mechanism for speculative search . . . . .	42
3-5	The expression that spawns off speculative tasks to evaluate a node's children. . . . .	42
3-6	Procedure for atomically updating a shared alpha-beta window. . . .	43
3-7	A speculative alpha-beta search. . . . .	45
3-8	Tree-splitting using more frequent updating . . . . .	46
3-9	An example of very suboptimal scheduling . . . . .	48

3-10	Using <i>dfuture</i> in "speculative" search. . . . .	48
3-11	parallel aspiration . . . . .	50
3-12	A parallel static evaluation procedure . . . . .	53
3-13	A parallel static evaluation procedure for end of game. . . . .	54
3-14	Parallel version of move procedure . . . . .	56
3-15	Parallel updating of the border. . . . .	57
3-16	parallel move generation . . . . .	57
3-17	parallel move verification procedrue . . . . .	59
4-1	A parallelism profile for a three-ply depth-first search. . . . .	61
4-2	Parallelism profile for parallel-exhaustive search on a four-ply search tree. . . . .	63
4-3	Profile of the Mandatory-Work-First procedure . . . . .	65
4-4	Speculative Search . . . . .	67
4-5	Speculative Search with more cutoff checks . . . . .	68
4-6	Speculative Search using <i>Dfuture</i> . . . . .	70
4-7	Parallel-aspiration search using fifteen alpha-beta partitions. . . . .	71

## List of Tables

4.1	Comparison of exhaustive minmax and alpha-beta search times. . . .	62
4.2	Statistics for parallel exhaustive search . . . . .	62
4.3	Statistics for mandatory work first search . . . . .	65
4.4	Results from speculative search . . . . .	66
4.5	Results from speculative search with multiple updates per node. . .	67
4.6	Statistics on performance of the parallel aspiration search . . . . .	70



# Chapter 1

## Introduction

### 1.1 Summary

The Von Neumann bottleneck present in traditional computers has prompted the development of various parallel architectures and programming languages. Among the latter is Multilisp, a version of Scheme that includes constructs for explicit parallelism.

This thesis demonstrates several parallel alpha-beta algorithms implemented in Multilisp. The game Othello is used as a platform for testing these algorithms. In addition, the use of finer-grain parallelism is explored, specifically in static evaluation and move generation. Runtime results generated on Concert, an experimental multiprocessor, are then used to draw conclusions about the comparative effectiveness of the parallel searches, deficiencies and strengths of Multilisp, and criteria for parallel hardware designed to support Multilisp.

### 1.2 Organization

The first part of this thesis, Chapter One, will be devoted to a description of Multilisp, a summary of experimental facilities for running applications written in Mul-

tilisp, and an outline of the objectives of this thesis.

Chapter Two introduces the issues involved in programming a computer to play two-player games. Examples of typical adversary searches, move generation routines and static evaluation procedures are demonstrated in Multello, a Multilisp program that plays the board game Othello.

In Chapter Three, I describe various approaches to adding parallelism to the game program introduced in Chapter Two, ranging from conservative additions of *future* to the procedures given in Chapter Two to more radical approaches involving speculative computation.

Chapter Four presents the experimental results of running these various game-playing algorithms on an actual multiprocessor.

Finally, in Chapter Five, I make some conclusions and give some suggestions for future work.

Included in Appendix A are listings of algorithms for both the basic sequential searches and the parallel versions of these searches. Appendix B contains a description of the Multello support routines used in the Multilisp procedures listed in the thesis. A complete table of runtime results is listed in Appendix C. Appendix D details more fully the special constructs of Multilisp.

## 1.3 Multilisp

### 1.3.1 Constructs for Parallelism

Multilisp [14] is an extension of Scheme [1] [18] that features several operators for explicitly employing parallelism in programs.

The principal construct for explicitly employing parallelism in Multilisp is

(future X)

in which *X* is any Lisp expression. The invocation of *future* will spawn a separate

task to evaluate  $X$  and also immediately returns a placeholder of type *future* as the value of  $X$ . But before then, the placeholder can be passed as an argument or used as any other Lisp expression.

If a process needs to access the value of a future-surrounded expression before it has mutated, then the current process is suspended until the value has finished mutating.

The other two principal features of Multilisp are *touch* and *delay*. If  $X$  is not a future, then the expression (*touch*  $X$ ) merely returns  $X$ . If  $X$  is a future, then (*touch*  $X$ ) will suspend the current computation until  $X$  has been determined, and then returns  $X$ . *Strict* operators, functions that require the value of an argument upon receiving it, implicitly touch their arguments.

*Delay* creates an object of type *future* that is not evaluated until touched. This is useful for delayed evaluation in streams. *Delay* is actually a variant of *future*.<sup>1</sup>

### 1.3.2 Using Futures

The placement of futures in a program is a non-trivial issue. A naive approach to adding parallelism would be to place a *future* around each expression in a Multilisp program. However, the use of *future* is not without cost. At present, the overhead involved in setting up a future is approximately five times that invoked by a Multilisp function call. [13] Although the long term may bring architectures that are specially designed to run Multilisp or similar languages and thus may feature special hardware support for *future* handling, the cost of setting up a future is unlikely to drop beneath the cost of a function call. This, plus the fact that the number of processors available is not unlimited, makes Multilisp more suitable for large-grained parallelism than fine-grained parallelism.

Moreover, placement of futures around *strict* operators [10], i.e. operators that need to know the values of its arguments to proceed, would be a waste of time,

---

<sup>1</sup>A more complete description of Multilisp's special constructs is given in Appendix D.



since the strict computation will *touch* the future cell and consequently halt execution until the cell finishes mutating to the needed value. This would result in an essentially sequential evaluation, plus the additional overhead created by using *future*.

In summary, *future*'s must be placed judiciously to achieve good performance. The inclusion of explicit concurrency in Multilisp permits the use of side effects in combination with parallelism. This allows possibilities for large-grained parallelism not possible in a purely functional language. However, using futures with side effects as yet remains an art rather than a science. [13]

### 1.3.3 Concert

Multilisp currently runs on Concert, [15] an experimental multiprocessor consisting of thirty-four 68000 microprocessors with local memory and a shared global memory. The processors are grouped in several clusters connected by the *RingBus*, a ring-shaped segmented bus.

Concert is not intended to be a prototype for parallel computers but, rather, is a testbed for multiprocessor applications. The results of examining multiprocessor performance with Concert and multiprocessor applications built upon Multilisp can then be used to determine criteria that needs to be addressed in constructing a more suitable parallel architecture. Since the purpose of Concert is to provide a real platform upon which to examine multiprocessor applications, it is built from mostly off-the-shelf parts.

Each cluster on the RingBus features a RingBus Interface Board (RIB), a portion of global memory, and typically four to eight MC68000 microprocessors, each associated with its own local memory. These elements are connected by a high speed Multibus.



## 1.4 Objectives

### 1.4.1 Evaluating Multilisp

One obvious result of testing new programs in Multilisp is the opportunity to evaluate Multilisp in both its implementation and its design philosophy. Multello can be thought of as an addition to the growing library of programs already written in Multilisp.

Among these programs are the Multilisp compiler itself[14], a quicksort routine[14], a semantic net retrieval program[13], a digital circuit simulation[7], a program to solve the traveling salesman problem[13], and a program to calculate fibonacci numbers[14].

In addition to pointing out deficiencies and benefits of Multilisp, writing a variety of moderate-sized to large applications provides an exercise in learning how to program effectively in Multilisp, e.g. the placement of futures, and a more realistic assessment of Multilisp not provided by simple benchmarks.

Thus, the program introduced in this thesis, Multello, will further the twofold effect of:

1. exploring parallel programming styles and algorithms, and also
2. pointing out the strengths and weaknesses of Multilisp. Deficiencies in the current Multilisp implementation can then be corrected in conjunction with the development of more advanced multiprocessors.

### 1.4.2 Comparing Search Techniques

Finally, Multello provides an opportunity to compare different approaches to parallelizing searches. This thesis will allow analysis of the performance tradeoffs between concurrency and work-minimization. Furthermore, the importance of various parts of the game search, such as static evaluation and node generation, on the final

searchtime can be found through comparison of search performance with different combinations of parallelization of these components. And finally, although the search in question is an adversary search, the results of the comparison should have implications toward the general problem of parallelizing search.

# Chapter 2

## Game-Playing

### 2.1 The Computer as Player

#### 2.1.1 History

Computer game-playing dates back to about 1800, when a man named Von Kempelen toured Europe with a chess-playing robot dressed up to look like a Turk. The automaton won chess games in both Europe and the United States. Not surprisingly, the Turk was really a fake, hiding a human chess player.

More genuine game-playing computers began to appear in the 1950's, based on independent papers by Claude Shannon and Alan Turing, but the first of modern chess programs did not appear until Richard Greenblatt developed *MacHack* in 1967. Currently, programs with names such as Belle and Cray Blitz have tournament rankings of over 2300 <sup>1</sup>.

---

<sup>1</sup>Beginner have rankings of 600-900 and world chess champions typically have a rankings from 2600 to 2800

### 2.1.2 Concepts

A computer strategy for two-player games usually involves a *game tree* of possible moves. This enables the computer to “look ahead” beyond its immediate moves to its opponent’s possible responses and the computer’s responses to its opponent’s responses, and so on. Thus the game tree consists of alternating *max* and *min* levels, the former on which the computer attempts to maximize its winning opportunities and the latter on which the opponent attempts to minimize those opportunities.

Most interesting games typically involve an enormous number of paths to account for a complete game<sup>2</sup>, so in practice the search tree is explicitly bounded by a specified depth, and a *static evaluation* is used to appraise the “winningness” of the game situation at the terminal node in the form of some numerical value.

This *minmax* strategy rests heavily on the validity of the static evaluation. The usefulness of assigning a number to describe a game situation is questionable. One example is the *horizon effect*[20], which arises during dynamic situations, such as the exchange of pieces in chess. A search terminating in the midst of such an exchange will assign a value to a move without taking into account the fact that there is going to be a significant change on the next move. A way to combat this problem is to use *heuristic continuation*, which checks for such dynamic situations and extends the limit of search as necessary.

Another enhancement to adversary search is the use of *progressive deepening*, which involves performing static evaluations at each ply of the search tree before extending the search tree another ply. This technique can be used in order to traverse the game tree more selectively and to make better use of time in tournament-like situations in which there is a time limit for making moves. Progressive deepening is made feasible by the fact that for a tree with a uniform branching factor  $b$ , the ratio of the number of nodes in the bottom level to the remaining nodes in the tree

---

<sup>2</sup>There are more possible board positions in a fifty-move chess game than there are known atoms in the universe!



is

$$\frac{b^d(b-1)}{b^d-1} \approx b-1 \quad (2.1)$$

A fundamental deficiency of the minmax strategy is that it does not take into account variations of the maximize/minimize stratagem. For example, an opponent may be a relatively weak player who might follow a line of play that the minmax search would rule out. A good strategy then may be to lay traps and try to lure the opponent into making some common mistakes.

Another scenario along the same lines is the case where the computer is losing the game. In this case, following a line of play specified by the minmax stratagem would lead to certain defeat. Since the computer has nothing to lose, it would be far better to gamble and try to complicate the game for the opponent by making risky but potentially winning moves.

Despite these flaws, the best chess computers today are still the ones that search exhaustively and quickly. Belle, for example, uses special hardware to speed up its search.

### 2.1.3 Othello

This chapter introduces fundamentals of adversary searches using the game Othello as a platform. Portions of an Othello-playing Multilisp program, called Multello, are used to demonstrate typical algorithms for computer game-playing.

Othello is played by two players on an eight-by-eight board. The game begins with four pieces on the board. (Figure 2-1). The playing pieces are discs black on one side and white on the other.

The rules of Othello are fairly simple:<sup>3</sup>

1. A move is not legal if it doesn't flip at least one of the opposing player's pieces.

---

<sup>3</sup>For a description of the game Othello and its rules, see Appendix D.

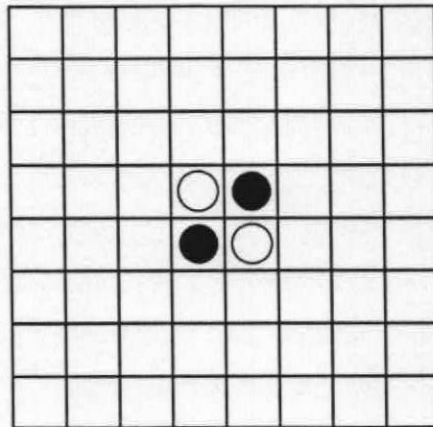


Figure 2-1: The opening position of an Othello game

2. If a player cannot make a legal move, he must pass.
3. If neither player can make a legal move, the game is over, and the player with the most pieces on the board wins.

To flip an opponent's piece, a player needs to *outflank* him by blocking the two ends of a line of one or more of his opponent's pieces. Once a row of pieces of one color has been outflanked, then these pieces are flipped over to display the color of the player who did the outflanking. a computer.

## 2.2 Adversary Search

### 2.2.1 Minimax

Following is a more formal description of the minimax algorithm:

To MINIMAX:

1. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the

result.

2. If the level is a minimizing level, use MINIMAX on the children of the current position. Report the minimum of the results.
3. Otherwise, the level is a maximizing level. Use MINIMAX on the children of the current position. Report the maximum of the results.

Another form of the minimax algorithm is Knuth's *negamax* algorithm. [17]

To NEGAMAX:

1. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
2. Otherwise, apply NEGAMAX on the children of the current position. Report the maximum of the negation of the results.

The minimax and negamax algorithms can be shown to be equivalent. It is often more convenient to use the minimax form for visualization of the search tree and to do analysis of the algorithm in negamax form. <sup>4</sup> A minimax procedure is listed in Figure 2-2 in negamax form. <sup>5</sup>

The procedure *DEPTH-FIRST* begins (Line 29) by calling the inner procedure *MAXIMIZE* on the current game, represented by global variable *\*game\**, along with the information that the search is at level zero.

*MAXIMIZE* first checks if any more moves can be made on the board by either player (Line 16). If not, then the game is over and a static evaluation for the end of the game is returned (Line 17).

---

<sup>4</sup>For the remainder of this thesis we will mostly use the terms *minimax* and *negamax* synonymously.

<sup>5</sup>A description of the Multello routines called by the code examples of this chapter is listed in Appendix B.

```

1. (define (depth-first bottom)
   'A depth-first minimax search.'

2. (define (maximize game level)

3. (define (max1 moves best)
   "assumes (car move) is legal move or moves is nil."
4. (if (no-move? moves)
5.     best
6.     (let* ((move (car moves))
7.            (new-value
8.              (minus
9.                (choice-value
10.                 (maximize (make-child game move)
11.                            (add1 level))))))
12.      (if (> (choice-value best) new-value)
13.          (max1 (cdr moves) best)
14.          (max1 (cdr moves)
15.                (make-choice move new-value))))))

16. (cond ((game-end? game)
17.        (make-choice (null-square) (end-eval game)))
18.      ((= level bottom)
19.        (make-choice (null-square) (static-eval game)))
20.      (t
21.        (let ((new-moves (legal-moves game)))
22.          (if (no-move? new-moves)
23.              (make-choice (null-square)
24.                            (minus (choice-value
25.                                    (maximize (pass game)
26.                                               (add1 level))))))
27.              (max1 new-moves (make-choice (null-square)
28.                                             (neg-inf))))))))

29. (choice-move (maximize *game* (level-0)))

```

Figure 2-2: Exhaustive minimax search



If the end of the game has not arrived, then *MAXIMIZE* checks whether the limit of search has been reached (Line 18). If the bottom of the search tree has indeed been reached, then a static evaluation for a mid-game board is returned (Line 19).

If neither of the above conditions are true, then *MAXIMIZE* proceeds to extend the search tree. If the player whose turn it is at this level has no move and therefore must pass, then *MAXIMIZE* simply hands a copy of this board on to the next search level (thus this node has one child).

Otherwise, *MAXIMIZE* takes the list of legal moves and passes it on to the tail-recursive procedure *MAX1*, which iterates through this list, generating a successor node for each possible move at this ply, and then applying *MAXIMIZE* to the successor (Lines 10-11). During this loop, *MAX1* keeps track of the highest valued child thus far, and when the loop terminates, returns the move used to create the child and the value associated with it (Line 8).

When the search is complete, the highest-valued move possible at the root node *\*game\** is returned (Line 29).

### 2.2.2 Alpha-Beta

As noted before, the minmax algorithm produces an exponentially growing search tree. A tree of depth  $d$  with a branch factor  $b$  will have  $b^d$  terminal nodes. This is a serious problem when players of the game in question normally have several options per turn, i.e. the search tree has a high branch factor. One method to combat this problem is to cut down the number of game paths explored by *alpha-beta pruning*.<sup>[20]</sup>

Alpha-beta pruning<sup>6</sup> is made possible by the realization that if the player knows his opponent can achieve a certain minimum score, than the player cannot possibly do better than that score along this search subtree. This allows him to “prune” the

---

<sup>6</sup>Knuth [17] gives a good summary of the this algorithm's history

rest of the subtree, that is, ignore it.

TO MINIMAX with ALPHA-BETA:

1. If the level is the top level, let  $\alpha = -\infty$  and  $\beta = \infty$ .
2. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
3. If the level is a minimizing level, then until all children are examined by MINIMAX or  $\alpha \geq \beta$ :
  - (a) Set  $\beta$  to the smaller of the given *beta* values and the smallest value so far reported by MINIMAX working on the children.
  - (b) Use MINIMAX on the next child of the current position, handing this new application of MINIMAX the current  $\alpha$  and  $\beta$ .Report  $\beta$ .
4. If the level is a maximizing level, then until all children are examined with MINIMAX or  $\alpha \geq \beta$ :
  - (a) Set  $\alpha$  to the larger of the given *alpha* values and the biggest value so far reported by MINIMAX working on the children.
  - (b) Use MINIMAX on the next child of the current position, handing this new application of MINIMAX the current  $\alpha$  and  $\beta$ .Report  $\alpha$ .

There is also a "negabeta" version of this algorithm:

To NEGABETA:

1. If the level is the top level, let  $\alpha = -\infty$  and let  $\beta = \infty$ .

2. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
3. Until all children are examined with MINIMAX or  $\alpha \geq \beta$ :
  - (a) Set  $\alpha$  to the larger of the given  $\alpha$  values and the maximum of the negation of the values so far reported by MINIMAX working on the children.
  - (b) Use MINIMAX on the next child of the current position, handing this new application of MINIMAX  $-\beta$  as its  $\alpha$  and  $-\alpha$  as its  $\beta$ .

Report  $\alpha$ .

The minimum number of static evaluations needed to discover the best move in an optimally arranged search tree is given by the following formula, where  $s$  is the minimum number of static evaluations,  $b$  is the branch factor, and  $d$  is the depth of the tree:

$$s = \begin{cases} 2b^{d/2} & \text{for } d \text{ even} \\ b^{(d+1)/2} + b^{(d-1)} - 1 & \text{for } d \text{ odd} \end{cases} \quad (2.2)$$

Note, however, that although the growth of the search tree is slower, it is still exponential.

The alpha-beta <sup>7</sup> procedure (Figure 2-3) only differs from the minimax one in the inclusion of the  $\alpha$  and  $\beta$  parameters accepted by *MAXIMIZE* and (Lines 14-15) where there is a check for the  $\alpha \geq \beta$  condition. If this cutoff condition is satisfied, *MAXIMIZE* merely returns (*cutoff-choice*), a null-move (*nil*) with the highest possible value,  $\infty$ . This value will be recognized as  $-\infty$  by the parent node and thus cannot supersede any other move.

---

<sup>7</sup>We shall use "alpha-beta" from now on to refer to the "nega-beta" procedure.



```

1. (define (alpha-beta bottom
2.           &optional (alpha (neg-inf)) (beta (inf)))
   'an alpha-beta search')

3. (define (maximize game level alpha beta)

4.   (define (max1 moves best)
5.     (if (no-move? moves)
6.         best
7.         (let* ((move (car moves))
8.               (new-value (minus
9.                           (choice-value
10.                            (maximize (make-child game move)
11.                                       (add1 level)
12.                                       (minus beta)
13.                                       (minus (choice-value best)))))))
14.           (cond ((>= new-value beta)
15.                  (cutoff-choice))
16.                 (t
17.                  (max1 (cdr moves)
18.                        (if (> new-value (choice-value best))
19.                            (make-choice move new-value)
20.                            best)))))))

21.   (cond ((game-end? game)
22.          (make-choice (null-square) (end-eval game)))
23.         ((= level bottom)
24.          (make-choice (null-square) (static-eval game)))
25.         (t
26.          (let ((new-moves (legal-moves game)))
27.              (if (no-move? new-moves)
28.                  (make-choice (null-square)
29.                                (minus (choice-value
30.                                        (maximize (pass game) (add1 level)
31.                                                  (minus beta)
32.                                                  (minus alpha))))))
33.                  (max1 new-moves (make-choice (null-square)
34.                                                alpha))))))

35. (choice-move (maximize *game* (level-0) alpha beta)))

```

Figure 2-3: Alpha-beta search



25	3	20	18	18	20	3	25
3	1	6	8	8	6	1	3
20	6	14	12	12	14	6	20
18	8	12	10	10	12	8	18
18	8	12	10	10	12	8	18
20	6	14	12	12	14	6	20
3	1	6	8	8	6	1	3
25	3	20	18	18	20	3	25

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Figure 2-4: Values given by Multello to board positions for, left) mid-game, and right) end-game.

## 2.3 Details

### 2.3.1 Static Evaluation

One reasonably effective method to judge a board position in Othello is to assign to each square on the board a position value (Figure 2-4. For example, the corner squares would have higher values than any of the other squares since a piece occupying a corner square cannot be outflanked.

The form of the static evaluation can then be given as

$$s = \sum_{i=1}^8 \sum_{j=1}^8 v_{i,j} \quad (2.3)$$

$$v_{i,j} = \begin{cases} p_{i,j} & \text{if square is occupied by player} \\ -p_{i,j} & \text{if square is occupied by opponent} \end{cases} \quad (2.4)$$

where  $p_{i,j}$  is the value of the board position denoted by  $i$  and  $j$ .

The algorithm for this evaluation is as follows:

To EVALUATE:

1. Let  $s$  be the sum of the values of the squares occupied by the current player.
2. Subtract from  $s$  the values of the squares occupied by opposing player.
3. Report  $s$ .

The evaluation of the end of the game also follows this form, since the winner of the end of the game is determined by the number of pieces each player has on the board. This evaluation can be done by simply substituting a constant value, say one, for each board position.

Figure 2-5 and Figure 2-6 list the procedures evaluating both mid-game and end-of-game boards, respectively. The function *EVAL-SQUARE* returns the mid-game value of a square, and *FINAL-EVAL-SQUARE* returns the end-game value of a square, which is always one. Both procedures iterate through all the squares on the board, represented by the global variable *\*SQUARES\**, and sum up values according to who occupies the square, adding zero if the square is unoccupied, adding the square value if it is occupied by the player, and subtracting the value if it is occupied by the opponent (This determination is made by *EVAL-SQUARE* and *FINAL-EVAL-SQUARE*).

### 2.3.2 Making a Move

The procedure to make a move, *DO-FLIPS* calls the function *FLIP1*, which will flip the appropriate pieces in the specified direction if those pieces are outflanked by moving onto *SQUARE*. *DO-FLIPS* calls *FLIP1* eight times, one in each of the possible flipping directions. *FLIP1* works by first retrieving the list of squares on which pieces can be flipped and then modifying those squares.

```
1. (define (evaluate board color)
    "evaluate board position (with respect to color)"

2.   (define (eval1 move-list)
3.     (if (null move-list)
4.         0
5.         (+ (eval-square board (car move-list) color)
6.            (eval1 (cdr move-list)))))

7.   (eval1 *squares*))
```

Figure 2-5: Procedure for static evaluation during mid-game.

```
1.(define (fevaluate board color)
    "evaluate board position (with respect to color)"

2.   (define (eval1 move-list)
3.     (if (null move-list)
4.         0
5.         (+ (final-eval-square board (car move-list) color)
6.            (eval1 (cdr move-list)))))

7.   (eval1 *squares*))
```

Figure 2-6: Procedure for static evaluation at end of the game.

1. (define (do-flips board square color)  
    ‘‘Makes move on board.  
    Modifies board.’’)
2. (flip1 board square (east) color)
3. (flip1 board square (ne) color)
4. (flip1 board square (north) color)
5. (flip1 board square (nw) color)
6. (flip1 board square (west) color)
7. (flip1 board square (sw) color)
8. (flip1 board square (south) color)
9. (flip1 board square (se) color)
10. (put-piece color board square))

Figure 2-7: Procedures to make a move.

At any point during an Othello game, each piece on the board is adjacent to at least one other non-empty square, since each move requires placing a piece next to at least one other (opposing) piece, and the game begins with a contiguous block of pieces in the center of the board. To take advantage of this property, Multello maintains a list of the empty squares surrounding this contiguous block in order to cut down the number of squares that should be checked for possible moves. Since each additional move “expands” this block, this list needs to be updated each time a move is made.

The procedure that performs this update (Figure 2-8) first removes the square just recently occupied by the move from the list, and then checks in each of the possible eight flipping directions adjacent to the move just recently made for the following conditions:

1. The square is on the board (Line 3).
2. The square is empty (Line 4).
3. The square is not already in the list (Line 5).



```

1. (define (update board move-list square)
   'Update list of empty squares surrounding pieces on board.')

2. (define (add square move-list)
   'Checks if square is real square, empty, and not already
   in list. Then atomically inserts it in list.')

3. (if (and (not (null square))
4.         (empty? board square)
5.         (not (member square move-list))))
6.     (insert square move-list))

7. (let ((new-list (remove square (copy move-list))))
8.     (add (inc-square square (east)) new-list)
9.     (add (inc-square square (ne)) new-list)
10.    (add (inc-square square (north)) new-list)
11.    (add (inc-square square (nw)) new-list)
12.    (add (inc-square square (west)) new-list)
13.    (add (inc-square square (sw)) new-list)
14.    (add (inc-square square (south)) new-list)
15.    (add (inc-square square (se)) new-list)
16.    new-list))

```

Figure 2-8: Procedure to update list of potential moves.

If all of the conditions are satisfied, then the adjacent square is added to the list.

### 2.3.3 Finding Legal Moves

The task of finding legal moves is reduced by the maintenance of a list of empty squares surrounding the pieces currently on the board. The procedure to find legal moves essentially just filters out all members of the list that do not constitute a move (Figure 2-9).

Since a move is legal if and only if it flips at least one of the opposing player's pieces, the procedure for checking if a move is legal is similar to the procedure for

```
1. (define (get-moves board moves color)
    'Returns a list of all legal moves (for COLOR) in MOVES.')
```

```
2. (cond ((no-move? moves)
3.       (null-square))
4.       ((flip-possible? board (car moves) color)
5.        (cons (car moves)
6.              (get-moves board (cdr moves) color))))
7.       (t
8.        (get-moves board (cdr moves) color))))
```

Figure 2-9: Procedure to generate list of legal moves.

actually making a move (Figure 2-10).

Like the procedure *DO-FLIPS*, the *FLIP-POSSIBLE?* function retrieves (using *FLIP*) lists of squares on which pieces can be flipped by this particular move, but instead of actually flipping them, it merely returns them. The *or* predicate (Line 2) then checks if any of these lists actually contain any squares, in which case this square presents a legal move. If all the lists are empty, then no opposing piece can be flipped and the square therefore does not present a legal move.

```
1. (define (flip-possible? board square color)
    "Returns t if move by color is possible on square"

2.  (or
3.    (flip board square (east) color)
4.    (flip board square (ne) color)
5.    (flip board square (north) color)
6.    (flip board square (nw) color)
7.    (flip board square (west) color)
8.    (flip board square (sw) color)
9.    (flip board square (south) color)
10.   (flip board square (se) color)))
```

Figure 2-10: Procedure to check if a move is legal.

# Chapter 3

## Adding Parallelism

### 3.1 Search

#### 3.1.1 Parallel Exhaustive Search

To reduce search time we might consider processing each node in the search tree in parallel. To parallelize the exhaustive search routine, we need to be able to generate and evaluate the children of a node concurrently.

To NEGAMAX:

1. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
2. Otherwise, apply NEGAMAX on the children of the current position *in parallel*. Report the maximum of the negation of the results.

This can be done by placing *future* around the expression in Lines 7–11 (Figure ??), which evaluates a child of the node currently being processed.

To be thorough, we wrap a *future* around the expression in Line 24, which evaluates a child in the special case when the player must pass and thus the node



at that position has only one child, identical except for the color of the player.

To ensure that the children of a node can be evaluated concurrently, any expression that touches one of the child values must be enclosed with *future*, also. Line 13, which compares the value of the most recently evaluated move to the best move found so far, is such an operation. We enclose this in *future* and now we have created a parallel exhaustive minimax search by simply adding three futures and not altering any of the code (Figure 3-1).

An alternative to this depth-first search is a breadth-first approach, in which each *new-value* future would be placed on a queue. To achieve maximum concurrency, however, we still would extract the highest value from the queue using a future for each comparison. Consequently, we would not be saving ourselves the use of any invocations of *future* if we used breadth-first instead of depth-first search.

This transformation of a depth-first search to a concurrent search merely by enclosing two expressions with *future* demonstrates the suitability of using futures for at least some applications. The algorithm is essentially unchanged—we have simply pointed out which expressions could benefit by wrapping *future* around them. As for the expected performance of this algorithm, we can expect a fairly large and constant burst of parallelism, probably saturating the available number of processors on anything other than a shallow search tree.

### 3.1.2 Mandatory Work First

Although we can achieve a fairly large degree of concurrency with the parallel exhaustive search, a large search tree may still result in many more tasks at one time than the number of processors available. We are still doing a lot of work that would not need to be done using alpha-beta pruning. Also, a bonus would be to achieve an algorithm that is not too dependent on the number of processors available, i.e. we would like an algorithm that performs reasonably well on one processor or many.

```

1. (define (parallel-exhaustive bottom)
   'A full concurrent search.'

2. (define (maximize game level)

3.   (define (max1 moves best)
4.     (if (no-move? moves)
5.         best
6.         (let* ((move (car moves))
7.                (new-value (future
8.                             (minus
9.                              (choice-value
10.                               (maximize (make-child game move)
11.                                         (add1 level)))))))
12.           (max1 (cdr moves)
13.                 (future (if (> new-value (choice-value best))
14.                           (make-choice move new-value)
15.                           best))))))

16.   (cond ((game-end? game)
17.          (make-choice (null-square) (end-eval game)))
18.         (= level bottom)
19.         (make-choice (null-square) (static-eval game))
20.         (t
21.          (let ((moves (legal-moves game)))
22.            (if (no-move? moves)
23.                (make-choice (null-square)
24.                              (future (minus
25.                                       (choice-value
26.                                        (maximize (pass game)
27.                                                  (add1 level))))))
28.                (max1 moves (make-choice (null-square) (neg-inf))))))
29.          (choice-move (maximize *game* (level-0))))

```

Figure 3-1: Exhaustive search with futures added.

By this criterion, we would like to have an algorithm that performs at least as well as alpha-beta, and the parallel exhaustive search performs no better than the sequential exhaustive minmax search on one processor. The natural choice to reducing work is to add concurrency to the alpha-beta procedure.

Adding parallelism to the alpha-beta routine is not as straightforward, however, as parallelizing an exhaustive search, since the decision of whether to cut off a search path depends on the results of previously evaluated search paths. So the alpha-beta algorithm as presented in Chapter Two contains an inherent sequentiality.

A conservative approach to parallelizing the alpha-beta procedure would be to process in parallel all nodes that would definitely be needed during the search, even in the case of maximum cutoff. In other words, we can evaluate all branches that would not be cut off under any circumstances in parallel, and then process the remaining branches sequentially, checking for cutoff.

To implement this, we must find a general method for determining which nodes need to be processed before we start checking for cutoff. We can make the following two claims:

**Theorem 1** *When the negabeta is applied to node  $v$ , if  $\beta = \infty$  then  $v$  will not be cut off.*

*Proof:* The condition for cutoff of node  $v$  is  $\alpha \geq \beta$ . The value of  $\beta$  never changes and the condition  $\alpha = \infty$  will never occur, because for every node  $w$  that is not a terminal node,  $\alpha = -\infty$  or  $\alpha$  is the negation of the value returned by applying negabeta to a child of  $w$ . For a terminal node  $x$ ,  $\alpha = s$ , where  $s$  is the static evaluation of  $w$  and  $-\infty < s < \infty$ . Thus, for every node  $w$ ,  $\alpha \neq \infty$ .

**Theorem 2** *If negabeta is being applied to node  $v$  of a search tree, then at least one child of  $v$  will be evaluated before  $v$  can be cutoff.*

*Proof:* Like Theorem One, the second theorem can be shown to be true by examining the condition for cutoff,  $\alpha \geq \beta$ . If node  $v$  were to be cutoff before



evaluating any of its children, then that would mean initially  $\alpha \geq \beta$ . Then  $-\beta \geq -\alpha$ . However, if this were true, then the parent node of  $v$  would have reached its cutoff condition and would not have applied the alpha-beta procedure to  $v$ . Thus, by contradiction, Theorem Two is proven true.

We can make one further claim:

**Theorem 3** *If, while negabeta is operating on node  $v$ ,  $\beta < \infty$ , then  $v$  can be cutoff after the evaluation of any child.*

Proof: Once again, the condition for cutoff is  $\alpha \geq \beta$ .  $\alpha = -\infty$  or  $\alpha = s$  where  $s$  is a static value. We define a static value to be any number  $s$  such that  $-\infty < s < \infty$ . Thus, for any  $\beta = c$ ,  $c < \infty$ , we can envision an  $s$ , such that  $s = c + 1$ .

To summarize, if negabeta is being applied to node  $v$ , two conditions exist:

1.  $\beta = \infty$ , in which case node  $v$  cannot be cutoff, and
2.  $\beta < \infty$ , in which case node  $v$  can be cutoff after one child of  $v$  has been evaluated, but only after one child has been evaluated.

We can combine the parallel exhaustive search and the alpha-beta algorithm to take advantage of these properties:

#### MANDATORY WORK FIRST:

1. If the level is the top level, let  $\alpha = -\infty$  and let  $\beta = \infty$ .
2. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
3. If  $\beta = \infty$  then apply MANDATORY WORK FIRST to all children in parallel. Report the highest value.
4. Otherwise, until all children are examined with MANDATORY WORK FIRST or  $\alpha \geq \beta$ :



- (a) Set  $\alpha$  to the larger of the given  $\alpha$  values and the maximum of the negation of the values so far reported by MINIMAX working on the children.
- (b) Use MINIMAX on the next child of the current position, handing this new application of MINIMAX  $-\beta$  as its  $\alpha$  and  $-\alpha$  as its  $\beta$ .

The modified Multello procedure (Figure 3-2) exploits these conditions for cutoff by including a check for the impossible-cutoff condition,  $\beta = \infty$ , in the alpha-beta routine of Figure 2-3. If  $\beta = \infty$ , we want to process all the children concurrently in the manner of our parallel exhaustive search. If  $\beta < \infty$  then we will go in sequence checking for cutoff before processing each child in the manner of our sequential alpha-beta algorithm. Note that we must place the check *after* Lines ?? rather than before so we can take advantage of Lemma 1.

This approach maximizes cutoff and consequently minimizes the total computational work. Aside from the relatively small overhead generated by checking for the impossible-cutoff condition and creating the appropriate futures, the total work should be the same as that in the sequential alpha-beta algorithm. Thus, we should expect performance at least somewhat better than the sequential alpha-beta algorithm.

From this method of parallelizing alpha-beta search, we can expect a large burst of parallel activity in the earlier stages of the search, when a portion of the search tree can be done in parallel, with a fairly steep drop down to a few processors as the remaining branches are generated and checked for cutoff.

### 3.1.3 Speculative Search

Thus far, we have presented two extreme approaches in parallelizing adversary searches. The parallel exhaustive search of Figure 3-1 attempts to maximally utilize

```

1. (define (work-first bottom
2.           &optional (alpha (neg-inf)) (beta (inf)))

3. (define (maximize game level alpha beta)

4.   (define (max1 moves best)
5.     (if (no-move? moves)
6.         best
7.         (let ((move (car moves))
8.               (new-value (future (minus
9.                                   (choice-value
10.                                  (maximize (make-child game move) (add1 level)
11.                                             (future (minus beta))
12.                                             (future (minus (choice-value best))))))))))
13.         (cond ((= beta (inf))
14.                (max1 (cdr moves)
15.                       (future
16.                        (if (> new-value (choice-value best))
17.                            (make-choice move new-value)
18.                            best))))
19.                ((>= new-value beta)
20.                 (cutoff-choice))
21.                (t
22.                 (max1 (cdr moves)
23.                        (if (> new-value (choice-value best))
24.                            (make-choice move new-value)
25.                            best))))))
26.   (cond ((game-end? game)
27.          (make-choice (null-square) (end-eval game)))
28.         ((= level bottom)
29.          (make-choice (null-square) (static-eval game)))
30.         (t
31.          (let ((new-moves (legal-moves game)))
32.              (if (no-move? new-moves)
33.                  (make-choice (null-square)
34.                                 (future (minus (choice-value
35.                                                  (maximize (pass game) (add1 level)
36.                                                       (future (minus beta))
37.                                                       (future (minus alpha))))))
38.                  (max1 new-moves
39.                         (make-choice (null-square) alpha))))))

40. (choice-move (maximize *game* (level-0) alpha beta)))

```

Figure 3-2: mandatory-work first procedure

processors at the expense of doing redundant work, since it does no pruning. The mandatory work first algorithm of Figure 3-2 takes the opposite tack and initially does in parallel only the work that must be done in any case, and then processes the rest of the search tree in sequence to maximize cutoff, thereby minimizing total computational work done.

A compromise between these two tactics might bring about better performance. Perhaps an alpha-beta search combined with a more aggressive use of parallelism could result in more efficient use of processors without doing such redundant work to extend the final search time. Such an approach would expand as many branches as there are available processors and, any time an alpha-beta cutoff condition occurs, terminate any processes that are deemed irrelevant by the cutoff. This spawning of tasks that may not be needed is called *speculative computation*.

Thus, we want to have something like the full search algorithm, in which we set up a future for every child, except that we want the option of terminating these tasks if any one of them results in cutoff of this node.

To SPECULATE:

1. If the level is the top level, let  $\alpha = -\infty$  and let  $\beta = \infty$ .
  2. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
  3. Apply SPECULATE to all children in parallel, passing them  $-\beta$  as their  $\alpha$  and  $-\alpha$  as their  $\beta$ .
    - (a) Each time a child returns a value, set  $\alpha$  to the larger of the current value and the negation of the value returned by the child.
    - (b) If  $\alpha \geq \beta$  terminate the remaining tasks.
- Report  $\alpha$ .



The only way for tasks to communicate with each other in Multilisp is through shared memory. The information to be shared includes the alpha-beta window and the best move chosen so far (the value corresponding to the best move is contained in the alpha-beta window). To enable sharing of this information, the parent task must pass the shared data structure to its children. Figure 3-3 lists some operations on such a data structure. <sup>1</sup>

This algorithm requires some explicit mechanism for terminating a task. However, there is no such *kill* mechanism existing in the current implementation of Multilisp. There is an available *determine* primitive that determines a future, but it can only be used on an undetermined future, and we cannot guarantee at any moment in time whether a future has just recently been determined.

Conveniently, the alpha-beta cutoff condition provides a way to signal a task to terminate. By requiring a task has to continually check whether its node is cutoff, we can send a kill message by setting its alpha-beta window to a cutoff condition. Consequently, we construct a pseudo-kill operator that sets  $\alpha \leftarrow \infty$  (Figure 3-4). <sup>2</sup>

Our pseudo-kill mechanism would be to have each node check for its own cutoff as well as its children. In either case of cutoff, then the process can merely signal, via shared data structures, to its spawned tasks that they are no longer required. This can be done by altering the alpha-beta window so it is cut off. Thus the children, in turn, when they execute, will notice there are cutoff and then will signal to their children to terminate. And thus we have a recursive *kill* construct, which will resolve a future and kill its task and all its recursively spawned tasks.

Another consideration is the possible effect of updating shared data. It is possible that one child may check the value of its alpha-beta window, decide that it has a better value, and then update the window. In the meantime, however, another parallel task may have decided to update the same window and done so between the

---

<sup>1</sup>Macros are used in Multello extensively to achieve reasonable search times.

<sup>2</sup>*letrec* is a Scheme construct that allows mutually recursive definitions.



1. (defmacro make-window (choice beta)  
    ‘‘Creates a shared alpha-beta window (choice is alpha  
      consed with best move found so far.’’
2.   ‘(cons ,choice ,beta))
  
3. (defmacro window-choice (window)  
    ‘‘Returns alpha and best move of alpha-beta window.’’
4.   ‘(car ,window))
  
5. (defmacro window-alpha (window)  
    ‘‘Returns alpha of alpha-beta window.’’
6.   ‘(cdar ,window))
  
7. (defmacro window-beta (window)  
    ‘‘Returns beta of alpha-beta window.’’
8.   ‘(cdr ,window))
  
9. (defmacro set-alpha (window value)  
    ‘‘Atomically sets alpha of alpha-beta window to value.  
      Returns old alpha.’’
10.  ‘(scdr (car ,window) ,value))
  
11. (defmacro set-beta (window value)  
    ‘‘Atomically sets beta of alpha-beta window to value.  
      Returns old beta.’’
12.  ‘(scdr ,window ,value))
  
13. (defmacro set-choice (window choice)  
    ‘‘Atomically sets alpha and best move of alpha-beta window  
      to choice. Returns old alpha and best move.’’
14.  ‘(scar ,window ,choice))
  
15. (defmacro cutoff? (window)  
    ‘‘Returns t if alpha is greater than or equal to beta.’’
16.  ‘(>= (cdar ,window) (cdr ,window)))

Figure 3-3: Operations on shared alpha-beta window for speculative search

1. (define (kill window)  
    ‘‘Signals to tasks that update this window to kill themselves.  
    Does this by setting cutoff condition, alpha >= beta.’’
2. (set-choice window (cutoff-choice))
3. (set-beta window (neg-inf)))

Figure 3-4: An *ad hoc* task-killing mechanism for speculative search

1. (defmacro maximize-members (game list level window)  
    ‘‘Spawns parallel tasks to evaluate the  
    children of a node.’’
2. (mapcar (lambda (move)
3. (future (maximize (make-child ,game ,move)
4. ,level ,window ,move))
5. ,list))

Figure 3-5: The expression that spawns off speculative tasks to evaluate a node's children.

```

1. (define (update-choice window value move)
   "Atomically updates window."

2.   (letrec ((old-choice (delay (set-choice window new-choice)))
3.            (new-choice (delay
4.                          (if (> value (choice-value old-choice))
5.                              (make-choice move value)
6.                              old-choice))))
7.     (touch old-choice)
8.     (touch new-choice)))

```

Figure 3-6: Procedure for atomically updating a shared alpha-beta window.

time the former child examined the window and the time it changed it. While this is a general problem in coordinating actions on shared data, in this case the result may be an inaccurate coupling of the best move found so far, and its associated value. Another problem may be that the best move found so far, plus its associated value, get erased by the more tardy child. This case, in addition to possibly resulting in a wrongly chosen move, could reduce chances for cutoff.

Once the parallel tasks for evaluating the children of a node are set up (Line 34), it is not sufficient for the procedure to loop around and wait for a cutoff to happen or for all the children to become determine. Since the scheduling of tasks is indeterminate, if the futures are not touched, there is no guarantee that the futures will terminate as long as the parent task does not touch them. It is conceivable that all of the processors could be usurped by similar processes, all waiting for their children to terminate or cause a cutoff condition, neither of which will happen as long as they are no processors available to process them.

Consequently, it is necessary to add some guarantee that the procedure will terminate, by explicitly touching each future to make sure they eventually resolve (Line 8 and Line 36). We can formalize this by saying that for a program to be correct, each future must be touched, either implicitly through a strict operator,

or explicitly by *touch*. The resulting procedure is shown in Figure 3-7.

Note that in the procedure of Figure 3-7, a node updates the alpha-beta window of its parents after all its children have run to completion; in other words, one update is made, and it is the correct one. An alternative is to increase cutoff possibilities by updating the parent's alpha-beta values after each change in the node's own alpha-beta values.

To SPECULATE with more frequent communication:

1. If the level is the top level, let  $\alpha = -\infty$  and let  $\beta = \infty$ .
2. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
3. Apply SPECULATE to all children in parallel, passing them  $-\beta$  as their  $\alpha$  and  $-\alpha$  as their  $\beta$ .
  - (a) Each time a child returns a value, set  $\alpha$  to the larger of the current value and the negation of the value returned by the child. Report  $\alpha$  but do not quit.
  - (b) If  $\alpha \geq \beta$  terminate the remaining tasks and quit.

Figure 3-8 shows a variation of the listing in Figure 3-7 that does an update after each child has been touched, thereby increasing cutoff possibilities for other nodes at the expense of doing more work.

It is not clear which version would result in better performance; ideally we would like a system in which comparisons and updates are triggered only by updates from child nodes. Such a mechanism may be more feasible in a message-passing model of computation [2]. Differences in performance between the procedures in Figure 3-7 and Figure 3-8 may indicate the extent of the overhead caused by locking of data during the updates as compared to the importance of increasing cutoff possibilities. Note that the two variations presented thus far are only two of many possibilities;



```

1. (define (work-more bottom
2.           &optional (alpha (neg-inf)) (beta (inf)))

3. (define (maximize game level window this-move)

4.   (define (max1 children new-window)
5.     (cond ((cutoff? window)
6.            (kill new-window))
7.           ((null children)
8.            (update-choice window
9.              (minus (window-alpha new-window)) this-move))
10.          (t
11.            (touch (car children))
12.            (max1 (cdr children) new-window))))

14.  (cond ((game-end? game)
15.         (update-choice window (end-eval game) this-move))
16.        ((= level bottom)
17.         (update-choice window (static-eval game) this-move)
18.         (t
19.          (let ((new-moves (legal-moves game))
20.                (new-window (make-window
21.                              (make-choice (null-square)
22.                                             (minus (window-beta window)))
23.                                             (minus (window-alpha window))))))
24.            (cond ((no-move? new-moves)
25.                   (maximize game (add1 level) new-window)
26.                   (update-choice window
27.                     (minus (window-alpha new-window)) this-move))
28.                  (t
29.                   (max1 (maximize-members game new-moves
30.                               (add1 level) new-window)
31.                           new-window))))))
32.  (let* ((root-window
33.          (make-window (make-choice (null-square) alpha) beta))
34.         (moves (legal-moves *game*)))
35.    (children (maximize-members *game* moves (level-1)
36.                               root-window)))
37.  (touch-members children)
38.  (choice-move (window-choice root-window))))

```

Figure 3-7: A speculative alpha-beta search.

```

1. (define (work-more1 bottom
2.           &optional (alpha (neg-inf)) (beta (inf)))

3. (define (maximize game level window this-move)

4.   (define (max1 children new-window)
5.     (cond ((cutoff? window)
6.           (kill new-window))
7.           ((not (null children))
8.            (touch (car children))
9.            (update-choice window
10.             (minus (window-alpha new-window)
11.                    this-move))
12.            (max1 (cdr children))))

13. (cond ((game-end? game)
14.        (update-choice window (end-eval game) this-move))
15.        ((= level bottom)
16.         (update-choice window (static-eval game) this-move)
17.         (t
18.          (let ((new-moves (legal-moves game))
19.                (new-window (make-window
20.                              (make-choice (null-square)
21.                                             (minus (window-beta window)))
22.                                             (minus (window-alpha window))))))
23.            (cond ((no-move? new-moves)
24.                   (maximize game (add1 level) new-window)
25.                   (update-choice window
26.                    (minus (window-alpha new-window)) this-move))
27.                  (t
28.                   (max1 (maximize-members game new-moves
29.                                             (add1 level) new-window)
30.                          new-window))))))
31. (let* ((root-window
32.         (make-window (make-choice (null-square) alpha) beta))
33.        (moves (legal-moves *game*))
34.        (children (maximize-members *game* moves (level-1)
35.                                     root-window)))
36. (touch-members children)
37. (choice-move (window-choice root-window)))

```

Figure 3-8: Tree-splitting using more frequent updating

we could have a task update its window twice for every child, or once for every other child, and so on.

Now, we run into another implementation decision. In addition to varying the frequency of communication between tasks, we might want to specify how these tasks are scheduled. As it now stands, the invocation of *future* places the current task on the task queue and begins execution of the newly-created task.

Thus, in our new alpha-beta procedure, the evaluation of the children of the node have priority over the loop that checks for cutoff. This may not be so undesirable except in the case that the cutoff-checking loop is placed on the task queue. Then there is no possible cutoff and consequent saving of work.

On the other hand, if we had more control over the scheduling of tasks, we could ensure that the cutoff-checking loop has higher priority than its children. And then we might want to specify that some of the children have higher priority than others, ensuring that some of the children are evaluated first and return relatively soon, increasing cutoff possibilities for the other children. Discovering that a node is cutoff would be a moot point if most of the subtrees had already been evaluated (Figure 3-9). The processor time would have been better spent on other separate branches.

So, in general, we might want to give tasks higher in the search tree a higher scheduling priority, and we might want to give the children of a node an increasing or decreasing left-to-right priority. In other words, we would like to be able to create an explicit *priority ordering*, with perhaps tasks to be killed at zero and mandatory tasks given a priority of infinity, or vica versa.

Unfortunately, Multilisp provides no such explicit task scheduling. We can, however, explore in a limited fashion the effects of different scheduling semantics using a version of *future* called *dfuture*.<sup>3</sup> This construct is equivalent to *future* with the exception that *dfuture* causes the newly-created task to placed on the idle

---

<sup>3</sup>see Appendix D for a description of Multilisp expressions.



Figure 3-9: An example of very suboptimal scheduling

```

1. (defmacro dmaximize-members (game list level window)
   ' 'An alternative to MAXIMIZE-MEMBERS. ' '

2.   '(mapcar (lambda (move)
3.             (dfuture (maximize (make-child ,game ,move)
4.                               ,level ,window ,move)))
5.             ,list))

```

Figure 3-10: Using *dfuture* in “speculative” search.

task queue, while the current task continues executing. The appropriately modified version of *maximize-members* is shown in Figure 3-10. Another possible variation is to use *sfuture*, which provides for FIFO task queueing.

From our speculative search algorithm, we should expect fairly large utilization of processors similar to that achieved in the parallel exhaustive search. However, the use of cutoff should decrease the total work involved, so a total computation time less than the mandatory work first algorithm should be possible. Factors that might hinder such a result include overhead introduced by the *ad hoc* kill construct (tasks may not be garbage-collected properly), overhead from the data locking introduced



by the atomicity requirements of checking and updating shared data, and inefficient scheduling of tasks.

```

1. (define *aspire-search* alpha-beta)
2. (define *aspire-num* 33)

3. (define (aspiration game bottom
4.         &optional (search *aspire-search*)
5.         (number *aspire-num*))
   '‘An alpha-beta search.’’

6. (let ((window (/ (- (inf) (neg-inf)) number)))

7.   (define (aspire alpha best)
8.     (if (>= (+ alpha window) (inf))
9.         best
10.        (aspire (+ alpha window)
11.                (future
12.                  (or
13.                    (search game bottom alpha (+ alpha window))
14.                    best))))))

15. (aspire (neg-inf) (null-square))))

```

Figure 3-11: parallel aspiration

### 3.1.4 Parallel-Aspiration Search

A strikingly different approach suggested by Baudet [6] is to divide the initial alpha-beta window,  $[-\infty, \infty]$  into smaller windows and run several full alpha-beta searches, each on a different processor, with these smaller windows. The search that returns a move with the highest value is the correct one. Such a *parallel aspiration search* is shown in Figure 3-11.

Since each processor is running one full alpha-beta search with a window smaller than the original, each of these searches should take less time than the original search would have. The attractiveness of this proposal lies in the fact that no communication is necessary between the narrow-window alpha-beta searches, so no time will be lost to locking of data, and since little data is shared between processes

(and even this is rarely accessed during the duration of the search) there should be less time lost to bus contention, especially if good advantage is made of locality of data.

Baudet [6] experienced more than  $k$ -fold speedup with  $k$  processors, for  $k \leq 3$ . He concludes from this that the alpha-beta algorithm is not optimal. His results show no significant gain in decreased computation time for  $k > 5$ .

Possible variations of this algorithm include using some communication between processes, say a flag, to signal tasks to stop if the best move has already been found, or to further alter their alpha-beta windows. Another possibility is to combine this method with other parallel versions of alpha-beta search, e.g. mandatory work first or one of the speculative approaches.

A drawback to this method of parallelizing alpha-beta search is its dependence on the knowledge of the number of processors available. Moreover, at least in Multilisp, this method requires explicit allocation of tasks to processors to efficiently utilize the available processors. If  $k$  processors are available, using less than  $k$  processors will not fully take advantage of potential parallelism. Using more than  $k$  processors will cause at least one processor to execute more than one full alpha-beta search.

## 3.2 Finer Grain Parallelism

### 3.2.1 Static Evaluation

In addition to parallelizing the actual search algorithm, we can perhaps exploit parallelism at a finer level. Static evaluation, one of the more time-consuming operations of the search, is a good candidate for such further concurrency, since the Multello evaluation technique involves examination of each square on the game board. By the same token, probably most board games could benefit from parallelization at the static evaluation level, since most likely a static evaluation will involve examination of most, if not all, portions of the game board.

Looking at the static evaluation procedures *EVALUATE* and *FEVALUATE*, we can see that simply wrapping a future around the addition operands will not be very effective, since the first operand, a call to *EVAL-SQUARE*, will take little time, while the second operand depends on the evaluation of the remainder of the board. The problem is that the tree of additions is extremely lopsided. To achieve more concurrency, we would like to see a more balanced tree. We can achieve this by using a divide-and-conquer approach, splitting the list *\*SQUARES\** into halves and splitting those halves into halves, and so on. The result is a balanced binary tree six deep, instead of the sixty-four ply tree created in the sequential version.

The static evaluation for Multello in both continuing and end-of-game positions involves summing the values of each square on the game board. Thus, we can parallelize this by wrapping a *future* around each operand of the addition operation.<sup>4</sup> In this example, the *future* is actually enclosed by the body of the tail-recursive loop, so the procedure returns a value, albeit undetermined, very quickly.

A possible refinement of this binary tree form of parallelization would be to stop using futures at a certain depth, say, when the board has been broken down to divisions of four squares. At this point, the cost of using a future begins to rival

---

<sup>4</sup>This is equivalent to using *pcall*. See Appendix B.



```
1. (define (evaluate board color)
    "A parallel static evaluation.")

2. (define (eval squares)
    "This sums up the values of all squares on the board
    using divide and conquer.")

3. (future
4.   (cond ((null squares)
5.         0)
6.         ((square? squares)
7.          (eval-square board squares color))
8.         (t
9.          (let ((right (nthcdr
10.                 (round (/ (length squares) 2)) squares))
11.                (left (ldiff right squares)))
12.              (+ (eval left)
13.                 (eval right)))))))

14. (eval *squares*))
```

Figure 3-12: A parallel static evaluation procedure

```
1. (define (final-eval board color)
   'A parallel static evaluation.')

2. (define (eval squares)
   'This sums up the values of all squares on the board
   using divide and conquer.')

3. (future
4.   (cond ((null squares)
5.         0)
6.         ((square? squares)
7.          (final-eval-square board squares color))
8.         (t
9.          (let ((right (nthcdr
10.                 (round (/ (length squares) 2)) squares))
11.                (left (ldiff right squares)))
12.                (+ (eval left)
13.                   (eval right)))))))

14. (eval *squares*))
```

Figure 3-13: A parallel static evaluation procedure for end of game.

the cost of sequentially evaluation the subdivisions.

### 3.2.2 Flipping Pieces

Another area in which parallelism may be exploited is in the the routines that actually perform a move. Games such as chess or tic-tac-toe which involve only simple changes to the game board probably cannot employ much concurrency, but games in which moves more drastically alter the board promise opportunities for more significant parallelism.

As we can see from the sequential implementation of the move generation algorithm in Multello, making a move involves changing the board in up to eight possible directions, each of which does not affect the other. Thus, it seems we can proceed with the eight flipping calls concurrently and then summarize the results at the end of the eight tries.

Once again, we must take care in setting up the separate parts of the move enaction in parallel tasks, for this is a side-effecting operation. Although it does not matter what order the differently directed flip operations are done, it is crucial that they be done before any other operations that depend on the state of the game board.

For example, if we call the flip procedure and then perform a static evaluation on the board before all the flips have completed, then an erroneous static value will result. Worse yet, if a copy of the current board is made, say in generating a successor position during a look-ahead search, before the flips have completed, then we will have created a non-possible game board, and the search will be worthless.

Consequently, if we create a future for each component of a move, then we must touch each of these futures at some point to guarantee that the move has been completed.

Figure 3-14 shows the *DO-FLIPS* procedure modified to process the eight parts of the flipping procedure, plus the actual placement of the new piece, concurrently.

```

1. (define (fdo-flips board square color)
    'Parallel version of do-flips.')

2. (let ((futures (list
3.     (future (flip1 board square (east) color))
4.     (future (flip1 board square (ne) color))
5.     (future (flip1 board square (north) color))
6.     (future (flip1 board square (nw) color))
7.     (future (flip1 board square (west) color))
8.     (future (flip1 board square (sw) color))
9.     (future (flip1 board square (south) color))
10.    (future (flip1 board square (se) color))
11.    (future (put-piece color board square))))))
12. (touch-members futures)))

```

Figure 3-14: Parallel version of move procedure

The futures created are combined into a list and a list version of *touch* is used to guarantee the completion of the move on exit from the procedure.

The same idea can be used to parallelize the procedure that updates the list of possibly legal moves after a move has been done. Like the flipping done during a move, all eight directions along the board must be checked, only in this case to see if it is empty and thus can be added to the list representing the perimeter of empty squares on the board. The resulting procedure is shown in Figure 3-15.

### 3.2.3 Move Generation

In addition to parallelizing the the enactment of moves on the game board, we can also parallelize the operation of generating the list of possible moves on a given turn. The parallelized filtering procedure is shown in Figure 3-16.

In generating the children of a node, it is usually necessary to filter out many of the possible moves as illegal. Even in Multello, which narrows down the list of possible moves to the perimeter surrounding the contiguous form of pieces on the



```

1. (define (fupdate board move-list square)
   "Parallel version of update."

2. (define (add square move-list)
3.   ""
4.   (if (and (not (null square))
5.           (empty? board square)
6.           (not (member square move-list)))
7.       (insert square move-list)))

8. (let* ((new-list (remove square (copy move-list)))
9.        (futures (list
10.                 (future (add (inc-square square (east)) new-list))
11.                 (future (add (inc-square square (ne)) new-list))
12.                 (future (add (inc-square square (north)) new-list))
13.                 (future (add (inc-square square (nw)) new-list))
14.                 (future (add (inc-square square (west)) new-list))
15.                 (future (add (inc-square square (sw)) new-list))
16.                 (future (add (inc-square square (south)) new-list))
17.                 (future (add (inc-square square (se)) new-list)))))
18.   (touch-members futures)
19.   new-list))

```

Figure 3-15: Parallel updating of the border.

```

1. (define (get-moves board moves color)
   "Returns a list of all possible moves with color in MOVES."

2. (future
3.   (cond ((no-move? moves)
4.          (null-square))
5.         ((flip-possible? board (car moves) color)
6.          (cons (car moves)
7.                (get-moves board (cdr moves) color))))
8.   (t
9.    (fget-moves board (cdr moves) color))))

```

Figure 3-16: parallel move generation

board, each member of the list has to be checked if it is a valid move first before it is used to generate a child.

Checking the legality of a move is trivial in games like tic-tac-toe or Go. For these games, verifying the legality of a move involves merely a check to whether the spot to be moved upon is empty.<sup>5</sup>

Othello and chess, on the other hand, provide more opportunity for parallel activity in verification of move legality. In chess, one criterion for a move to be legal involves making sure that the move does not place the current player's king in check. This would, in effect, involve a full search of the possible moves for the opposing player to determine if he could capture the king.

Othello, to a much lesser extent, provides possibilities for parallelizing move-legality verification in the same manner that we added parallelism to the move-generation procedure. Instead of flipping pieces when possible in eight different directions, here we merely check if a flip is possible in each of the eight directions and return true if any of them do (Figure ??). Like the move-generation procedure, we can do these eight checks concurrently, although *or* is strict, so we must first accumulate the *future*'ed lists in a list and then iterate through it until a non-null list is found. The overhead incurred by this might not justify the attempt to parallelize this procedure.

---

<sup>5</sup> Actually, in Go, a check also must be made to ensure that the current move does not exactly reverse the effect of a previous move.

```
1. (define (fflip-possible? board square color)
   "Parallel version of flip-possible?."

2.   (define (or-members list)
3.     (cond ((null list)
4.           nil
5.           ((not (null (car list)))
6.            t)
7.           ((or-members (cdr list))))))

8. (let ((or-list
9.       (list
10.        (flip board square (east) color)
11.        (flip board square (ne) color)
12.        (flip board square (north) color)
13.        (flip board square (nw) color)
14.        (flip board square (west) color)
15.        (flip board square (sw) color)
16.        (flip board square (south) color)
17.        (flip board square (se) color))))
18. (or-members or-list))
```

Figure 3-17: parallel move verification procedrue

??

# Chapter 4

## Results

### 4.1 Gathering Data

#### 4.1.1 A Test Case

We use as a test case the opening position of an Othello game (Figure 2-1). From the symmetry of this position, we can expect a fairly large number of alpha-beta pruning opportunities in a search tree of moderate to large depth.

Also, the small number of pieces on the board results in an initially low branch factor. The opening position allows four possible moves, so the root node will have four children. The branch factor overall should grow as the search tree gets deeper into the game and evaluates positions with more pieces on the board.

#### 4.1.2 Parallelism Profiles

Several different forms of data are desired in analyzing a test run. These include the number of processors actively running tasks at any one time, overhead created from the invocation of future, time lost to bus and memory contention, and, of course, final computation time. This data can be analyzed in the form of a *parallelism profile*. An example of such a timeline is shown in (Figure 4-1) The profile is a graph



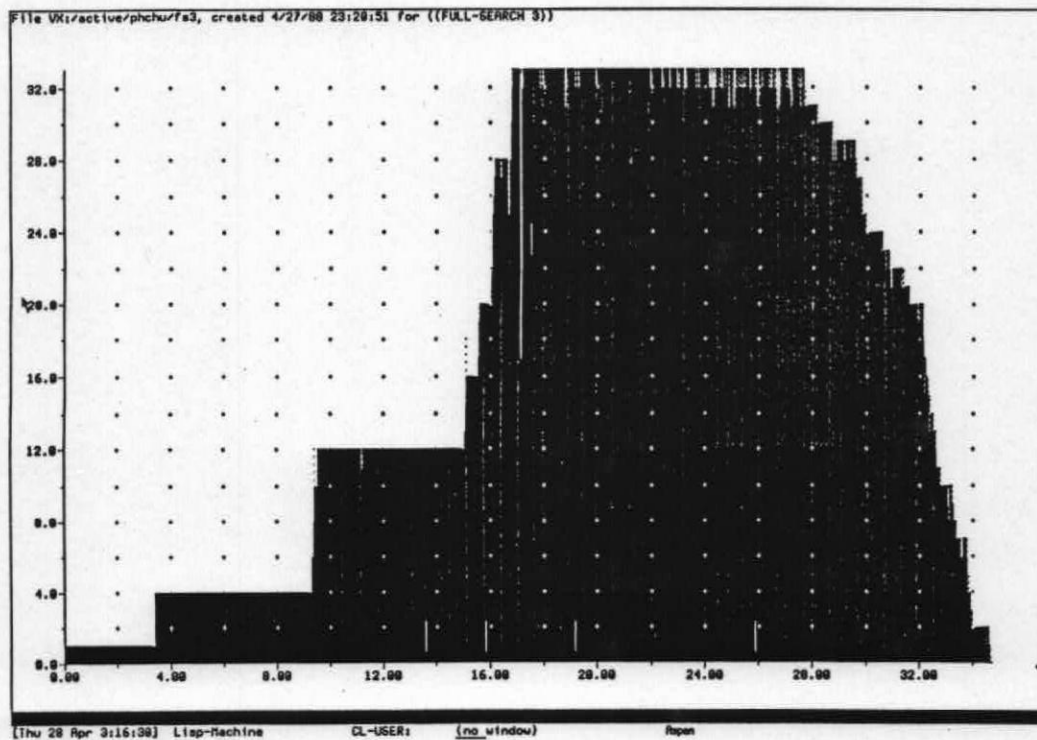


Figure 4-1: A parallelism profile for a three-ply depth-first search.

of number of processors in use vs. time. The area filled in black is the total “real” computation time. The gray-filled area is processor time spent on the overhead required by the handling of futures. Time lost to locking of data is represented by the most sparsely-filled area, and idle processor time is the remaining black area.

### 4.1.3 Sequential Search

The presence of alpha-beta cutoff is noticeable starting at ply three (Figure 4.1). The savings in search due to the cutoff is not as much as we might expect, possibly because the static evaluations are relatively quick compared to the process of generating successor nodes, which entails generating legal moves and making these moves on copies of a board.

The two searches perform similarly shallow searches, but on four-ply and deeper searches the alpha-beta cutoffs make a dramatic difference. However, despite the

search	two-ply	three-ply	four-ply
depth-first	19.67s	77.45s	250.98s
alpha-beta	19.55s	64.78s	225.78s

Table 4.1: Comparison of exhaustive minmax and alpha-beta search times.

	level 1	level 2	level 3	level 4
final time	7.98s	15.02s	34.72s	133.57s
idle time	92.1%	82.1%	45.2%	15.5%
compute time	7.8%	17.8%	54.3%	83.7%
future time	0.1%	0.1%	0.4%	0.6%
overhead time	0.0%	0.0%	0.1%	0.2%

Table 4.2: Statistics for parallel exhaustive search

large savings resulting from alpha-beta, the exponential growth of both algorithms is apparent.

#### 4.1.4 Parallel Exhaustive Search

The parallel exhaustive search in contrast does quite well time-wise. (Figure 4.2) Extra work incurred by the overhead of using futures and data locking is close to negligible. The full search makes fairly efficient use of available processors. (Figure 4-2) As the search grows deeper, the utilization grows more efficient, its parallelism profile begins to resemble a rectangle.

The staircase-like effect on the left portion of the graph probably corresponds to the different search levels, at each level of the search tree the tasks created to evaluate the children of that level's nodes pile up. Supporting this conjecture is the fact that each "step" is higher than the previous, and the first step consists of a four-processor rise, corresponding to the number of children at ply one.

The horizontal span of these steps probably represent the time taken to filter out the list of legal moves from which the children nodes are made. The noticeable width of these steps relative suggest that time-saving opportunities might lie in parallelizing the *GET-MOVES* procedure.



Of interesting note is the apparent "tail" of the graph in Figure 4-2, where, as the computation nears its end, the graph begins to drop and then rises again, briefly, before tapering off, again. This might signify the presence of a number of tasks that could not be processed before certain computations, prohibited by the saturation of the processors, occurred.

For example, one might envision a search of a five-ply game tree in which the processors are saturated by a number of static evaluations while some tasks to process nodes on ply four are waiting in the task queue. Once some of the processors are freed, there is some delay while the newly activated ply four tasks compute their lists of legal moves from which they generate their successor nodes. This delay, plus the subsequent activation of the successor nodes could account for such a dip in the graph, although the accumulated tasks in question may not be the ones cited in this scenario.

Once again, the parallelization of the *legal-moves* function holds some prospect of shortening such delays between the activation of a node's and the spawning of its children tasks. Another way to avoid this bottleneck might lie in the provision of more control over the scheduling of tasks.

#### 4.1.5 Using Mandatory Work-First

As expected, the mandatory work first procedure does better than its sequential alpha-beta counterpart. The reduced computational load in this parallel alpha-beta search, however, does not enable it to achieve search times as good as the parallel exhaustive search. (Figure 4.3)

The profile of the mandatory work first algorithm applied to a five-ply game tree (Figure 4-3) shows a large burst of parallelism near the beginning of the search which rapidly tapers off to one processor. This burst grows increasingly sharper and narrower relative to the entire search graph as the search grows deeper.

These results show the parallel exhaustive search to be far superior to the manda-



	level 1	level 2	level 3	level 4
final time	7.88s	27.62s	116.32s	460.5s
idle time	92.1%	92.5%	91.8%	92.8%
compute time	7.8%	7.4%	8.1%	7.1%
future time	0.1%	0.1%	0.1%	0.0%
overhead time	0.0%	0.0%	0.0%	0.0%

Table 4.3: Statistics for mandatory work first search

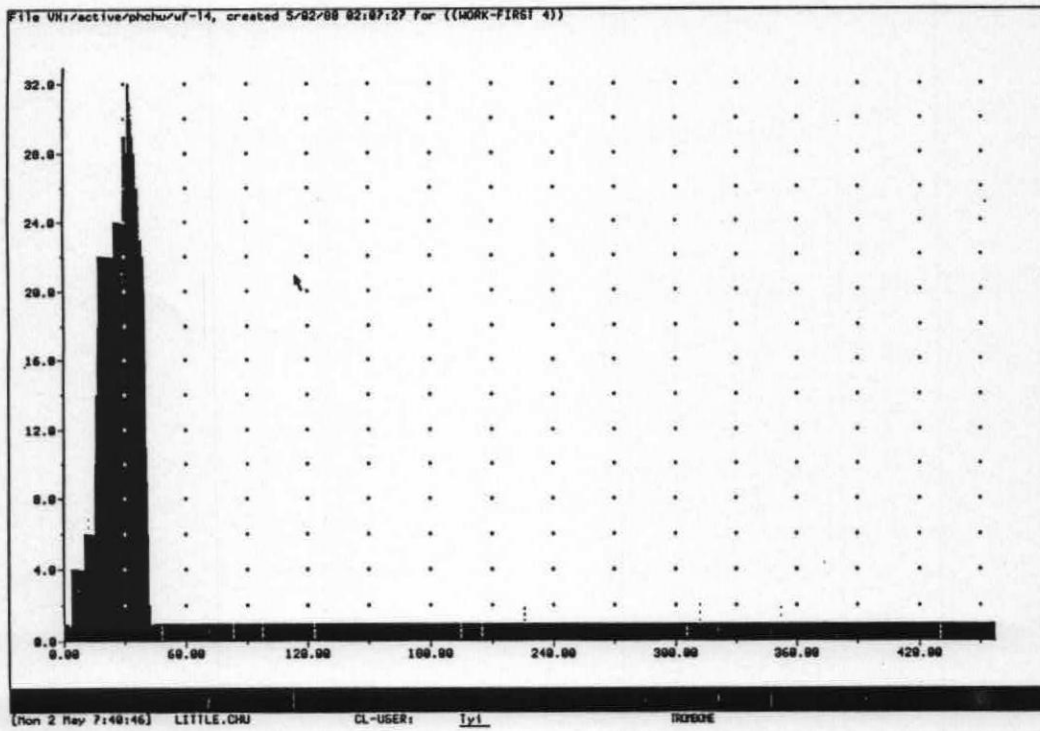


Figure 4.3: Profile of the Mandatory-Work-First procedure

search	two-ply	three-ply	four-ply	five-ply
final time	6.48s	14.73s	34.28s	129.38s
idle time	90.8%	79.8%	42.9%	12.0%
compute time	9.1%	20.0%	56.5%	87.1%
future overhead	0.1%	0.2%	0.5%	0.7%
other	0.0%	0.0%	0.1%	0.2%

Table 4.4: Results from speculative search

tory work first search, at least on machines such as Concert featuring a large number of processors. It is likely that the mandatory work-first search would fare better relative to the parallel exhaustive search on a machine with only a few, say three to five processors. And perhaps on extremely deep searches the mandatory work first procedure would catch up to the full parallel search by dint of its lesser total computational work, since at some point the parallel full search saturates the available processes and tasks start piling up on the task queue.

It is interesting to note that the percentage of the total processor time (busy and idle) used to do “real” computing is fairly constant.

#### 4.1.6 Speculative Search

The speculative search, *WORK-MORE*, first introduced in Chapter Three (Figure 3-7) gives somewhat disappointing results. It appears not to do significantly better than the parallel exhaustive search. The overhead from futures and memory contention is markedly higher than in the parallel-exhaustive search and the mandatory work-first search, but not to a troubling degree.

The variation of the speculative search in which a node updates its alpha-beta window after resolving each child does a bit better. (Table 4.5) The profile of this multiple-update speculative search explains to a certain extent this newfound improvement (Figure 4-5). Notice that the search ends rather abruptly with a sheer drop in the parallelism graph. This indicates that the search returned a value even though some of its processes had not finished executing.

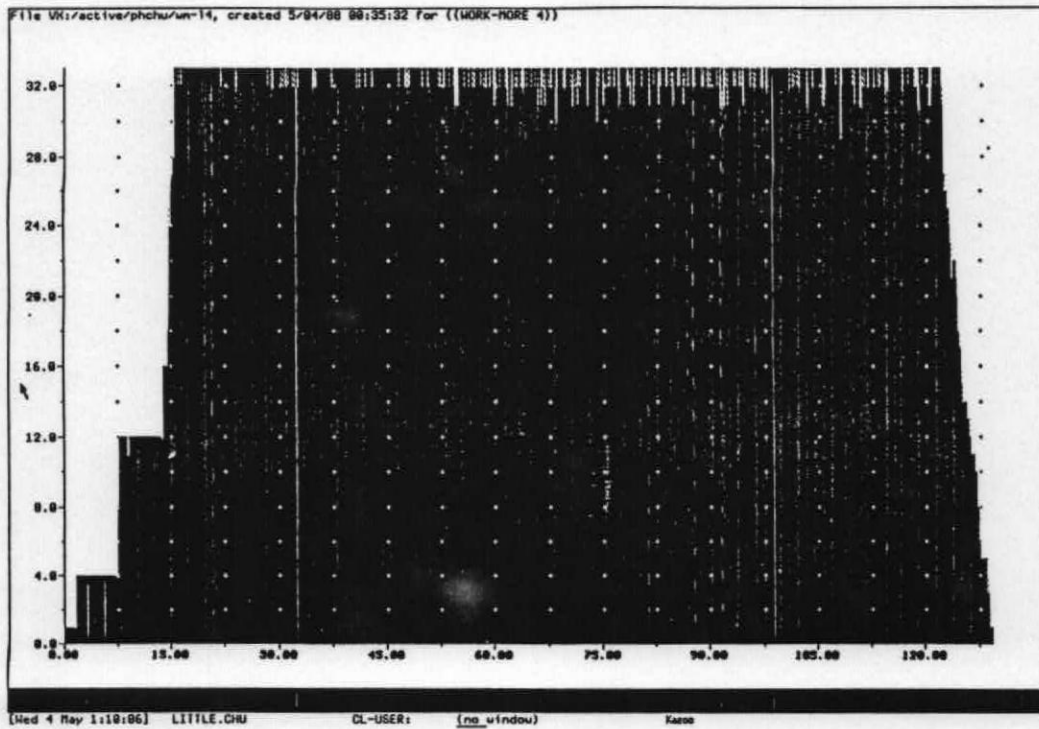


Figure 4-4: Speculative Search

search	two-ply	three-ply	four-ply	five-ply
final time	6.32s	13.82s	29.62s	107.80s
idle time	90.9%	80.1%	39.5%	10.9%
compute time	9.0%	19.7%	60.0%	88.3%
future overhead	0.2%	0.2%	0.5%	0.7%
other	0.0%	0.0%	0.1%	0.2%

Table 4.5: Results from speculative search with multiple updates per node.

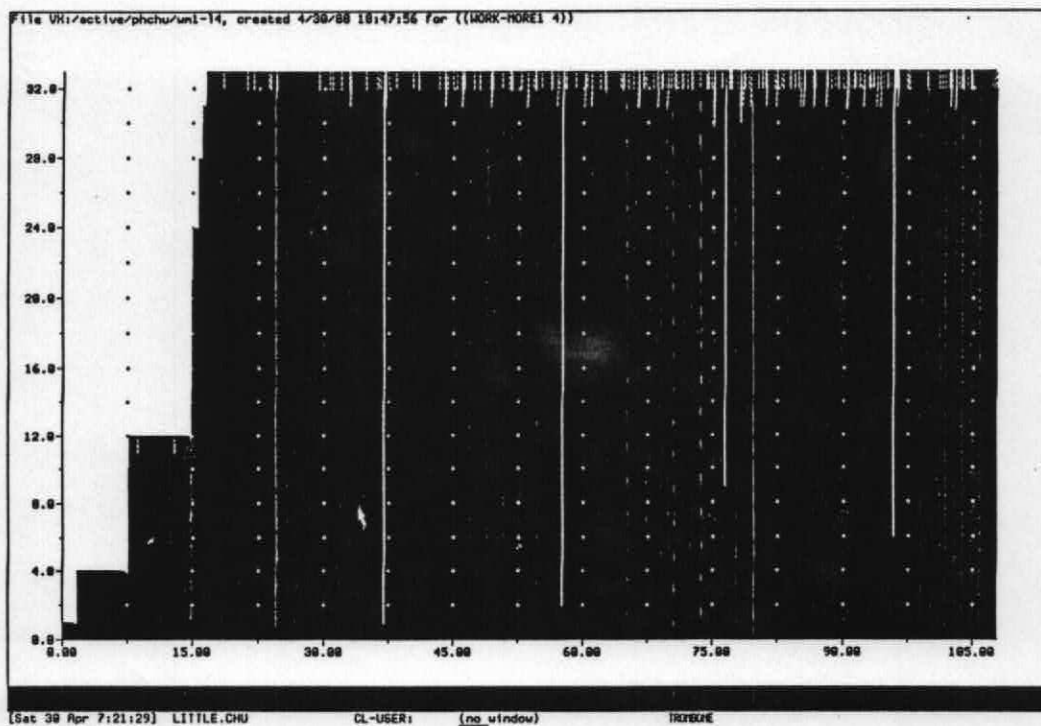


Figure 4-5: Speculative Search with more cutoff checks



In addition to the granularity of updating alpha-beta windows, different scheduling semantics seems to have a significant effect on the resulting parallelism profile and search time. For example, the use of *dfuture* instead of *future* negates the time savings achieved by *WORK-MORE1*. Unlike the latter search procedure *DWORK-MORE1* fails to return a value before all portions of the search has been completed (Figure ??), as it ends in a more gradual curve representing the tapering off of tasks.

This can be explained by noting that to maximize cutoff, what we need to do is not necessarily give priority to processing *lower* portions of the search tree, but instead we need to give priority to evaluating a minimum number of search paths, the portion evaluated in parallel in the mandatory work first search.<sup>1</sup>

This depth-first priority in fact is probably better provided by *future* than *dfuture*, since the use of *future* in our speculative search gives precedence to the evaluation of the children of a node, albeit in a left to right ordering. What would really be desirable is some way to dictate that the search paths necessary for cutoff are done first.

The importance of task scheduling is further demonstrated by a sample trial of the speculative search using *sfuture*. This variation, using an alpha-beta update after each child has been touched, featured the same cliff-ending effect as *work-more1*, yet still took about as long as the parallel exhaustive search.

#### 4.1.7 Using Parallel-Aspiration Search

The use of parallel aspiration search on the test case showed no improvement over sequential alpha-beta performance. In fact, increasing the number alpha-beta partitions increased the final search time. (Figure 4.6)

The lack of speedup suggests that the effectiveness of the parallel-aspiration

---

<sup>1</sup>Although it doesn't really matter *which* paths we evaluate, as long as we evaluate enough to make cutoff of other search paths possible.

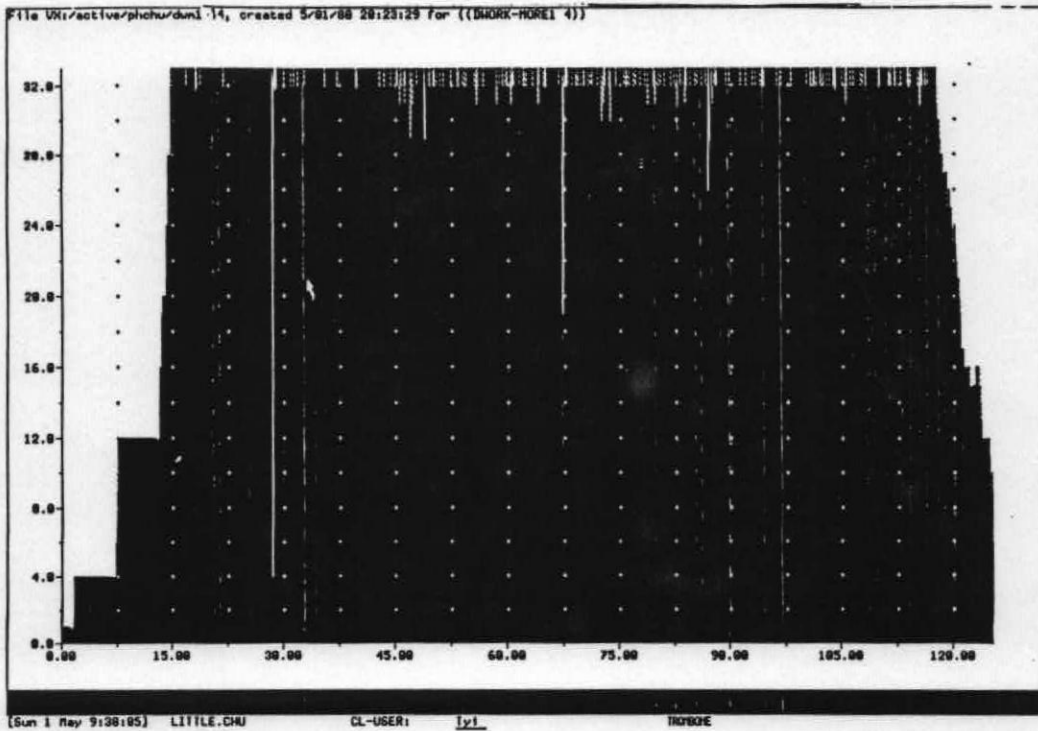


Figure 4-6: Speculative Search using Dfuture

windows	1	2	3	10	15	32
time	259.93s	259.35s	251.45s	278.38s	287.00s	312.17s

Table 4.6: Statistics on performance of the parallel aspiration search

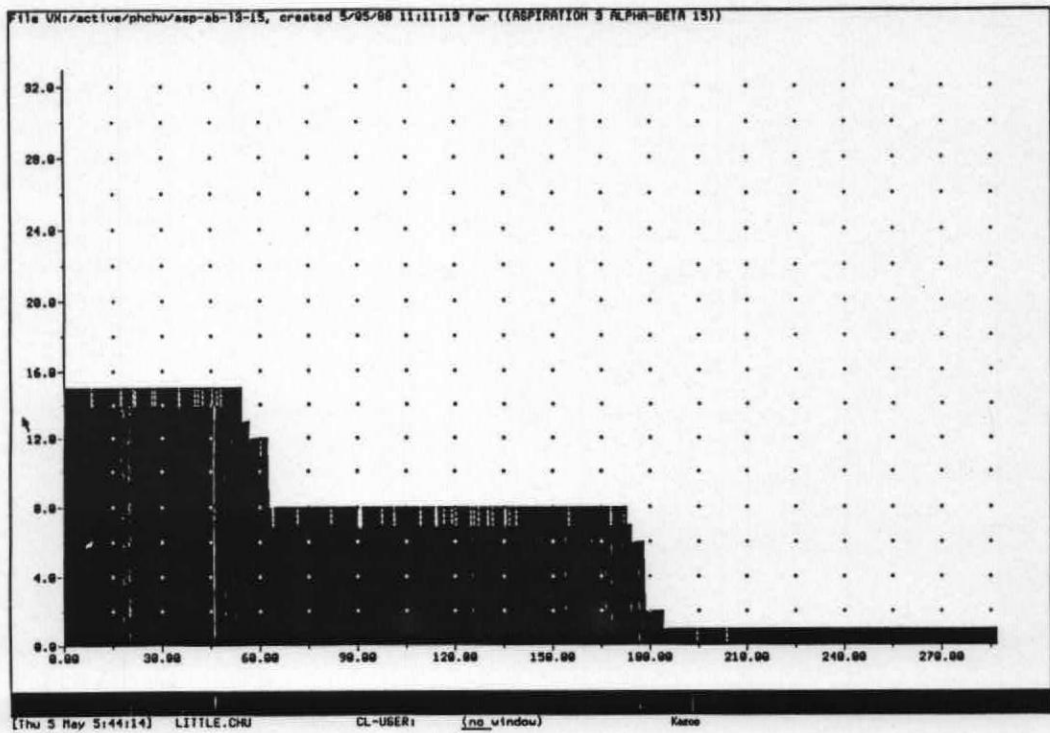


Figure 4-7: Parallel-aspiration search using fifteen alpha-beta partitions.

search lies not so much in the number of alpha-beta partitions as the prudent choice of the  $\alpha$  and  $\beta$  parameters. In [6], Baudet presupposes that his window-selection function, *SELECT-NEW-INTERVAL*, has some knowledge of the expected distribution of the move values to be found, which he argues is not an unreasonable assumption in game-playing programs.

The failure of a naive division of the alpha-beta window among available processors can be seen vividly in the profile of Figure 4-7. We know that the longest horizontal bar represents the search which actually returns the correct move, since all of the other bars terminate sooner, and the aspiration procedure returns as soon as it finds the right move. It is interesting to note that the searches on the alpha-beta windows that do not enclose the correct move do terminate more quickly than the search returning the final move, but a good number of them still take much longer than the parallel exhaustive search and the speculative searches.

The correlation between the addition of more alpha-beta divisions and *increased* computation time has a couple of possible explanations:

1. Increasing the number of searches that operate on the same global game object might dramatically increase delays due to bus contention.
2. Multiple searches on globally shared data might also increase the number of garbage collection operations during the search period. This hypothesis is supported by the observation of numerous garbage collection notifications during testing of the parallel-aspiration procedures.

Both possibilities point out aspects of the inner workings of Multilisp that warrant improvement, especially if deficiencies in either or both mechanisms leads to such performance degradation as evidenced by the parallel-aspiration results.



## Chapter 5

# Concluding Remarks

### 5.1 Conclusions

#### 5.1.1 Parallel Search

Adding futures to an exhaustive search seems to work fairly efficiently for fairly shallow search trees, but at some point, say a depth of three, the computational work saturates the available number of processors.

Still, the parallel exhaustive search is much more effective than the mandatory work first approach, due to the relatively small number of nodes that can be processed in a search tree without possibility of alpha-beta cutoff.

The attempt at a compromise solution, a search using speculative computation, did not fare much better than the parallel exhaustive search, but did suggest that better performance might be achieved if there existed more support for speculative computation in Multilisp, particularly in the form of more control over the scheduling of tasks. The slightly better performance of the multiple-update speculative approach, procedure *work-more1* in Figure 3-7, over other variations shows that adding computational work in exchange for more cutoff possibilities may be a good tradeoff.

The results of employing an "aspiration" technique were disappointing. Not only did this method fail to result in any speedup, it actually lengthened search time in correspondence to the number of partitions that the original alpha-beta window was divided into. The lack of speedup suggests that the promise of the aspiration technique lies more in the intelligent selection of the alpha-beta windows rather than a brute-force division into the available number of processors. Overall, it seems that the effectiveness of the aspiration technique presupposes a greater knowledge of the distribution of static values and the available number of processors than the other search methods.

In summary, the dramatic decrease in search time evidenced by the parallel-exhaustive search even on a five-ply game tree demonstrates that the area of computer search is one that can benefit greatly from application on multiprocessor architectures. Furthermore, the results from our attempts at speculative search indicate that speculative computation may be well suited for this type of problem and that support mechanisms for speculative computation should be pursued.

### 5.1.2 Multilisp

The use of a moderately large Multilisp program to explore opportunities for parallelism in computer game-playing allows us to distinguish many of the strengths and weaknesses of Multilisp.

The *future* paradigm turns out to be convenient and useful in parallelizing mandatory computation. Adding concurrency to a an exhaustive minmax search merely entailed wrapping *future* around certain key expressions. Likewise, the mandatory work first procedure differed from its sequential counterpart mainly by a check for the no-cutoff condition. In the latter case, the process of adding futures to the code resulted in a greater understanding of the sequential algorithm.

In addition, the necessity for using shared arrays in Multello (copying a board everytime a move is made would be prohibitively time-consuming) vindicates Mul-

tilisp's support of explicit parallelism and side effects.

The most prominent deficiency in Multilisp demonstrated during the course of this thesis is the lack of support for speculative computation. Useful would be features such as explicit task-killing directives and more programmer control over the scheduling of tasks. Also, the results of the parallel aspiration search suggested that garbage collection and bus and memory contention could be a problem in some cases.

In the other searches we tried, time lost due to memory locking apparently was not a serious problem. Also negligible was the computational overhead due to handling of futures was negligible, which is not surprising considering the large granularity of searches. It remains to be seen whether the cost of invoking *future* becomes a problem in finer-grain applications, such as parallelizing the static evaluation.

In summary, we have verified that Multilisp works quite well at parallelizing large-grained mandatory computation, and as a general-purpose language that supports both functional programming and side-effecting procedures it is well-suited to applications like Multello that require both for convenient implementation and efficient execution.

## 5.2 Future Work

### 5.2.1 Enhancing Multilisp

The initial results from Multello show speculative tasking to be a promising way to efficiently utilize multiprocessors for applications like game searches. However, the Multilisp procedures demonstrated in this thesis have been hindered by the lack of support for speculative computation in Multilisp.

Among the features needed are:

- Some way to kill a task after it has been determined that the task is not



relevant to its parent computation.

- Some way to explicitly prioritize speculative tasks so as to give more promising tasks, i.e. tasks whose values have a greater probability of being needed, a higher scheduling priority. It is unlikely, especially in an application such as alpha-beta search, that a compiler or interpreter can determine such priority implicitly.

### 5.2.2 More on Searching

This thesis is just an initial attempt to survey possible ways of speeding up game searches in a multiprocessor environment. Certainly many of the areas touched upon warrant further exploration.

For example, parallelizing the game search at a much finer level, e.g. the static evaluation or the generation of legal moves, could become an important speedup factor, especially for searches that do not make very efficient use of available processors. The mandatory work first search is one algorithm that might look much better compared to the parallel-exhaustive search if more advantage was taken of finer-grain parallelism. Furthermore, forms of speculative search might take good advantage of this additional parallelism. It seems likely that a speculative search combined with efficient task scheduling and a high degree of finer-grain parallelism may provide the most optimal game search using the concepts presented in this thesis. In any case, finer-grain parallelism should be explored at least to gather more information on how effective Multilisp is on these components of the game search.

Another area replete with unanswered questions is the parallel-aspiration search. The rather straightforward application of it in Chapter Three proved to be naive, suggesting that we should put more effort into exploring the questions of what sort of alpha-beta partitioning is needed to achieve a decrease in search time and how a procedure can determine these partitions.



Also, the parallel aspiration search allows opportunities to mix and match search algorithms. For example, instead of calling the sequential alpha-beta search, the aspiration loop could call the mandatory work-first search or even one of the speculative alpha-beta searches. Considering that many of the failed subsearches in the aspiration example (Chapter Four) took a small fraction of the total search time, an aspiration search combined with a mandatory work first or speculative alpha-beta search might result in very efficient processor utilization.

Another area of interest might be to analyze Baudet's report that using  $k$  processors, for  $0 < k < 4$ , he achieved more than  $k$ -fold speedup. It would be informative to try to duplicate this result and to explore his conclusion that the alpha-beta algorithm is suboptimal.

# Bibliography

- [1] Abelson, H. and Sussman, G. J. *The Structure and Interpretation of Computer Programs*. MIT Press 1984.
- [2] Agha, G. *Actors*. MIT Press 1987.
- [3] Akl, S. G. "The Computational Costs of Parallel Alpha-Beta Search." Queen's University at Kingston.
- [4] Akl, S. G., Barnard, D. T., and Doran, R. J. "Design Analysis and Implementation of a Parallel Alpha-Beta Algorithm." Department of Computing and Information Science. Queen's University, Kingston, Ontario, Canada, Technical Report 80-98. April 1980.
- [5] Akl, S. G., and Doran, R. J. "A Comparison of Parallel Implementations of the Alpha-Beta and Scout Tree Search Algorithm Using the Game of Checkers." Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada. TR 81-121, April 1981.
- [6] Baudet, G. M., "The Design and Analysis of Algorithms for Asynchronous Multiprocessors." Department of Computer Science. Carnegie-Mellon University, Pittsburgh. TR CMU-CS-78-116 April 28, 1978.
- [7] Bradley, E. and Halstead, R. H. "Simulating Logic Circuits: a Multiprocessor Application." M.I.T. Laboratory for Computer Science. May 28, 1987.

- [8] Charniak, E. and McDermott, D. *Introduction to Artificial Intelligence*. Addison Wesley, 1985.
- [9] Finkel, R. A. and Fishburn, J. P. "Parallelism in Alpha-Beta Search." *Journal of Artificial Intelligence*. Vol. 18, 1982.
- [10] Gray, L. "Using Futures in Parallel Computation." S.M. Thesis. Massachusetts Institute of Technology 1986.
- [11] Halstead, R. H., Loaiza, J. R., Ma, M. H. "The Multilisp Manual." M.I.T. Parallel Processing Group, June 1986.
- [12] Halstead, R. H. "Parallel Symbolic Computing." *ACM Transactions on Programming Languages and Systems* Vol. 7, No. 4, October 1985, pp. 501-538.
- [13] Halstead, R. H. "An Assessment of Multilisp: Lessons and Experience." *International Journal of Parallel Programming*, December 1986, Plenum Press, New York.
- [14] Halstead, R. H. "Parallel Computing Using Multilisp." *Parallel Computation and Computers for Artificial Intelligence*, J. Kowalik, ed., Kluwer Academic Publishers, 1987.
- [15] Halstead, R. H., Anderson, T. L., Osborne, R. and Sterling, T. L. "Concert: Design of a Multiprocessor Development System." M.I.T. Parallel Processing Group 1985.
- [16] Hasegawa, Goro with Maxine Brady. *How to Win at Othello*. Jove Publications, 1977.
- [17] Knuth, Donald E., Moore, Ronald W. "An Analysis of Alpha-Beta Pruning." Computer Science Department, Stanford University. TR August 1974.

- [18] Rees, J. and Clinger, W. (Eds.) "The Revised Report on the Algorithmic Language Scheme."
- [19] Steele, Guy. *Common Lisp: The Language*. Digital Press, 1984.
- [20] Winston, Patrick H. *Artificial Intelligence*. Addison Wesley 1984.