Game Tree Search

on

Massively Parallel Systems

Submitted in partial fulfillment
of the requirements for the degree of
Doctor rer. nat. in the Department of Mathematics and Computer Science,
University of Paderborn

by

Rainer Feldmann

Department of mathematics and computer science
University of Paderborn
Paderborn, Germany

Thesis committee:
Dr. Althöfer, Prof. Dr. B. Monien, Prof. Dr. F. Rammig

August, 1993

Preface to the English Version

This book is a translation of my Phd thesis titled "Spielbaumsuche auf massiv parallelen Systemen". It is the result of the research I did during the years 1989 to 1993 in the Department of Mathematics and Computer Science of the University of Paderborn, Germany.
Several people read the translation of my thesis and helped finding mistakes. Especially I would like to thank Lisa Jauss for correcting my translation and Prof. T. A. Marsland for his careful reading and his very helpful comments.

Paderborn, October 1993 Rainer Feldmann

Contents

1 Introduction **5**
1.1 Computer Chess and Artificial Intelligence . . . 5
1.2 History of Computer Chess . . . 5
1.3 Results . . . 8
1.4 Overview . . . 9
1.5 Basic Definitions . . . 10

2 The Sequential Game Tree Search Algorithm **13**
2.1 The Scout-Algorithm . . . 13
2.2 $\alpha\beta$-Enhancements . . . 21
2.2.1 Transposition Table . . . 22
2.2.2 Iterative Deepening . . . 28
2.2.3 Aspiration Search . . . 29
2.2.4 Killer Heuristic . . . 29
2.2.5 Principal Variation and Refutation Heuristic . . . 31
2.2.6 History Heuristic . . . 31
2.2.7 Move Ordering . . . 31
2.3 Implementation Details . . . 33
2.4 Hardware for the Sequential Algorithm . . . 35
2.5 Performance of the Sequential Algorithm . . . 36
2.5.1 Description of the Set of Test Positions . . . 36
2.5.2 Data for Different Search Depths . . . 37
2.5.3 Scout vs. $\alpha\beta$ . . . 37
2.5.4 Distance from the Minimal Game Tree . . . 39
2.5.5 The Effects of the Transposition Table . . . 41

3 The Parallel Game Tree Search Algorithm **43**
3.1 Hardware for the Parallel Algorithm . . . 43
3.1.1 Comparing DeBruijn Network, Torus and Grid . . . 45
3.2 Related Work . . . 46
3.2.1 Parallel Static Evaluation/Move Generation . . . 46
3.2.2 Parallel Window Search . . . 46
3.2.3 Game Tree Decomposition . . . 47
3.3 Dynamic Game Tree Decomposition . . . 51
3.4 Basic Version of Our Parallel Scout-Algorithm . . . 52
3.4.1 Start of a Master-Slave Relationship . . . 52
3.4.2 End of a Master-Slave Relationship . . . 55
3.4.3 Improvements of Search Windows . . . 57
3.4.4 The Distributed Algorithm . . . 59
3.4.5 Thoughts on the Distribution of Work Load . . . 67
3.5 Local Killer and History Heuristics . . . 68
3.6 The Distributed Transposition Table . . . 69
3.6.1 Local, Global or Distributed Hash Tables ? . . . 69

3.6.2 Implementation of a Distributed Transposition Table . . . 71
3.7 The Young Brothers Wait Concept . . . 74
3.7.1 The YBWC for Type-1 and Type-2 Nodes . . . 76
3.7.2 The Extended YBWC for Type-2 Nodes . . . 77
3.7.3 Combining the YBWC-1-2 and the YBWC* . . . 78
3.8 Load Distribution Strategies . . . 79
3.8.1 Necessity of a Global Search for Work . . . 79
3.8.2 Local Search for Work . . . 81
3.8.3 Global Search for Work With Returns . . . 82
3.8.4 Combining Local and Global Search for Work . . . 83

4 Performance on up to 256 Processors 85
4.1 Measures . . . 85
4.2 Measuring of the Nondeterministic Behaviour . . . 87
4.3 Behaviour of the Best Distributed Version . . . 92
4.3.1 Speedups for Several Search Depths . . . 92
4.3.2 Reasons for the Speedups Measured . . . 93
4.3.3 "Superlinear" Speedup in the Quiescence Search ? . . . 97
4.3.4 Measuring the Performance . . . 98
4.3.5 Speedups Under Tournament Conditions . . . 100
4.3.6 Efficiency . . . 103
4.3.7 8-ply Search with the DB(8): Single Positions . . . 106
4.4 Measuring of the Performance Loss . . . 107
4.5 Costs and Gains of a Distributed Transposition Table . . . 109
4.5.1 Gains of a Distributed Transposition Table . . . 110
4.5.2 Costs of the Distributed Transposition Table . . . 113
4.5.3 Conclusion . . . 116
4.6 Decrease of the Search Overhead by the YBWC . . . 116
4.7 Use of the Load Distribution Strategies . . . 120
4.8 Experiments with Varying Topologies . . . 127
4.9 Influence of the Dynamic Load Distribution . . . 130

5 Behaviour on More Than 256 Processors 133
5.1 Changes of the Experiments . . . 133
5.2 The Standard Version . . . 134
5.3 Scout vs $\alpha\beta$: Zero Window Search ? . . . 137
5.4 Reduced Transposition Table Accesses . . . 138
5.5 DB(8) vs. GC(32,32) under tournament conditions . . . 139

6 Conclusions and Open Problems 143

A A BCH Hash Function 145

B The Bratko Kopec Positions 149

References 151

Index 155

List of Figures 159

List of Tables 160

Chapter 1

Introduction

1.1 Computer Chess and Artificial Intelligence

Tree search algorithms play an important role in many applications in the field of artificial intelligence. For instance, theorem provers, expert systems, robot control systems and game playing programs contain tree searching algorithms as their basic part. Tree searching is used whenever decisions must be made that are based on complex knowledge, that cannot be implemented directly on a machine.

Game playing programs provide an excellent test bed for search algorithms for several reasons. Games like chess, checkers, go, nine-men-morris and many others define simple "worlds" in a very exact manner. The games are defined by some very simple rules which are easy to implement on a machine. However, these games are of high complexity. For example for many centuries the game of chess has been viewed as a test bed for humans' strategic and tactical abilities. Famous chess players are considered as intelligent as scientists, politicians or managers of successful companies. In addition, games themselves are often a mirror image of a small part of the society, that invented the games. For example, von Neumann and Morgenstern [74] describe the similarities of game theory and economic behaviour.

The game of chess is one of the most famous games in the world. During the last centuries humans have developed much of expert knowledge, which, saved in books and nowadays in databases, successively improves the quality of human play. The ELO rating was defined [19], which allows an assessment of the playing strength of human or machine chess players and, therefore, allows a comparison of the playing strength of two players. For these reasons computer chess is viewed today as a measurement of the success of research in the whole field of artificial intelligence.

1.2 History of Computer Chess

The history of machines playing chess started in the early 20th century. Torres y Quevedo built a machine that was able to win with rook and king against a king. Zuse described a formalization of the game of chess in his work "Der Plankalkül" [78] in 1945. First detailed descriptions that allow programming a machine to play chess are due to Shannon [66] and Turing [73]. The first running chess programs were implemented during the late 1950s. Their playing strength was

very poor. The first program that achieved an ELO rating was the chess program *MACHACK* by Greenblatt et al. in 1967 [32]. During a tournament in Massachusetts the program achieved a rating of 1640 (USCF). In 1974 the first computer chess world championships took place in Stockholm, Sweden. The title was won by the Russian program *KAISSA*. Faster and faster hardware and improved software allowed a significant increase in playing strength. Levy and Newborn [44, p. 5] describe this increase of playing strength from a rating of 1640 ELO in 1967 to more than 2500 ELO points today. At the time of writing only very few programs exist that are able to play at a level of 2400, the playing strength of an international master. For now, only *DEEP THOUGHT* has the playing strength of a grandmaster or slightly less. For a detailed description of the history of computer chess the reader is referred to the book by Levy and Newborn [44].

The history of computer chess makes clear the enormous improvement of the playing strengths of chess programs during the last decades. The following paragraph briefly describes the development of hardware and software for chess playing machines that made this improvement possible.

Software and Hardware Development

The game of chess, perhaps, is the most prominent example of a two person - zero sum - game with complete information and alternating right to move. Two players play against each other. At every moment the complete status of the game is known to both players. The players have contrary goals, i.e. a situation that is evaluated "good" for one of the players is evaluated "bad" for the opponent and vice versa. Zermelo [77] formalized the notion of a position which is a theoretical win, draw or loss in this kind of game. From this formalization the following algorithm, the so called minmax algorithm, can easily be derived: For a two person - zero sum - game with complete information let MAX be the player whose turn it is in the starting position, and let MIN be the opponent of MAX. Every terminal position v is assigned a value $\mathcal{F}(v)$ by the following rule:

$$\mathcal{F}(v) := \begin{cases} 1, & \text{if } v \text{ is a win for MAX} \\ 0, & \text{if } v \text{ is a draw} \\ -1, & \text{if } v \text{ is a win for MIN} \end{cases}$$

For an arbitrary nonterminal position s, with legal moves of the player to move leading to successor positions $s_1, \ldots s_b$, a value

$$\mathcal{F}(s) := \begin{cases} max\{\mathcal{F}(s_1), \ldots, \mathcal{F}(s_b)\}, & \text{if MAX is to move} \\ min\{\mathcal{F}(s_1), \ldots, \mathcal{F}(s_b)\}, & \text{if MIN is to move} \end{cases}$$

can be computed recursively.

If infinite sequences of moves are not allowed in the game, then by the above formula a value $\mathcal{F}(s)$ can be computed for every position s in finitely many steps. $\mathcal{F}(s) = 1$, if MAX can force a win from position s, it is 0, if neither MAX nor MIN can force a win and it is -1, if MIN is able to force a win.

The minmax algorithm obtained by recursively computing the value $\mathcal{F}(s)$ generates a tree, the so called game tree with root s : The nodes of the tree are the positions of the game, the edges of the tree correspond to the moves of the players MAX and MIN. The root of the tree corresponds to s, the successors of s in the game tree are the positions reachable from s by one ply. The leaves of the game tree correspond to the terminal positions of the game reachable from s.

Software

The minmax algorithm generates every node of the game tree with root s, and therefore, in a uniform game tree of width b and depth d, it generates b^d leaves. The following argument shows, that this, in general, is not optimal: Suppose the value $\mathcal{F}(s)$ of a position s with MAX to move should be computed. Then, any algorithm may halt when it finds a successor s_j of s with $\mathcal{F}(s_j) = 1$, because no other successor of s can give MAX a better result. An analogous case arises if for a successor of a node with MIN to move the value 0 is computed. Cancelling a computation at a node s after having computed the value of a successor of s is called a *cutoff*. Cutoffs were invented by Newell, Shaw and Simon [55]. In addition to the direct cutoffs described above deep cutoffs were introduced by Samuel [62] as well as Slagle and Dixon [68]. This results in an algorithm for game tree search that is called *$\alpha\beta$-algorithm*. Knuth and Moore [40] present an analysis of the $\alpha\beta$-algorithm. In the best possible case the $\alpha\beta$-algorithm visits

$$b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$$

leaves of any b/d-uniform game tree, in the worst case it visits, like the minmax algorithm, b^d leaves. The best behaviour is achieved if the first successor of every nonterminal node in the game tree is the best successor, i.e. it has the maximum $\mathcal{F}$-value of all the successors if MAX is to play and it has the minimum $\mathcal{F}$-value of all successors if MIN is to play. Knuth and Moore show that every algorithm that determines the $\mathcal{F}$-value of the root of a b/d-uniform game tree must visit at least the number of leaves the $\alpha\beta$-algorithm visits in the best possible case. From then on, the software development concentrated on improving the sorting of the successors of inner game tree nodes. Methods to sort the legal moves in a position were developed.

Greenblatt et al. [32] introduced the transposition table for their program *MACHACK VI*. The transposition table is a hash table for positions, in which results already computed for a position are stored. If the same position arises again during the search, e.g. by transposition of moves, the result stored in the transposition table may save searching identical subtrees of the whole game tree.

The method of window search and the killer heuristic were first used in the Russian program *KAISSA*, the killer heuristic was called "best move service". The killer heuristic saves a few moves for every level of the game tree, which proved to be good in that level in other subtrees of the game tree. These moves, if legal, are then sorted to the beginning of the move list of inner game tree nodes. The killer heuristic is best described by Akl and Newborn [1].

The method of window search uses the fact, that in many cases the value of the root of the game tree can be estimated roughly, so that an interval $[\alpha, \beta] \neq [-\infty, \infty]$ can be given, which contains the value of the root with high probability. In these cases, one first tries to prove that the value of the root is in the interval $[\alpha, \beta]$, by doing a window search. If the result x of the window search is not properly included in the interval $[\alpha, \beta]$, the root value has to be determined precisely by researching the game tree with the search window $[-\infty, x]$ or $[x, \infty]$ respectively.

Slate and Atkin [69] describe the method of iterative deepening. During the first iteration only the direct successors of the root node are generated, i.e. the game tree is searched to a depth of 1. Afterwards the game tree is searched to depths 2, 3, etc. up to a final search depth d. The results computed during the iterations with smaller search depths are used to improve the sorting of the successors of inner game tree nodes during the extensive searches to larger search depths. Therefore the most extensive searches are speeded up by the fast searches to smaller search depths. Especially when combined with the use of a transposition table a large

improvement of the sorting of successors in the game tree is achieved.

Schaeffer [63] describes the History Heuristic showing that with the help of this heuristic the sorting of successors of game tree nodes can be improved significantly.

Hardware

Parallel to the development of the above described software tools, the game tree search was speeded up by the use of faster and faster hardware. Condon and Thomson [16] describe the hardware of the chess machine *BELLE*. With the help of special hardware *BELLE* won the 1980 world computer chess championships in Linz, Austria. One of the worlds' best chess programs, *HITECH*, described by Berliner [11], uses special hardware to speed up the game tree search. The best chess machine at the time of writing, *DEEP THOUGHT*, gets most of its playing strength from the special hardware used to search the game tree at an amazing speed. The hardware of *DEEP THOUGHT* is described by Hsu [34].

The world champion program of 1983 and 1986, *CRAY BLITZ*, uses a Cray X-MP 4/8, at first as a sequential program. Hyatt et al. [37] describe their program *CRAY BLITZ* and a simple use of a two processor Cray X-MP during the world championships of 1983. Other programs began to use parallel machines to speed up the game tree search, e.g. *SUN PHOENIX*. Schaeffer [64] describes a parallel algorithm for the chess program *SUN PHOENIX*, running on a network of workstations. The parallel algorithm used in the less known chess program, *WAYCOOL*, implemented on an Intel Hypercube machine, is given by Otto and Felten [56]. Most recently a chess program has been developed for the CM5 [43]. Most of the parallel algorithms implemented behave quite well for small numbers of processors, however, are rather inefficient when running on a medium number of processors. Section 3.2 gives an overview over the parallel game tree search algorithms published up to now.

1.3 Results

In this thesis we present a parallel game tree search algorithm, which runs efficiently even on massively parallel systems. This game tree search algorithm is used in our chess program *ZUGZWANG*, vice world champion at the computer chess championships 1992 in Madrid, Spain. During this tournament *ZUGZWANG* ran on a 1024 processor system. Thus, the algorithm presented in this work is the first parallel game tree search algorithm that is used in a successfully playing chess program running on massively parallel hardware.

Furthermore, the algorithm presented here runs efficiently on a loosely coupled system of processors. With this it is well-suited to be used on parallel architectures realizable today.

With the help of experiments we will present reasons for the efficient behaviour of our algorithm and will investigate our implementation of a distributed transposition table. For this we simulate a global hash table in a distributed system by accessing the hash table with the help of messages. The main advantage in our approach is that the transposition table grows with the processor system, and this improves the search of the parallel algorithm. On the other side CPU time must be spent to route the messages necessary to access the hash table. Through experiments we will show that the use of the distributed transposition table compensates for the costs that have to be paid for routing the messages for table accesses.

To avoid superfluous work in a parallel environment we developed the Young Brothers Wait Concept (YBWC). The YBWC allows parallel evaluation of successors of inner game tree nodes, only if the first successor of the inner node is evaluated completely. In the sequential algorithm many subtrees are cut off, because they are known not to have any influence on the root value. The YBWC is a simple method, which allows the parallel algorithm to cut off many subtrees similar to the sequential one. In this thesis we will study the effects of the YBWC, as well as the effects of two variations of that method.

By the YBWC, parallel evaluation of some subtrees is delayed. This results in idle times for processors. In order to keep these idle times as small as possible, we developed special load distribution strategies. The effects of these strategies will be studied in this book. Our load distribution is based upon a combination of local and global searches for subproblems. Global search for work is used only if it is quite likely that the search for work will be successful. In this thesis we will show that the load distribution strategies chosen not only guarantee a good average processor work load, but also improve the search behaviour of our distributed algorithm.

1.4 Overview

First, we describe the sequential method, that will be the basis of our distributed algorithm, as well as the hardware used to implement this algorithm. Next, we analyze, with the help of some experiments, the performance of the sequential algorithm. We will especially show that the basis of our distributed algorithm is a very efficient search program using state-of-the-art heuristics to speed up the sequential search. In order to show this, we introduce the notion of the distance of a game tree from the minimal game tree. In addition we will investigate the load of the sequential program's transposition table.

Chapter 3 contains the description of our distributed Scout-algorithm. It opens with the definitions of the processor topologies used and a short overview about related work in the field of parallel or distributed game tree search. Here, we make clear the differences between our approach and the algorithms published so far. This introductory part is followed by a description of the basic version of our distributed Scout-algorithm as well as a parallelization of the most important heuristics. We describe the Young Brothers Wait Concept, two variations of the basic YBWC, and the load distribution strategies used, and we outline possibilities to reduce the communication overhead by the use of local methods for load distribution.

Chapter 4 contains the results obtained with our distributed algorithm using less or equal to 256 T800 processors. We first define the measures necessary to describe the performance of our distributed algorithm. Then we examine the results of experiments measuring the degree of the nondeterministic behaviour. Afterwards, we present the results of the experiments done with the version of our distributed algorithm described in Chapter 3. Here, some interesting conclusions concerning the behaviour of the sequential Scout-algorithm in non uniform game trees and the behaviour of the distributed algorithm on faster processors are made.

The rest of Chapter 4 contains a description of the effects caused by some features of the program. We investigate the loss of performance which, in our program, is caused by routing messages, discuss the costs and gains of the distributed transposition table and describe the effects of the variations of the Young Brothers Wait Concept. Here, we discuss an experiment which makes clear the differences between our distributed algorithm and the basic PV-Split algorithm. Next, the effects of the local load distribution methods are studied. An interesting conclusion drawn

from these experiments, is the fact that a combination of local and global strategies not only reduces the communication overhead but also improves the search of the distributed algorithm. Then, we compare the behaviour of our distributed Scout-algorithm on a grid and a torus of processors to the behaviour on a DeBruijn topology. The experiments show that the processor topology is, for the hardware used in our experiments, of great importance to the efficiency of our distributed algorithm. In order to compare our dynamic load distribution with a static one we then describe the performance of a version of our distributed algorithm with a more static game tree decomposition.

Chapter 5 contains the results of our distributed algorithm using up to 1024 T805 processors. It opens with a description of the performance of the version analyzed in Section 4.3 on a larger machine without any changes in the program. Then, we introduce two minor changes to the algorithm and describe the improvement of the efficiency caused by these changes.

1.5 Basic Definitions

Here, we define the most important notions used in this book. By $\mathbb{N}$ ($\mathbb{N}_0$) we denote the set of the natural numbers (including 0), by $\mathbb{Z}$ we denote the set of integers.

Definition 1.1 *(Graph)*

A directed graph $G = (V, E)$ is a tuple consisting of a set of nodes V and a set of edges $E \subset V \times V$. If $E \subset \mathcal{P}^2(V)$, then G is called an undirected graph.

For a directed graph $G = (V, E)$ the *indegree* (*outdegree*) of a node $v \in V$ is defined as

$$deg_i(v) := |\{v' \in V \mid (v', v) \in E\}| \ (deg_o(v) := |\{v' \in V \mid (v, v') \in E\}|).$$

For an undirected graph $G = (V, E)$, $deg(v) := |\{v' \in V \mid \{v, v'\} \in E\}|$ for any $v \in V$ is called the *degree* of v.

Definition 1.2 *(Path)*

Let $G = (V, E)$ be an (un)directed graph. A sequence $W = (v_1, \ldots v_n)$ of nodes is called a path in G, if $(v_i, v_{i+1}) \in E$ ($\{v_i, v_{i+1}\} \in E$) for all $i \in \{1, \ldots, n-1\}$.

$W = (v_1, \ldots v_n)$ is called *simple*, if $v_i \neq v_j$ for all $i, j \in \{1, \ldots, n\}$, $i \neq j$. W is called a *cycle*, if $v_1 = v_n$. The *length* of W is $n - 1$. For all $v, v' \in V$ the *distance* $\Delta_G(v, v')$ of v and v' in G is defined as the length of the shortest path from v to v' in G. The *diameter* of G is defined as $\Delta(G) := max\{\Delta_G(v, v') \mid v, v' \in V\}$.

Definition 1.3 *(Average distance)*

Let $G = (V, E)$ be a graph. The average distance of G is defined as

$$\Delta_\Phi(G) := \frac{1}{|V|^2} \cdot \sum_{v, v' \in V} \Delta_G(v, v')$$

Definition 1.4 *(Tree)*

A directed graph $G = (V, E)$ is called a tree, if

1. *G does not contain a cycle and*
2. *Exactly one node of G has indegree 0, all other nodes have indegree 1.*

If $G = (V, E)$ is a tree, then the node having indegree 0 is called the *root* of G. A node having outdegree 0 is called a *leaf* of G, nodes with outdegree > 0 are called *inner* nodes. If $v \in V$ is a node of G not equal to the root of G, then the (uniquely determined) node v' with $(v', v) \in E$ is called the *father* of v. If $v \in V$ is not a leaf, then the nodes $v' \in V$ with $(v, v') \in E$ are called *successors* or *sons* of v. Any two successors of a node are called *brothers*. For every node $v \in V$ the *depth* of v is the distance from the root of the tree to v. All nodes having the same depth d in G form a *level* in G. A tree $G = (V, E)$ is called *b/d-uniform*, if all inner nodes of G have outdegree b and all the leaves of G are at depth d.

Definition 1.5 *(Subtree)*

Let $G = (V, E)$ be a tree with root v_r, let $v \in V$ be a node of G.
$G(v) := (V(v), E(v))$ with

$$\begin{aligned} V(v) &:= \{v' \in V \mid \text{ the shortest path from } v_r \text{ to } v' \text{ contains } v\} \\ E(v) &:= \{(v', v'') \in E \mid v', v'' \in V(v)\} \end{aligned}$$

is called subtree of G with root v.

The following notation of the nodes of a tree is helpful in many situations.

Definition 1.6 *(Dewey decimal system)*

Let $G = (V, E)$ be a tree.

1. *The root of G is called ε.*
2. *The successors of a node $v \in V$ with $deg_o(v) = b \in \mathbb{N}_0$ are named $v.1, \ldots, v.b$.*

With the help of the Dewey decimal notations of the nodes we are able to distinguish the successors of an inner node of G in the following way:

Definition 1.7 *(left,right successor(s))*

Let $G = (V, E)$ be a tree, let $v \in V$ be an inner node of G.

1. *The node $v.1$ is called the left successor of v.*
2. *All other successors of v are called right successors.*

By using the above notations and definitions we are able to define a game tree:

Definition 1.8 *(Game tree)*

A game tree $G = (V, E, f)$ is a 3-tuple satisfying

1. *$G = (V, E)$ is a tree and $V \neq \emptyset$.*
2. *$f : V \to \mathbb{Z} \cup \{\pm\infty\}$.*

The nodes of a game tree will correspond to the positions of the game, the edges will correspond to the moves from one position to a successor position. The function f is called the *(static) evaluation function* for G. It maps the leaves of G into the ordered set $\mathbb{Z} \cup \{\pm\infty\}$. Nodes of even depth in G are called *MAX-nodes*, nodes of odd depth in G are called *MIN-nodes*.

Definition 1.9 *(Minmax value)*

Let $G = (V, E, f)$ be a game tree. Then the minmax value $F(v)$ of a node $v \in V$ is defined as

$$F(v) := \begin{cases} f(v) & \text{if } v \text{ is a leaf of } G. \\ max\{F(v') \mid (v, v') \in E\} & \text{if } v \text{ is an inner MAX-node of } G. \\ min\{F(v') \mid (v, v') \in E\} & \text{if } v \text{ is an inner MIN-node of } G. \end{cases}$$

By the use of the notion of the minmax value, the game tree search problem can be formalized as follows:

Definition 1.10 *(Game tree search problem)*

The following problem is defined to be the game tree search problem:

Given: A game tree $G = (V, E, f)$ with root ε.
Compute: $F(\varepsilon)$.

Every algorithm computing $F(\varepsilon)$ is called *game tree search algorithm*. For any inner node of a game tree we define the notion of the best successor as follows:

Definition 1.11 *(best successor)*

Let $G = (V, E, f)$ be a game tree, let $v \in V$ be an inner node of G. Then a successor $v.j$ is called best successor of v, if

- *$F(v.j) > F(v.i)$, $1 \leq i < j$ and $F(v.j) \geq F(v.i)$, $i \geq j$, if v is a MAX-node.*
- *$F(v.j) < F(v.i)$, $1 \leq i < j$ and $F(v.j) \leq F(v.i)$, $i \geq j$, if v is a MIN-node.*

With this we are able to define a unique principal variation for every game tree:

Definition 1.12 *(Principal variation)*

Let $G = (V, E, f)$ be a game tree with root ε. Then the principal variation $(v_0, \ldots, v_t)$ of G is defined by:

- *$v_0 := \varepsilon$.*
- *v_{i+1} is the best successor of v_i, $0 \leq i < t$.*
- *v_t is a leaf of G.*

Chapter 2

The Sequential Game Tree Search Algorithm

2.1 The Scout-Algorithm

The $\alpha\beta$-algorithm is the basis of all successful game tree search algorithms. Figure 2.1 shows the basic form of the $\alpha\beta$-algorithm.

```
function alphabeta(v : node; α, β : integer): integer;
var i, b : integer;
begin
    generate all successors v.1, ..., v.b of v;
    if b = 0 then return(f(v));                /* static evaluation of a leaf */
    for i := 1 to b do
    begin
        if v is MAX-node then
        begin
            α := max(α, alphabeta(v.i, α, β));
            if α ≥ β then return(α);           /* cutoff */
            if i = b then return(α);           /* all successors evaluated */
        end else                               /* v is MIN-node */
        begin
            β := min(β, alphabeta(v.i, α, β));
            if α ≥ β then return(β);           /* cutoff */
            if i = b then return(β);           /* all successors evaluated */
        end
    end
end;
```

Figure 2.1: Basic form of the $\alpha\beta$-algorithm

Knuth and Moore [40] show that the following inequalities hold for all nodes v of any game tree:

Let $x :=$ alphabeta(v, α, β). Then

$$\begin{array}{ll} x \leq \alpha & \text{if } F(v) \leq \alpha \\ x = F(v) & \text{if } \alpha < F(v) < \beta \\ x \geq \beta & \text{if } F(v) \geq \beta \end{array}$$

Therefore, the function call "$x :=$ alphabeta$(\varepsilon, -\infty, \infty)$" returns $F(\varepsilon)$.

Reinefeld [59] presents many variations of the $\alpha\beta$-algorithm. In our program we use the Scout-algorithm with fail soft improvement as the basic sequential algorithm. This algorithm is presented in Figure 2.2 in a pseudo code. A similar description together with a more detailed analysis can be found in Reinefeld's book [59, p. 34-48]. Pearl [58] shows that the run time of the Scout-algorithm is asymptotically optimal.

For the Scout-algorithm with fail soft improvement from Figure 2.2 the following inequalities hold for every node of any game tree:

Let $x :=$ scout(v, α, β). Then

$$\begin{array}{ll} F(v) \leq x \leq \alpha & \text{if } F(v) \leq \alpha \\ F(v) = x & \text{if } \alpha < F(v) < \beta \\ F(v) \geq x \geq \beta & \text{if } F(v) \geq \beta \end{array}$$

The function call

$$x := \text{scout}(\varepsilon, -\infty, \infty);$$

therefore again returns $F(\varepsilon)$.

At the beginning of a computation at a node v, all successors of v are generated by the Scout-algorithm. If there are no successors of v ($b = 0$), then v is a leaf of the game tree and the value of the static evaluation function for v, $f(v)$, is returned. If v is not a leaf, then all successors of v are searched subsequently. The first successor $v.1$ of v is searched by a call of the function *scout* with bounds α and β, all right successors are searched with the *zero width window* $[u, u+1]$. If v is a MAX-node, then a research has to be done for a successor $v.i$, if the following conditions hold:

1. $(x > u)$: The bound x returned for $v.i$ is an improvement of the best bound found so far for v.

2. $(x < \beta)$: The bound returned for $v.i$ is not sufficient to produce a cutoff at v. If the bound returned for $v.i$ cuts off the other successors of v, no research is necessary.

3. $(i > 1)$: Only right successors of v were searched with a zero width window. Therefore a research is necessary for right sons only.

4. If all successors of $v.i$ are leaves in the game tree then, despite the fact that these leaves may have been searched with zero width window, the exact values of these leaves were returned and combined to the value of $v.i$. All these exact values were larger than u, since otherwise the value of $v.i$ would not have been larger than u (Note that $v.i$ is a MIN-node). Therefore, at node $v.i$ no cutoff could have occurred, i.e all successors of $v.i$ were

```
function scout(v : node; α, β : integer): integer;
var i, b, x, low, val, high : integer;
begin
    generate all successors v.1, ..., v.b of v;
    if b = 0 then return(f(v));                    /* static evaluation */
    low := α; high := β;
    if v is MAX-node then val := −∞ else val := ∞;
    for i := 1 to b do
    begin
        x := scout(v.i, low, high);                /* zero window search */
        if v is MAX-node then
        begin
            if (x > low) and (x < β) and (i > 1) and
              at least one successor of v.i is not a leaf then
              x := scout(v.i, low, β);             /* research */
            low := max(low, x);
            val := max(val, x);
            if val ≥ β then return(val);           /* cutoff */
            high := low + 1;
        end
        else                                       /* v is MIN-node */
        begin
            if (x < high) and (x > α) and (i > 1) and
              at least one successor of v.i is not a leaf then
              x := scout(v.i, α, x);               /* research */
            high := min(high, x);
            val := min(val, x);
            if val ≤ α then return(val);           /* cutoff */
            low := high − 1;
        end
    end;
    return(val);
end;
```

Figure 2.2: The Scout-algorithm

evaluated. Thus the bound for the node $v.i$ was computed on the basis of all exact values necessary to compute the exact value of node $v.i$. Therefore, the value computed for $v.i$ was the minmax value of $v.i$. A research of $v.i$ would return the same value and therefore is superfluous. This was first noted by Reinefeld [59].

Since v is a MAX-node, the lower bound for v may be improved by searching some successor of v. If a cutoff is achieved by such a search ($m \geq \beta$), then the function *scout* returns the cutoff value. Otherwise the zero width window is recomputed for the searches of the next successor of node v. The algorithm behaves analogously at MIN-nodes.

Figures 2.3 and 2.4 contain an equivalent version of the Scout-algorithm. The procedures "downwards" and "upwards" called in the procedure "iterate" are described in Figure 2.4. This iterative version of the Scout-algorithm will be the basis of our distributed game tree search algorithm. The static evaluation function $f(v)$ of our algorithm will be described in more detail by Mysliwietz [50].

With the help of this version some usefuls notations can be defined. For the following definitions let τ be a point of time during the computation in which the iterative Scout-algorithm is not working in the procedures "upwards" or "downwards".

Definition 2.1 *(Actual variation)*

Let $G = (V, E, f)$ be a game tree, let ε be the root of G. Let a processor P be searching G after the call of the procedure "iterative scout". Let t be the depth variable of the procedure "iterative scout", which is changed only by a call of one of the procedures "upwards" or "downwards". We call the sequence $(v_0 = \varepsilon, v_1, \ldots, v_t)$ of the contents of the variables v_i, $0 \leq i \leq t$ the actual variation of P at time τ. The nodes $v_0, \ldots, v_t$ are called actual nodes.

The actual variation changes with every call of one of the procedures "upwards" or "downwards" during the search in a game tree. Some nodes of the game tree may never become actual. However, through researches, nodes may become actual several times. Of importance is the content of the variables α and β at a time a node becomes actual:

Definition 2.2 *(Search window)*

Let $G = (V, E, f)$ be a game tree, let $v \in V$ be a node of G of depth i, which becomes actual at time τ. Let α and β be the contents of the variables α_i and β_i at time τ. Then $[\alpha, \beta]$ is called the search window of node v.

Definition 2.3 *(Subproblem)*

Let $G = (V, E, f)$ be a game tree, let $v \in V$, let $\alpha, \beta \in \mathbb{Z} \cup \{\pm\infty\}$. A subproblem with root v and search window $[\alpha, \beta]$ is the following problem:

Given: The subtree $G(v) = (V(v), E(v), f)$ with root v of G, $\alpha, \beta \in \mathbb{Z} \cup \{\pm\infty\}$.

Compute: A value x with

$$\begin{array}{ll} F(v) \leq x \leq \alpha, & \text{if } F(v) \leq \alpha \\ F(v) = x, & \text{if } \alpha < F(v) < \beta \\ F(v) \geq x \geq \beta, & \text{if } F(v) \geq \beta \end{array}$$

```
procedure iterate(var cutoff, improvement: boolean);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search windows at level i,
   let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables for level i. */
 begin
     cutoff:= false; improvement:= false;
     if the successors of v_t are not yet generated then
     begin
         generate all successors v_t.1, ..., v_t.b_t of v_t;
         if b_t = 0 then begin val_t := f(v_t); upwards(t); end
                  else downwards(t, v_t.1, [α_t, β_t]);
     end else
     begin                                  /* Going upwards from a successor v_t.i of v_t */
         if v_t is MAX-node then
         begin
             if (val_{t+1} > low_t) and (val_{t+1} < β_t) and (i > 1) and
               at least one successor of v_t.i is not a leaf then
               downwards(t, v.i, [low_t, β_t])                    /* Research */
             else
             begin
                 if (val_{t+1} > low_t) then improvement:= true;
                 if (val_{t+1} ≥ β_t) then cutoff:= true;
                 low_t := max(low_t, val_{t+1}); val_t := max(val_t, val_{t+1}); high_t := low_t + 1;
             end
         end else                                                /* v_t is MIN-node */
         begin
             if (val_{t+1} < high_t) and (val_{t+1} > α_t) and (i > 1) and
               at least one successor of v_t.i is not a leaf then
               downwards(t, v.i, [α, low_t])                      /* Research */
             else
             begin
                 if (val_{t+1} < high_t) then improvement:= true;
                 if (val_{t+1} ≤ α_t) then cutoff:= true;
                 high_t := min(high_t, val_{t+1}); val_t := min(val_t, val_{t+1}); low_t := high_t − 1;
             end
         end;
         if cutoff or all successors of v_t searched then upwards(t)
           else downwards(t, v_{i+1}, [low_t, high_t]);
     end
 end;
```

Figure 2.3: Iteration for the iterative Scout-algorithm

```
procedure upwards(var t: level);
/* Shortens the actual variation from (v_r, ..., v_t) to (v_r, ..., v_{t-1}) */
begin
    t := t - 1;
end;

procedure downwards(var t: depth; v: node; [α, β]: search window);
/* Lengthens the actual variation from (v_r, ..., v_t) to (v_r, ..., v_t, v) */
begin
    t := t + 1;
    v_t := v;
    α_t := α; β_t := β;
    low_t := α; high_t := β;
    if v_t is MAX-node then val_t := -∞
                       else val_t := ∞;
end;

function iterative scout(v :node, [α, β] : search window): integer;
/* Iterative Scout-algorithm.
   The variables "cutoff" and "improvement" are necessary for the use
   in the distributed algorithm */
var t : depth;
    ...
begin
    /* Initialize the actual variation with the root only */
    α_0 := α; β_0 := β;
    low_0 := α; val_0 := -∞; high_0 := β;
    v_0 := root ε of the game tree;
    /* Perform "iterate" until the root node is solved */
    repeat
        iterate(cutoff, improvement);
    until all successors of v_0 are evaluated;
    iterative scout:= val_0;
end;
```

Figure 2.4: Iterative Scout-algorithm

A subproblem may e.g. be solved by calling $x := scout(v, \alpha, \beta.)$ The *depth* of a subproblem is the depth of the root of the subproblem in G.

Definition 2.4 *(Cutoff/fail node)*

Let $G = (V, E, f)$ be a game tree, let $v \in V$ be a node of G, which becomes actual at time τ, let $[\alpha, \beta]$ be the search window of v at time τ.

v is called cutoff node, if

$$\begin{array}{ll} F(v) \geq \beta & \text{if } v \text{ is a MAX-node} \\ F(v) \leq \alpha & \text{if } v \text{ is a MIN-node} \end{array}$$

v is called fail node, if

$$\begin{array}{ll} F(v) \leq \alpha & \text{if } v \text{ is a MAX-node} \\ F(v) \geq \beta & \text{if } v \text{ is a MIN-node} \end{array}$$

We denote the work done by the Scout-algorithm in a subtree of a fail node v by *fail search.* If v is a cutoff node and the cutoff is caused by the evaluation of a successor $v.j$ of v, then the cutoff is called a *direct cutoff*, if there exists a brother u to the left of v with $F(u) \geq F(v.j)$, if v is a MIN-node and $F(u) \leq F(v.j)$, if v is a MAX-node. The cutoff at v is caused by the value of its brother u. Cutoffs that are not direct cutoffs are called *deep* Cutoffs.

The following example shows how the Scout-algorithm works compared to the pure $\alpha\beta$-algorithm.

Figure 2.5 at the top shows a game tree and the evaluations of some of the leaves. The evaluations of the other leaves are arbitrary. In the second tree, all nodes of the game tree are marked black that are visited by the Scout-algorithm before the exact value of the rightmost son of the root is determined by a research:

The algorithm recursively descends from the root v of the game tree to the node $v.1.1.1.1$. It evaluates this node with the help of the static evaluation function yielding a value of 3. Then the node $v.1.1.1.2$ is evaluated to 5, which results in a value of 3 for the node $v.1.1.1$. The algorithm proceeds searching node $v.1.1.2$ with search window $[3, 4]$, in order to test, whether $F(v.1.1.2) > 3$. After evaluation of the sons of $v.1.1.2$, this node is assigned a bound of ≥ 6 because of the fail soft improvement. Thus a research seems to be necessary. However, this research can be saved, since both sons of $v.1.1.2$ are leaves and the value returned for the sons was their exact value. Therefore, we have $F(v.1.1.2) = 6$. Next the node $v.1.1.3$ is searched with a search window $[6, 7]$. After evaluating $v.1.1.3.1$ with $F(v.1.1.3.1) = f(v.1.1.3.1) = 5$ it is obvious that $F(v.1.1.3) \leq 5$. Thus the evaluation of $v.1.1$ may be stopped, resulting in an exact value of $F(v.1.1) = 6$, without searching the node $v.1.1.3.2$. (direct cuoff at node $v.1.1.3$). The algorithm starts evaluating the node $v.1.2$ with a search window $[5, 6]$, in order to test, whether $F(v.1.2) < 6$. After the evaluation of the leaves $v.1.2.1.1$ and $v.1.2.1.2$ we have $F(v.1.2.1) = 9$ and therefore it is obvious that $F(v.1.2) \geq 9$. Thus an evaluation of the nodes $v.1.2.2$ and $v.1.2.3$ is saved. We now have $F(v.1) = 6$.

The evaluation of the node $v.2$ is begun with a search window $[6, 7]$ to test whether $F(v.2) > 6$. The algorithm evaluates the leaves $v.2.1.1.1$ and $v.2.1.1.2$ each with a value of 8, yielding $F(v.2.1.1) = 8$. Therefore, $F(v.2.1) \geq 8$. Since the search window at node $v.2.1$ is $[6, 7]$ and since $v.2.1$ is a MAX-node, a cutoff is obtained at $v.2.1$. The right brothers of $v.2.1$ will not be evaluated. The algorithm proceeds searching node $v.2.2$. After the evaluation of the leaves $v.2.2.1.1, v.2.2.2.1$ and $v.2.2.3.1$ with values $6, 5$ and 6, it is obvious that $F(v.2.2) \leq .6$ Therefore,

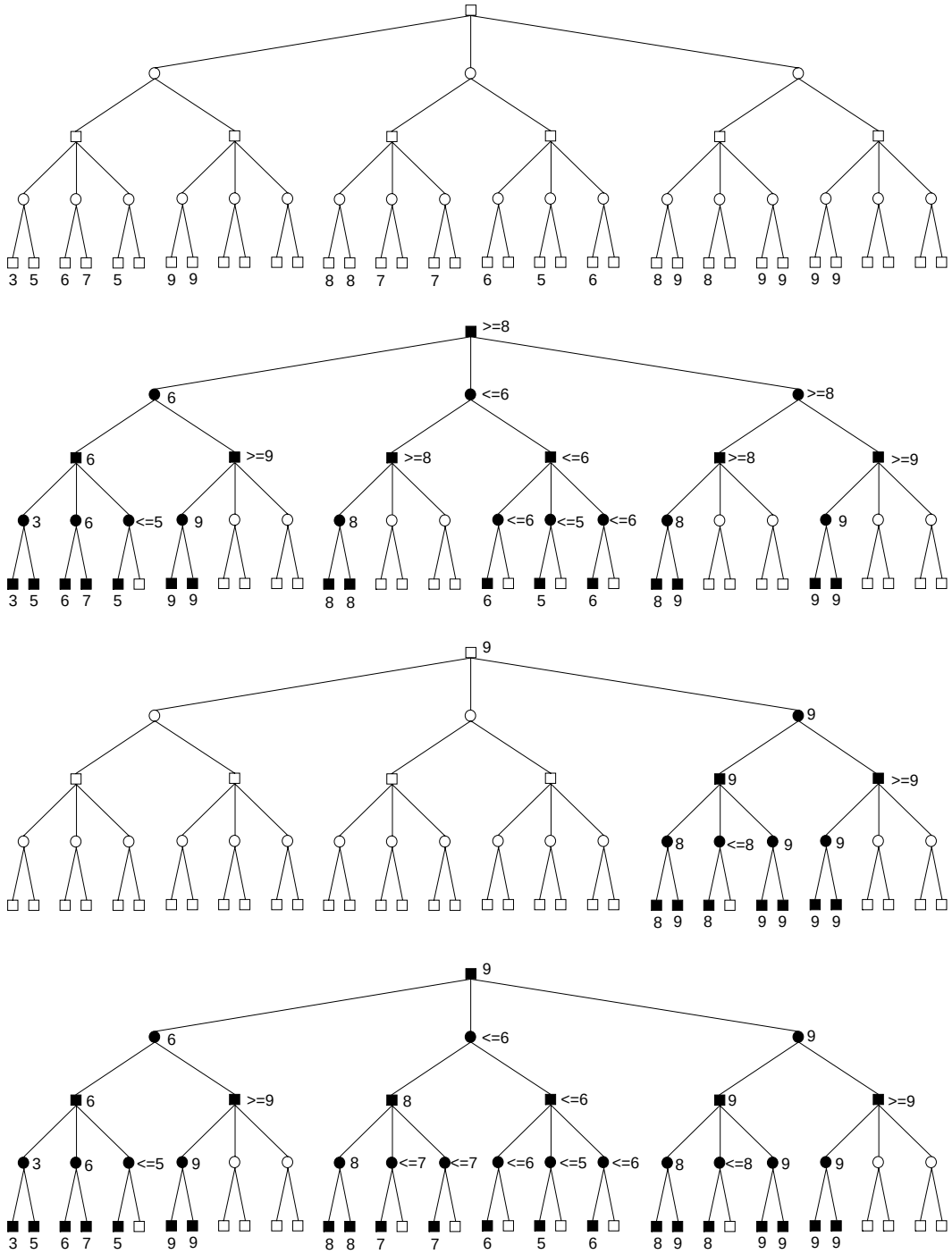


Figure 2.5: Searching a game tree with the Scout-and the $\alpha\beta$-algorithm

we have $F(v.2) \leq 6$, knowing that the node $v.2$ is a less favourable alternative to the node $v.1$ for the MAX-player.

The algorithm starts searching node $v.3$ with search window $[6, 7]$. After the evaluation of the leaves $v.3.1.1.1$ and $v.3.1.1.2$ it computes $F(v.3.1.1) = 8$. Therefore, a cutoff is obtained at node $v.3.1$ since the search window is $[6, 7]$. The algorithm continues the search at node $v.3.2$. After the evaluation of $v.3.2.1.1$ and $v.3.2.1.2$ each with a value of 9 $F(v.3.2) \geq 9$ may be concluded. And so $F(v.3) \geq 8$. It follows that the node $v.3$ is a better alternative compared to the best alternative $v.1$ found so far for the player MAX at the root. If only the best move at the root of the game tree is to be computed , then the search may be stopped at this point. The move from v to node $v.3$ guarantees the MAX-player a value of at least 8 thus being better than any alternative. However, we are looking at the problem to determine the value $F(v)$ of the root, which at this stage of the computation is known only to be greater than or equal to 8.

The exact value of the node $v.3$ must therefore be computed by a research The algorithm starts searching at node $v.3$ with search window $[8, \infty]$. The nodes visited during this research are marked black in the third game tree of Figure 2.5. Then $F(v.3) = 9$ is computed yielding $F(v) = 9$.

In order to compute $F(v) = 9$, the algorithm must evaluate a total of 19 different leaves, however the leaves $v.3.1.1.1$ and $v.3.2.1.1$, as well as $v.3.2.1.1$ and $v.3.2.1.2$, were evaluated twice. This results in a total of 23 static evaluations to compute $F(v)$. The game tree at the bottom of Figure 2.5 shows the nodes which are visited by the $\alpha\beta$-algorithm. In this example the $\alpha\beta$-algorithm needs a total of 21 static evaluations of some leaves in order to compute $F(v) = 9$. Note that the $\alpha\beta$-algorithm must evaluate the nodes $v.2.1.2.1$ and $v.2.1.3.1$, which were not visited by the Scout-algorithm.

In practical tests the Scout-algorithm proves to be superior compared to the $\alpha\beta$-algorithm. Reinefeld [59, p. 143ff] describes experiments with synthetical generated, practically oriented game trees in which the Scout-algorithm saves roughly 25% of the leaf evaluations of the $\alpha\beta$-algorithm. Observation 2.3 on page 37 shows that in the sequential version of our chess program, which serves as an application here, the Scout-algorithm proves to be superior compared to the $\alpha\beta$-algorithm as well.

2.2 $\alpha\beta$-Enhancements

In the previous sections we described the basic version of the sequential Scout-algorithm. The efficiency of this algorithm depends heavily on the ordering of the successors of inner game tree nodes. Therefore, in today's state-of-the-art game playing programs, easily computable heuristics are used to sort moves. Furthermore, in real applications, often acyclic graphs instead of trees are to be searched. This is due to the fact that in many games the position which results in updating three plies a, b and c from a given position v, is the same as the position after updating the plies c, b and a. Thus, in the graph to be searched, two directed paths may lead from some node v to some other node w. Of course, one would like to avoid searching the subtree with the root w twice. Therefore, a transposition table is used. Despite this we will refer to searching trees, instead of searching acyclic graphs.

The implementation of the transposition table, as well as the other heuristics used to sort moves, are described in the next sections. The heuristics used are iterative deepening, aspiration search, the killer heuristic, principal variation and refutation heuristic, and history heuristic. After

explaining each heuristic we will describe the overall move ordering of our algorithm.

2.2.1 Transposition Table

The transposition table is a hash table for nodes and for the results already computed for these nodes. An entry of the transposition table is a 6-tuple

$$(s, x, k, z, t, c).$$

Here the meanings of the components are

- s : *Key:* 30 bits, which serve as an "identification" key for the position stored in the entry.
- x : *Value:* The value computed for the position stored.
- k : *Flag:* Indicates , whether "value" is an exact value, an upper bound or a lower bound.
- z : *Move:* The best move found for the position stored.
- t : *Depth:* Search depth to which the "value" was computed.
- c : *Time stamp:* This component is not used when computing test positions as described in this work. When playing a whole game the time stamp helps to overwrite old entries in the hash table by actual ones.

The transposition table consists of 160000 entries. Two operations, namely "Store" and "Read", are defined for the transposition table. Before we start to describe these operations, we will present the hash function used.

The BCH hash function

In order to use the transposition table efficiently, our game tree searching algorithm uses a BCH hash function. These kinds of hash functions are built from the BCH codes, which are well known in the field of algebraic coding theory. First, we want to discuss why the use of these hash functions makes sense. We are looking for a hash function

$$h: \quad \{v \mid v \text{ is a chess position}\} \quad \rightarrow \quad \{0, \ldots, max - 1\} \ ,$$

where max is the number of entries of the hash table. A nice property of the hash function would be if it maps the set of chess positions into the set $\{0, \ldots, max - 1\}$ with as few as possible *collisions*, i.e. situations, in which $h(u) = h(v)$ but $u \neq v$. Since the number of hash table entries is expected to be very small as compared to the number of chess positions, this constraint is not really helpful for practical purposes. However, the constraint to keep the number of collisions small may be reformulated: We are looking for a hash function so that for every chess position v the depth of the game tree with root v, which is mapped without any collisions into the hash table is as large as possible.

With this we get a hash function, which produces collisions only when mapping two game tree nodes into the hash table, that have large distance in the game tree. A hash function that provides collision security for all nodes in a game tree of depth 2, is preferable to a hash function that guarantees this security only for trees of depth 1.

Looking at the problem in this way leads to a similar problem in the field of coding theory: Given a word $w \in \{0,1\}^m$ consisting of m bits, which are to be transmitted via a noisy channel. Then, transmitting the word may change a few bits. Therefore, one is looking for a function

$$A: \quad \{0,1\}^m \quad \rightarrow \quad \{0,1\}^k,$$

such that the transmission of $n = m + k$ bits (m information bits and k code bits) allows the detection rather the correction, of as many as possible faulty bits. Here k should be as small as possible. Efficient codes in this sense are BCH codes. The theory of BCH codes will not be evaluated in this book, because some results from the theory of finite fields are necessary in order to understand BCH codes. BCH codes are described e.g. in Berlekamp's book [9]. Warnock and Wendroff [76] were the first to use BCH codes as a hash function in a chess program.

Representation of a chess position

A chess position is represented by a sequence of bits $(b_0, \ldots, b_{m-1})$ in the following manner: In the game of chess each player has 6 different pieces, so together we have 12 different pieces on the board, which we enumerate as $\{0, \ldots, 11\}$. These pieces may be placed on 64 squares, enumerated as $\{0, \ldots, 63\}$. Now we define

$$b_{64 \cdot i + j} = 1 \Leftrightarrow \text{A piece of type } i \text{ is placed on square } j.$$

With this we have a representation of a chess position consisting of $64 \cdot 12 - 16 \cdot 2 = 736$ bits, because there are 16 squares on the board which may not contain white nor black pawns. The rights to castle are coded into four additional bits. Eight bits are used to code en passent squares. In endgame positions the player to move may be coded into another bit. Thus, we have a unique representation consisting of 748 or 749 bits.

This kind of representation has the advantage that the bit sequences corresponding to a chess position and a successor of this position, which is reached by a normal move, differ only in two bits; i.e. the representations of positions lying on a variation of l normal plies, differ in at most $2 \cdot l$ bits, in many cases, however, in fewer bits. Here normal moves are non-capturing moves that do not change the rights to castle and do not change the possibilities to capture en passent. Suppose, we had a coding function A with k code bits, which allows the detection of every $2 \cdot l$ bit errors in a sequence of m bits. Then $A(b) \neq A(c)$, if the sequences of bits b and c differ in at most $2 \cdot l$ bits. Such a function is then a useful hash function for hash tables of size 2^k. A BCH hash function is constructed explicitly in Appendix A.

Advantages of a BCH hash function

Warnock and Wendroff [76] present a table, which compares the number of protected bits to the number of bits necessary to guarantee this protection. This information is provided in Table 2.1. Here the underlying code for l protected bits is a combination of a BCH code for the protection of $l - 1$ bits and a parity check.

It follows that a hash table of size $160000 \approx 2^{17.3}$ protects two bits. Hash functions with this property may easily be constructed without knowledge from the field of coding theory. However, there is another advantage of the BCH hash functions: In Section 3.6 we will present our implementation of a distributed hash table, which, using p processors, allows the access of

Protected	2	4	6	8	10	12	14
Code bits	11	21	31	41	51	61	71

Table 2.1: Number of bits protected by a BCH code

a hash table of size $p \cdot 160000$. That is, in a system with $256 \cdot 160000 \approx 2^{25.3}$ entries 4 bits are protected. In this case, all positions in any game tree of depth 1 containing only normal moves have different hash values. BCH codes, therefore, support scalability by providing higher security and thus guaranteeing less collisions in larger parallel systems, which search larger game trees.

The use of BCH hash functions becomes more important the larger the hash table is. However, much more important than the theoretical bounds from Table 2.1, is the good behaviour of BCH hash functions in practice.

It is not necessary to store the representation of a chess position explicitly in the program. The hash value of a successor $v.j$ of a node v can efficiently be computed from the hash value of the node v and the move from v to $v.j$ by some few XOR-operations.

Generating a BCH-key

There is another advantage of BCH codes: In order to identify the position associated with an entry of the hash table after reading the entry, in usual hashing methods every entry contains a unique key for the position, which is stored in the entry. Above we described the construction of a unique key consisting of 749 bits. However, comparing 749 bits when an entry is read from the hash table slows down the access to the hash table unnecessarily. Suppose with 256 processors a search tree of depth 8 is to be searched. In this game tree the hash table will be accessed only for nodes in level 0 to 7, i.e. every position the hash table is accessed for can be reached from the root of the game tree by playing at most 7 plies. Suppose, hypothetically, that the game tree contains only normal moves. Then the representations of all the positions the hash table is accessed for differ by, at most, 28 bits. In order to guarantee that these positions are mapped without any collision, a BCH code with 141 code bits would be necessary. In a hash table with 2^{25} entries, only $2^{141-25} = 2^{116}$ bits for the key are necessary to identify any position the hash table is accessed for. These bits are stored as the key in the hash table entry. For our chess program we reduced these 116 bits to 30 bits, because the program ran for about one year using a 60 bit key, when a 30 bit key would have been enough to avoid any faulty identifications.

With the help of a BCH hash function the hash table is accessed for storing and reading during a search in a game tree of depth d for all nodes in depths $< d$. The following two sections describe the accesses of storing and reading.

Storing an Entry

After the Scout-algorithm from Figure 2.2, respectively Figures 2.3 and 2.4, has computed a value x and a best move z for a node v with a search depth $t \geq 1$ below v, the algorithm tries

to store the results. With the help of a BCH hash function

$$\begin{array}{rcl} h: \ \{v \mid v \text{ is a chess position}\} & \rightarrow & \{0,\ldots,max-1\} \times \{0,1\}^{30} \\ v & \rightarrow & (index, key) \end{array}$$

a hashing index as well as an identification key is computed.

The procedure "store TR-entry" in Figure 2.6 describes, how results of the search in subtrees are stored into the transposition table.

If the corresponding entry is empty, then the result (x, z, t) for the position having key s is stored in the entry together with an appropriate flag

$$k \in \{\text{exact value, upper bound, lower bound}\}$$

and a time stamp c. If the entry is not empty, then the new result overwrites the old result, if the search depth t of the new result is larger than the search depth of the old result. If this is not the case, then the new result overwrites the old one, when the time stamp of the new result is larger than the time stamp of the old one. During the computation of a single test position only one time stamp is used. We would like to conclude that our transposition table is organized in a depth-oriented manner without collision resolution.

The identification key of an entry does not provide a unique key for a node v, because there are much more than 2^{30} legal chess positions. However, it is sufficient, to reduce the probability of a collision to a very small number. The probability of a collision is furthermore reduced by the use of a BCH hash function, since the BCH hash function guarantees small subtrees to be free of collisions.

The procedure "TR-put" from Figure 2.6 composes a transposition table entry from the search window and the value computed for a node v_t. To include storing of transposition table entries into the search algorithm every call of the procedure "upwards" in procedure "iterate" of Figure 2.3 is replaced by the third algorithm of Figure 2.6.

Reading an Entry

Before the search below an inner node v_t is started, the search algorithm accesses the transposition table to confirm, whether it contains results computed earlier for the node v_t or not.

Procedure "TR-get" in Figure 2.7 describes how a result obtained from the transposition table is used to speed up the search. After computing the index ix_t and the key s_t of v_t a call of the procedure "read TR-entry" returns a transposition table entry $(s, x, k, z, t, c) = T_{ix_t}$. This entry is interpreted to contain an entry for v_t, if $s = s_t$.

How to use a transposition table entry

If a node v_t is to be searched with a search depth of $t' > t$, then the first successor generated is the one, which is reached via the move z, if z is not a fail move. z is called fail move, if v_t is a MAX-node and x is an upper bound or v_t is a MIN-node and x is a lower bound. The search below v_t continues. Through this, the transposition table, together with the iterative deepening heuristic which will be described later, provides the most important method for the sorting of moves.

```
procedure store TR-entry(ix : index; (s, x, k, z, t, c) : entry);
/* Let T be the transposition table, let T_i be the ith entry of T */
begin
    if T_ix is empty then T_ix := (s, x, k, z, t, c)
    else if t > T_ix.t then T_ix := (s, x, k, z, t, c)
         else if c > T_ix.c then T_ix := (s, x, k, z, t, c);
end;


procedure TR-put(ix : index; s : key);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search windows at depth i,
   let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables for depth i,
   let z be the move from v_t to the "best" successor,
   let c be the actual time stamp,
   let bfd be the search depth, to which the game tree has to be searched.
   Let h(v_t) = (ix, s) be already computed (in "downwards" to v_t). */
var k : flag;
begin
    if t < bfd then
    begin
        if val_t ≤ α_t then k := upper bound;
        else if val_t ≥ β_t then k := lower bound;
             else k := exact value;
        store TR-entry(ix, (s, val_t, k, z, bfd − t, c));
    end
end;


/*** In "iterate": ***/
/* Let (v_r, ..., v_t) be the actual variation, let h(v_t) = (ix_t, s_t). */
if cutoff or all successors of v_t are evaluated then
begin
    TR-put(ix_t, s_t); upwards(t);
end;
```

Figure 2.6: Storing entries of the transposition table

```
procedure read TR-entry(ix : index; var (s,x,k,z,t,c) : entry);
/* Let T be the transposition table, let T_i, 0 ≤ i < max be the ith entry of T */
begin (s,x,k,z,t,c) := (T_ix.s, T_ix.x, T_ix.k, T_ix.z, T_ix.t, T_ix.c); end;


procedure TR-get(t : depth);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search window at depth i,
   let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables in depth i,
   let bfd be the search depth, to which the game tree is to be searched.
   Let T be the transposition table, let T_i, 0 ≤ i < max be the ith entry of T */
var (s,x,k,z,t,c) : entry;
begin
    if bfd − t > 0 then
    begin
      (ix_t, s_t) := h(v_t); read TR-entry(ix_t, (s,x,k,z,t,c));
      if s = s_t then                                        /* Entry found for v_t */
      begin /* Let v_t.j be the successor of v_t, which is reached from v_t by playing z */
          if (bfd − t <> t) and z is not a fail move then    /* resort successors */
            (v_t.1, ..., v_t.b_t) := (v_t.j, v_t.1, v_t.2, ..., v_t.j − 1, v_t.j + 1, ..., v_t.b_t);
          if bfd − t = t then
          begin
              if k = exact value then                        /* stop searching below v_t */
              begin val_t := x; upwards(t); end else
              begin
                  if k = upper bound then                    /* update window */
                    begin β_t := min(β_t, x); high_t := min(high_t, x); end;
                  if k = lower bound then                    /* update window */
                    begin α_t := max(α_t, x); low_t := max(low_t, x); end;
                  if z is not a fail move then               /* resort successors */
                    (v_t.1, ..., v_t.b_t) := (v_t.j, v_t.1, v_t.2, ..., v_t.j − 1, v_t.j + 1, ..., v_t.b_t);
              end;
              if (low_t ≥ β_t) or (high_t ≤ α_t) then upwards(t);   /* cutoff */
          end
      end
    end
end;

/*** In "iterate": ***/
/* Let (v_r, ..., v_{t−1}) be the actual variation. */
if b_t = 0 then begin ... end else begin downwards(...); TR-get(t); end;
```

Figure 2.7: Reading entries from the transposition table

If the search depth $t' = t$, then the search of the subtree with root v_t can be saved, if x is an exact value. If x is an upper bound, then the β-bound of the search window of v_t is changed to $\beta := min(\beta, x)$, if x is a lower bound, then α is changed to $\alpha := max(\alpha, x)$. The first successor generated is the one which is reached from v via the move z, if z is not a fail move. The smaller search windows speed up the search in the subtree below v_t. Choosing the first successor often guarantees a good ordering of the successors.

If $t' < t$, then there is already a result with larger search depth than necessary available for the node v_t. During a game, this result will be used as described in the case $t' = t$ with pleasure, because the larger search depth makes it even more valuable for the search than the result the algorithm is going to compute. During a game this case happens quite often, because the transposition table is not cleared between two moves played. During the evaluation of a test position, however, this case happens rather rarely. Furthermore, the result of a parallel search may depend on whether a transposition table entry is already available or not, which may change with the numbers of processors used.

Therefore, using the transposition table entries with larger search depth than required, makes it impossible to control the correctness of the parallel program by looking at the results of the searches and comparing them to the results produced by the sequential version. Thereby, whenever the program is searching test positions, entries with $t' < t$ are used just like entries with $t' > t$. This holds for all experiments referred to in this work.

Mysliwietz [50] shows that the number of nodes searched when evaluating our set of test positions remains almost unchanged by this restriction. For special end game positions, however, he shows that by using the entries with larger search depth the efficiency as well as the playing strength may be improved significantly.

If every call of the procedure "downwards" in procedure "iterate" of Figure 2.3 is replaced by the algorithm at the bottom of Figure 2.7, then the transposition table is accessed for every node v_t above the leaves before the search of the subtree with root v_t is started.

2.2.2 Iterative Deepening

Instead of directly searching to a large search depth d, it often helps to search iteratively to a search depth of $1, 2, 3, \ldots$. The following are some good reasons to do so:

1. The results computed during the searches to a small search depth are used to improve the move ordering for the searches to larger search depths. In order to do so the transposition table is of great importance: Positions corresponding to inner game tree nodes during the searches to large depths often have been evaluated to smaller depths before. The best moves computed with smaller search depths often remain the best during the deeper searches. Thus, the method of iterative deepening, together with the use of the transposition table, improves the move ordering by, in many cases, sorting the best move to the top of the move list. Therefore, during the deeper searches, the game trees that have to be searched are very close to the minimal game tree, and thus are searched very quickly.

2. In a match under tournament conditions the method of iterative deepening simplifies the time control. Since the game trees to be searched are very well ordered, the time spent during the first iterations of the iterative deepening often provides a good approximation

of the time necessary to search one ply deeper. On the basis of this approximation it is much easier to make a decision whether or not to search the next iteration.

3. Even if in some cases the above mentioned approximation is wrong and the iteration can not be finished in time, the program is able to play the best move computed in the previous iteration after cancelling the search.

Since the size of the game trees that have to be searched grows exponentially in the search depth, the method of iterative deepening can be used without introducing large overhead. Without the move ordering information computed in previous iterations, however, the move ordering is much worse. Therefore the method of iterative deepening is of high importance for the efficiency of the sequential search.

2.2.3 Aspiration Search

In many cases, especially when using the method of iterative deepening, the value of the root of the game tree can be estimated quite exactly before the search starts. If the value of the root is approximated to be x, then an initial search window $[x - \varepsilon_u, x + \varepsilon_o]$ is used for the root of the game tree. The procedure *scout* is then called for the root of the game tree by

$$y := scout(\varepsilon, x - \varepsilon_u, x + \varepsilon_o).$$

If $x - \varepsilon_u < y < x + \varepsilon_o$ then y has been computed exactly. If $y \leq x - \varepsilon_u$ or $y \geq x + \varepsilon_o$ resp., then the whole game tree has to be researched with the initial search window $[-\infty, y]$ resp. $[y, \infty]$ to compute the real minmax value of the root.

In our chess program the values of the root have often been observed to vary depending on whether the search depth is odd or even. Therefore, the initial search window for the jth iteration is chosen as follows:

Let F_{j-1} be the value of the root of the game tree computed in a $j-1$ ply search. The search window for the jth iteration is

$$\begin{array}{ll} [F_{j-1} - 0.1, F_{j-1} + 0.25] & \text{if } j \text{ is odd} \\ [F_{j-1} - 0.25, F_{j-1} + 0.1] & \text{if } j \text{ is even} \end{array}$$

Here 0.1 is one tenth of the material value of a pawn, 0.25 is a quarter of a pawn. One reason for the variation observed in the values of the root is, that the player making the last ply on the principal variation often is able to improve its position slightly by making the ply, while the opponent is not allowed to counter this improvement by simply making another ply.

2.2.4 Killer Heuristic

The killer heuristic is based on the observation, that in the same level of the game tree the best move at different nodes often is the same. To recognize these moves and to put them at the top of the move list, two so called killer moves are stored in a killer list for every level of the game tree. In our program killer lists must not contain capturing moves, because these are sorted to the top of the move lists anyway.

For every killer move a bonus is stored. This bonus is used to keep the killer lists sorted as suggested by Slate and Atkin [69]. The algorithm "update killer" is shown in Figure 2.8.

```
procedure update killer move(t : level; z : move);
/* Let K_t be the killer list of level t,
   let K_t1 and K_t2 be the two elements of K_t */
begin
    if (cutoff in level t) and (z is not a capturing move) then
    begin
        if K_t1.move = z then K_t1.bonus := K_t1.bonus + 1 else
        if K_t2.move = z then
        begin
            K_t2.bonus := K_t2.bonus + 1;
            if K_t2.bonus > K_t1.bonus then exchange K_t1 with K_t2;
        end else
        begin
            K_t2.move := z; K_t2.bonus := 1;
        end
    end
end;


/*** In "iterate" ***/
/* Let (v_r, ..., v_t) be the actual variation,
   let z be the move from v_t to the best successor of v_t. */
begin
    update killer move(t, z); upwards(...);
end;
```

Figure 2.8: Update of the killer lists

If a cutoff has been computed at some node v_t in level t by the search of a subtree rooted at some successor $v_t.j$ of v_t and if $v_t.j$ is reached from v_t by a noncapturing move z, then the following algorithm updates the killer list of level t: Check, whether z is already contained in the killer list. If so, then increment the bonus of z by one. If z has been at position two in the killer list, then resort the killer list if necessary. If z is not contained in the killer list, then insert z at position two with a bonus of one.

Thus, the algorithm first has the possibility to store a move at position one of the killer list, which produced a cutoff in very many cases. Storing this move makes sense, because it will probably produce further cutoffs in the unsearched part of the game tree. Second, the algorithm is able to store a killer move at position two of the killer list, that cuts off some subtrees only in a small part of the whole game tree. Such a move may be replaced quite soon by another killer move after leaving the special part of the game tree.

2.2.5 Principal Variation and Refutation Heuristic

By the method of iterative deepening at the beginning of a d ply search, $d > 1$, the principal variation of length $d-1$ is known from the previous iteration. This principal variation is searched as the leftmost variation during the d ply search. Furthermore, during the $d-1$ ply search a variation has been computed for every successor of the root not on the principal variation, which refutes this successor compared to the successor of the root on the principal variation. The first k plies of these variations, the so called refutation variations, are used to refute the successors of the root during the d ply search as soon as possible by making them the leftmost variations of the corresponding subtrees of the root. This is done in the hope that the moves of the refutation variations will suffice to refute the successors of the root even during the d ply search. In our program $k = 6$.

The principal variation heuristic as well as the refutation heuristic in some positions propose a move that should be placed at the top of the move list, because during the search to a smaller depth it lead to the best successor. Together with the use of a large transposition table the effect of these heuristics is very small. Moves proposed by the refutation heuristic for a successor of the root are almost certainly contained in the transposition table, because collision resolving is done in a depth oriented manner using a BCH hash function. The second ply of a refutation variation in most cases consists of a fail move, i.e. it is not even guaranteed to be the best move at the corresponding position at level two during the $d-1$ ply search.

2.2.6 History Heuristic

The transposition table, the killer heuristic and the refutation heuristic propose moves, which appeared to be good moves in some position or some level of the game tree resp., to be sorted to the top of the move list. In general, however, only very few moves are captured by these heuristics. All the other moves are then sorted with the help of the history heuristic. For the history heuristic it is assumed that a move is specified uniquely by the from-square a and the to-square b. A history table, H, then is a 64×64 matrix, in which bonuses are stored for moves from a to b. Initially, every entry of H is set to 0. If a move z from a to b causes a cutoff at some node of the game tree, the entry (a, b) of H is incremented by one. Similarly, (a, b) is incremented by one, if z is found to be the best move at some node v, as long as for this node an exact value is computed.

Figure 2.9 contains a procedure "update history" for updating the history table, as well as the call of this procedure in "iterate".

2.2.7 Move Ordering

With the help of the above mentioned heuristics the successors of a node v are now ordered in the following manner:

1. If a principal variation or refutation variation is available for v, then the first successor to be searched is the successor proposed by the principal variation or refutation variation.

2. If in the transposition table an entry (s, x, k, z, t, c) is found for v, then the successor of v, which is reached via move z, is searched first if z is not a fail move, i.e. v is a MAX-node

```
procedure update history(z : move);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search windows at level i,
   let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables of level i,
   Let H be the history table,
   let H_{i,j}, 1 ≤ i, j ≤ 64 be the elements of H */
var fail search: boolean;
begin
    if v_t is MAX-node then fail search:= val_t ≤ α
                       else fail search:= val_t ≥ β;
    if not fail search then
      /* Let a be the from-square of z and b be the to-square of z */
      H_{a,b} := H_{a,b} + 1;
end;


/*** In "iterate" ***/
/* Let (v_r, ..., v_t) be the actual variation,
   let z be the move from v_t to the best successor of v_t. */
begin
    update history(z); upwards(...);
end;
```

Figure 2.9: Update of the history table

and k = "upper bound" or v is a MIN-node and k = "lower bound". In these cases the entry for v results from a fail search, such that the best successor of v was not determined during this search. Using the move z will often result in searching a bad successor first. In some experiments it turned out that the sequential algorithm visits a few more nodes, if these fail moves are searched first.

3. After the successor proposed by the transposition table all successors are searched, which are reached via a *positive capture* move. A move is called a positive capture move, if it is a capture move and a major piece is captured or an undefended piece is captured or a piece, that is not defended enough, is captured. For example, a move "Knight takes Bishop" is a positive capture. These moves are sorted in a decreasing manner according to the value of the captured pieces. Different moves capturing pieces of the same value are sorted such that moves with minor capturing pieces are searched first.

   The positive capture moves seem to offer at least material balance. Of course, this material balance is not guaranteed, because notations like "not defended enough" can only be approximated. The resulting move ordering of the capturing moves is usual in state of the art chess programs. It has the advantage that the moves can be generated in the described order, which saves sorting the moves.

4. After the successors reached via positive captures, those reached via killer moves are investigated, as long as the killer moves are legal in position v. Here the first killer move

is investigated before the second one.

5. After the killer moves, all successors reached via nonpositive capture moves are searched. Here the move "Bishop takes Knight", for instance, is a nonpositive capture move, because in our program the bishops value is slightly higher than the knight's one. The nonpositive captures are generated in the same order as the positive captures.

6. After all the above mentioned successors have been searched, the remaining legal moves are sorted in an increasing manner according to their history values. The corresponding successors are searched in the resulting order.

2.3 Implementation Details

Many details of the implementation of the sequential search algorithm are omitted in the descriptions above. Most of these details are unimportant for the structure of the game tree to be searched. Some of them, however, like search extensions or search depth reductions as used in many of the state of the art game playing programs, largely influence the structure of the game tree to be searched, and therefore will be mentioned here. A more detailed description of the conditions for search extensions or search depth reductions used in our program is presented by Mysliwietz [50].

Search Extensions

In many chess positions it makes sense to evaluate the subtree below this position deeper than indicated by the remaining search depth for the position. Especially, if the player to move is in a dangerous situation and the outcome of this situation can not be approximated statically. An extreme example of such a situation is a leaf of the game tree, in which the player to move has material advantage but is in check. Evaluated statically, the leaf may receive a high value for the player to move, which, however, may be completely unrealistic, because the side to move is mated It is possible to evaluate a mate statically, however, for this example, a mate shall be detected by being checked without having any legal move. Of course it makes sense to search a node like this deeper, instead of evaluating its value statically. Search extensions are introduced into the program in order to give the computer the possibility to evaluate the outcome of the dangerous situation better than possible by the static evaluation function. The search depth at a leaf like the above described one is increased by one.

Search extensions are not only useful at leaves of the game tree. Even at inner nodes, in which the side to move is in a dangerous situation, it may be helpful to evaluate the corresponding subtree one ply deeper, in order to approximate the outcome of the dangerous situation better. The search depth for these nodes is increased by one, as well.

Search extensions in our current chess program are domain dependent and therefore independent of the number of processors, the information contained in the heuristics and the search windows used.

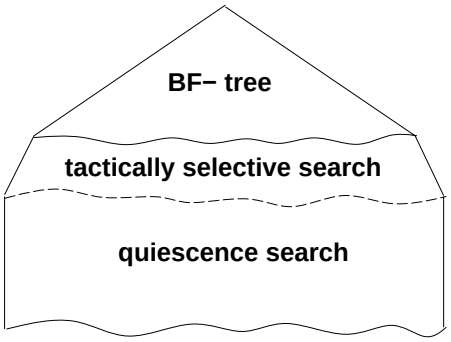


Figure 2.10: Search phases

Search Depth Reductions

Just as it may make sense to extend the search at a node, it may make sense to reduce the search depth for positions, which are reached via some "obviously bad" move. These search depth reductions help to avoid searching too many nodes in subtrees, which, with high probability, are irrelevant for the value of the root. As an example of an obviously bad move we consider a noncapturing, nonchecking move, which puts a major piece to a square, where it is not defended but attacked by some opponents piece. However, if the search to a reduced search depth returns a value, which is included in the search window, the corresponding subtree has to be researched with the full search depth. Thus, the results of subproblems depend on the search windows available for the actual nodes and, therefore, depend on how far parallelism is used to evaluate this node. Checking the correctness of a parallel program, however, requires, that the sequential and the parallel algorithm return the same result, i.e. the same minmax value of the root, independently of the number of processors.

Therefore, for the experiments in this work search depth reductions are not used.

Brute Force Search, Selective Search, Quiescence Search

In chess programs today, it is usual to search only a few successors of the inner nodes at a larger level of the game tree, even if no cutoff is computed at that node. In the upper part of the tree, the so called *brute force tree*, all successors of inner nodes are searched. The part below the brute force tree is denoted by *selective search tree*. In this part, only those moves are searched which look interesting "at a first glance". In a third phase, called *quiescence search*, only the moves leading to more quiet positions, i.e. the moves resolving tactical difficulties, are investigated. Capturing moves, especially, are searched during the quiescence search. Some of the moves creating very heavy threads, may be considered also.

During the selective search phase and the quiescence search, important moves may be overlooked by the static procedure while deciding which move to search and which not. This leads to an erroneous evaluation of a position, an effect widely known as the *horizon effect*. Even commercial top machines suffer heavily from horizon effects. The mistakes made are the worse, the higher they happen in the game tree, since the opponent has the best chance to recognize them at this point.

In our chess program we use selective search very rarely: Instead of doing a pure quiescence search at the leaves of the brute force tree, we extend the quiescence search by including some tactical threads like knights or pawns forks, moves of free pawns to the 7th or 8th rank, etc, at the leaves of the brute force tree as well as during the next three plies of the quiescence search. Moves, which might be important for the positional value of the positions were, at the time of writing, not included in the quiescence search, such that for the program version presented in this work, one should talk about a tactically selective search or an extended quiescence search respectively. Therefore, in the following all the nodes of the selective search tree and of the quiescence search tree are denoted *quiescence search nodes*. Figure 2.10 shows the different search phases.

Characteristics of the Quiescence Search

The searches at the inner nodes of the brute force tree differ from the searches at the leaves of the brute force tree and the quiescence search nodes in another important point. At the leaves of the brute force tree and at the quiescence search nodes a cutoff may be computed without searching any successor. Before the first successor is generated, the static evaluation of such a node is computed. This value is used as if it were the exact value of a successor. By this we obtain a dramatic reduction of the size of the quiescence search trees.

This method, however, has the disadvantage that the static evaluation function may return an erroneous value, as described above. Therefore, the use of the static evaluation function to produce additional cutoffs is not allowed, if the player to move has to face a serious threat. By this, not all, but many of the erroneous evaluations are avoided. In most of the quiescence search nodes and brute force leaves the use of the static evaluation is allowed. We note the following

Observation 2.1 *Subtrees of the quiescence search trees differ in the following two points from subtrees of the brute force tree:*

1. *They are more narrow and more irregular in depth.*
2. *Cutoffs may be obtained without searching any successor by simply computing the static evaluation function of a node.*

2.4 Hardware for the Sequential Algorithm

For most of our experiments we use the T800 Transputer of Inmos. The T800 is a Risc processor with four bidirectional links, which may be connected to other Transputers. The transmission rate is 20 Mbit/second simultaneously in both directions.

Figure 2.11 shows the T800 Transputer. Each processor has 4 MByte main memory. The clock rate is 25 MHz. According to information from the manufacturers from 1989, the T800 is able to run at a speed of 12.5 MIPS and 1.5 MFLOPS. It should be noted here, that our chess program *ZUGZWANG* does not use any floating point operations.

The Transputer supports the scheduling of quasi-parallel processes by a hardware scheduler. With the help of this hardware scheduler the CPU is able to efficiently run quasi-parallel processes and to communicate via all four links in both directions in parallel.

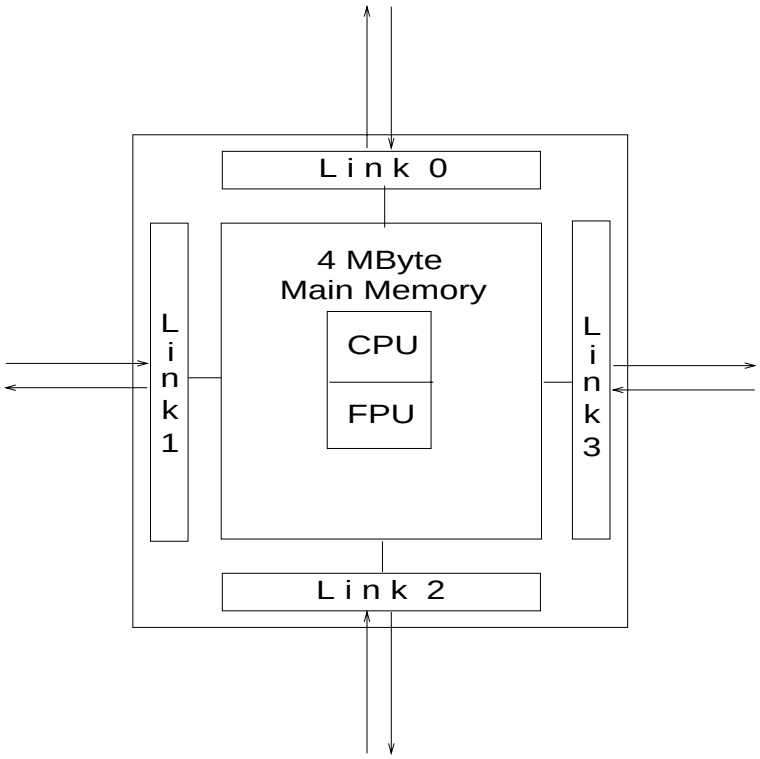


Figure 2.11: Schematic description of a Transputer

In Chapter 5 of this work we will present experiments which are based on the T805 Transputer. The T805 differs from the T800 by the increased clock rate of 30 MHz, and thus runs at a speed which is exactly $\frac{6}{5}$ times the speed of the T800.

Both Transputers support the use of the programming language Occam, which is used for the implementation of *ZUGZWANG*. A description of Occam can be found in the book of Burns [14]. For the description of parts of the program in this work, however, the use of Occam constructs was avoided.

2.5 Performance of the Sequential Algorithm

In this section we will show, by using various statistics, that the sequential version of our chess program, *ZUGZWANG*, with the help of the heuristics described in Section 2.2, searches game trees very efficiently. Furthermore, we will describe data from experiments, which will be the basis for speedup measurements in Chapter 4.

2.5.1 Description of the Set of Test Positions

In all experiments presented in this work we will use the 24 positions $B01, \ldots, B24$ of the Bratko Kopec test [13] as test input for our algorithm. Originally these positions were used to check the playing strength of chess programs. Included are a mating problem, $B01$, positions

from the opening phase, e.g. $B04$, from the middle game, e.g. $B19$, and from the endgame, e.g. $B08$. In Section B of the Appendix the 24 positions of the Bratko Kopec test are shown.

The best move computed by our program in an 8-ply search differs in 5 positions ($B04$, $B09$, $B13$, $B17$ and $B24$) from the best move found during a 7-ply search. This corresponds to a rate of 20.8%. In ten subsequent games against other computers, *ZUGZWANG* changed the best move during the last iteration in 17.5% of the moves. In ten subsequent games against humans rated around 2300 ELO it changed the best move in the last iteration in 24.3%. Thus, the positions of the Bratko Kopec test set can be regarded to be representative with respect to the ordering of the successors of the root.

In Chapter 4 we will discuss the results of our algorithm obtained from the 3- to 8- and 9*-ply searches. Here, searching to a search depth of 9* means searching the positions $B04$, $B05$, $B09$, $B13$ and $B14$ with a search depth of 8 and all the other positions with a search depth of 9. $B04$, $B05$, $B09$ and $B14$ will have the longest running times of an 8-ply search with 256 processors, a 9-ply search of position $B13$ could not be finished without an overflow of some statistics, such that no statistical data is available for a sequential 9-ply search of $B13$.

2.5.2 Data for Different Search Depths

Most of the experiments presented in this work were done for 8-ply searches on the Bratko Kopec positions. Table 2.2 contains the results of the sequential Scout-algorithm for 8-ply searches on the 24 positions $B01$ to $B24$.

The lower part of Table 2.2 contains, the results for the search depths 5, 6, 7 and 9*, summed up for all the 24 positions. The second column of Table 2.2 contains the move computed for the corresponding position. It is marked with a "!", if it is a solution of the position. The third column contains the minmax value of the root. Here 1000 points is the material value of a pawn. The fourth column shows, whether this move was already the best move of the 7-ply search (Y). In these positions, the first successor of the root is the best successor of the 8-ply search. The fifth column gives the depth of the deepest nodes visited. The sixth column contains the number of nodes visited in the brute force tree including the leaves. The seventh column contains the number of nodes visited below the leaves of the brute force tree. The eighth column gives the time needed for the 8-ply search in seconds.

Observation 2.2 *During an* 8*-ply search the sequential Scout-algorithm visits roughly* 523 *per second on the average.*

2.5.3 Scout vs. $\alpha\beta$

The data contained in Table 2.3 results from 3- to 8-ply searches of the sequential $\alpha\beta$-algorithm instead of the Scout-algorithm. We note:

Observation 2.3 *The $\alpha\beta$-algorithm visits between* 1.0% *to* 8.3% *more nodes and needs between* 1.3% *and* 9.7% *more time than the Scout-algorithm.*

One reason for the decreasing advantage for deeper searches of the Scout-algorithm compared to the $\alpha\beta$-algorithm may be given by Observation 2.8 on page 42.

Dpth 1-8	Move	Value	First	Maxdpth	#BFT-nodes	#QS-nodes	time
B01	d6-d1!	99999	Y	11	281	380	2
B02	e4-e5	415	Y	24	1737779	1760701	5972
B03	f6-f5!	60	Y	28	3223438	2658858	11255
B04	e5-e6!	250	N	31	7530789	54260945	120014
B05	c3-d5!	1300	Y	28	12023781	20279553	60498
B06	g5-g6!	710	Y	18	394980	95593	801
B07	h5-f6!	799	Y	36	3438436	7676069	22613
B08	f4-f5!	310	Y	22	170786	87537	412
B09	f1-d3	2200	N	28	6263076	14421095	38170
B10	c6-e5!	1060	Y	27	5242305	7218911	24214
B11	f2-f4!	320	Y	33	3305353	3397519	13516
B12	d7-f5!	200	Y	26	2827920	3112995	11202
B13	a1-c1	250	N	24	5933214	2958740	18171
B14	d1-d2!	3080	Y	34	9354003	24028034	72628
B15	g4-g7!	700	Y	29	3641986	22193307	43595
B16	d2-e4!	890	Y	28	4136781	4398220	16100
B17	h7-h6	-170	N	25	6267754	2597581	16551
B18	f7-f5	230	Y	31	8112799	13597223	41682
B19	e8-e4!	520	Y	26	2524680	2668685	9496
B20	c3-b5	390	Y	28	3750786	2993856	13273
B21	f5-h6!	1580	Y	30	7711003	14252825	36551
B22	b7-e4!	820	Y	29	5306579	6969017	23428
B23	c8-f5	89	Y	30	6146130	6206720	24523
B24	f2-f4!	780	N	26	6407798	12286766	35562
Σ	17 (11 + 6)				115452437	230121130	660229
Σ : 1-5					1133692	894666	3915
Σ : 1-6					4222551	5286520	18409
Σ : 1-7					27762697	27733157	101046
Σ : 1-9*					582996318	853974974	2621294

Table 2.2: Sequential search with the Scout-algorithm

	#BFT-nodes	#QS-nodes	time
Σ : 1-5	1143287	1053642	4296
Σ : 1-6	4249198	5878815	19799
Σ : 1-7	28050266	30315035	107009
Σ : 1-8	117840193	231202446	668893

Table 2.3: Sequential search with the $\alpha\beta$-algorithm

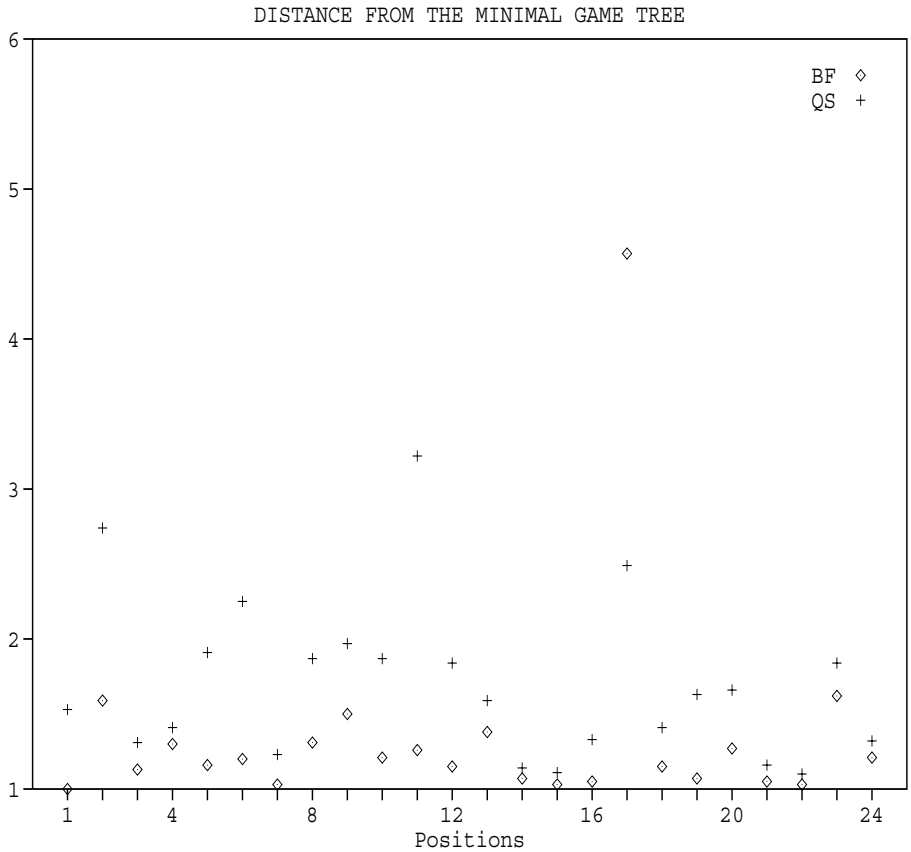


Figure 2.12: Distance from the minimal game tree: Scout, 8-ply search

2.5.4 Distance from the Minimal Game Tree

One measure for the efficiency of a sequential game tree search algorithm, is the size of the game tree visited compared to the size of the minimal game tree. In order to measure the size of the minimal game tree we use the definition of the *type* of a node for the nodes of the minimal game tree from Knuth and Moore [40]:

Definition 2.5 *(Type of nodes)*

Let $G = (V, E, f)$ be a game tree with root ε. Then the partial mapping

$$type : V \rightarrow \{1, 2, 3\}$$

is defined by

$$\begin{array}{ll} type(\varepsilon) := 1 & \\ type(v.j) := 1 & \textit{if } type(v) = 1 \textit{ and } j = 1 \\ type(v.j) := 2 & \textit{if } type(v) = 1 \textit{ and } j > 1 \\ type(v.j) := 3 & \textit{if } type(v) = 2 \textit{ and } j = 1 \\ type(v.j) := 2 & \textit{if } type(v) = 3 \end{array}$$

Otherwise type is undefined.

Through the mapping *type* exactly the nodes of the minimal game tree of G get associated a type.

Definition 2.6 *(minimal game tree)*

The minimal game tree $M = (V_M, E_M, f)$ of a game tree $G = (V, E, f)$ is defined to be

$$\begin{aligned} V_M &:= \{v \in V \mid type(v) \text{ is defined}\} \text{ and} \\ E_M &:= \{(u, v) \in E \mid u, v \in V_M\} \end{aligned}$$

Slagle and Dixon [68] were the first to show that the minimal game tree of a b/d - uniform game tree has exactly $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ leaves. Knuth and Moore [40] showed that the minimal game tree can be identified by using definition 2.6. We will use this definition to count the nodes of the minimal game tree visited during the search of the game tree.

Definition 2.7 *(Distance from the minimal game tree)*

Let $G = (V, E, f)$ be the game tree visited by the sequential game tree search algorithm and let M be its minimal game tree according to definition 2.6. Let m be the number of nodes of the minimal game tree visited and let n be the total number of nodes visited. Here nodes visited more than once due to researches are counted every time they are visited. Then we define

$$\Delta_{min} := \frac{n}{m}$$

to be the distance from the minimal game tree.

In the case of a research at a node v the minimal subtree of the subtree rooted at v in part may be visited twice and will then be counted twice for the computation of Δ_{min}. It always holds $\Delta_{min} \geq 1$.

Figure 2.12 shows the distance from the minimal game tree for an 8-ply search of the Bratko Kopec positions using the Scout-algorithm, separated in the brute force parts and the quiescence search parts of the game trees searched. It becomes clear, that the distance from the minimal game tree, with the exception of $B17$, is smaller in the brute force tree than in the quiescence search. The reason for this is the use of the transposition table, which allows very good move ordering in the brute force part of the tree. However, it is not used in the quiescence search. The move ordering of the brute force tree of $B17$ is very poor, because the best move changes several times from depth to depth. The average distance from the minimal game tree is 1.23 in the brute force tree and 1.40 in the quiescence search. We note:

Observation 2.4 *The average distances from the minimal game tree of* 1.23 *and* 1.40 *resp. show, that the sequential algorithm generates very well ordered search trees. The only significant exception in the Bratko Kopec set of positions is position $B17$ having distances of* 3.70 *and* 2.07*resp.*

The difference in the ordering of the successors of nodes in the brute force search and the quiescence search is expressed also in the statistics about cutoffs in Table 2.4. The first two lines show, that 96.91% of all the cutoffs are caused by the first successor of a node, if this successor is an inner node or a leaf of the brute force tree. In the quiescence search tree 94.90% of the cutoffs

	Cutoffs by any successor				
%	1.	Pos	Kil	Neg	Norm
BFT	96.91	0.41	1.28	0.75	0.65
QST	94.90	2.14	0.83	1.31	0.82

	Cutoffs by right successors			
%	Pos	Kil	Neg	Norm
BFT	13.15	41.57	24.26	21.02
QST	42.00	16.27	25.69	16.04

Table 2.4: Cutoffs in the BFT and QST

are caused by the first successor. Here the first move may have been an arbitrary move, e.g. from the transposition table, from the killer lists, etc. In the ordering of the moves presented on page 31, the first move is usually followed by the positive captures, which in the brute force tree cause a percentage of 0.41% of all the cutoffs, and in the quiescence search even a percentage of 2.14%. The positive captures are followed by the killer moves, and the nonpositive captures. All the other nodes are summed up in the column "Norm". In the third and the fourth lines the percentages of cutoffs caused by the different sets of moves but not by the first successor are shown. It becomes clear, that in the brute force tree the moves delivered by the killer heuristic do well. The killer moves produce 41.57% of all the cutoffs caused by right successors, which is more than caused by any other class of moves. In the quiescence search tree this fraction must obviously be reduced, since in this part of the tree at almost all inner nodes only capturing moves are generated.

Observation 2.5 *During the brute force search almost* 97% *of all the cutoffs are caused by left successors. During the quiescence search this fraction reduces insignificantly to* 95%.

Observation 2.6 *During the brute force search almost* 55%, *during the quiescence search more than* 58% *of all cutoffs caused by right successors are caused by positive capture or killer moves.*

2.5.5 The Effects of the Transposition Table

Observation 2.5 states, that most of the cutoffs are achieved by evaluating only the first successor. The first successor is often reached via a move from the transposition table.

Table 2.5 contains statistics for the transposition table. It contains the percentage of successful accesses of the read- and the store-operations for the 5-ply to 8-ply searches. The columns "value", "bound" and "move" show the percentage of how often the exact value, the bound or only the move delivered from a read-operation could be used. with respect to the total number of read-accesses. The column "collisions" presents the percentage of collisions with respect to the total number of store-operations. In the case of a collision one of the old or new entry is deleted. The column "Xcoll" contains the percentage of collisions, which were to be expected, if all entries were stored independently and identically distributed in the transposition table. This expected number of collisions is defined as

	read (%)			store (%)	
search depth	value	bound	move	collisions	Xcoll
5	0.0161	17.94	18.31	2.33	1.59
6	0.0043	21.68	9.54	13.08	11.38
7	0.0018	22.28	13.58	39.39	34.15
8	0.0007	15.59	6.23	84.93	84.69

Table 2.5: Accesses to the transposition table

Definition 2.8 *(expected number of collisions)*

Let m be the number of transposition table entries, let $i_1, \ldots, i_s \in \{0, \ldots, m-1\}$ be the indices of a sequence of s store-operations for s pairwise different positions. Then the expected number of collisions e is defined by

$$e = s - m \cdot (1 - (\frac{m-1}{m})^s).$$

e is the number of collisions, which one would expect for a hash function mapping all the positions independently and identically distributed into the hash table.

Here $(\frac{m-1}{m})^s$ is the probability, that an entry of the table is still empty after s store-operations for s pairwise different positions, if the hash function maps the positions identically distributed and independently into the set of indices. Then $m \cdot (1 - (\frac{m-1}{m})^s)$ is the expected number of nonempty entries after s store-operations, and therefore e is the expected number of collisions after s store-operations.

The expected number of collisions was computed for every game tree, which was searched for the above described experiment. The sum of these numbers gives the expected number of collisions for the whole experiment.

We note the following observations.

Observation 2.7 *The BCH hash function shows a good behaviour even for a small table size of* 160000 *entries. The expected number of collisions is only insignificantly smaller than the number of collisions measured.*

Observation 2.8 *The transposition table is overloaded during an* 8*-ply search. Nearly* 85% *of the store-operations cause a collision and therefore the loss of one entry. Even during a* 7*-ply search nearly* 40% *collisions indicate a slight overload.*

The last observation may give a reason for the decrease of the advantage of the Scout-algorithm compared to the $\alpha\beta$-algorithm with larger search depths: Researches occurring in the Scout-algorithm may become inefficient with larger search depths, since the transposition table is overloaded then. The advantage of the Scout-algorithm nearly disappears.

We would like to conclude, that the sequential algorithm is a very efficient game tree searching procedure, even if the transposition table is heavily overloaded for the deeper searches. The Scout-algorithm is slightly superior to the pure $\alpha\beta$-algorithm.

Chapter 3

The Parallel Game Tree Search Algorithm

3.1 Hardware for the Parallel Algorithm

The parallel computer used for our experiments is based on the T800 resp. T805 processor already described in Section 2.4. Both processors have four links, which enable the processors to communicate with other processors. Therefore processor topologies of degree four are preferred.

With the exception of the experiments of Chapter 5 all data described in this book are measured on a Parsytec SC 320 machine. The basic part of this parallel computer is the T800 Transputer. The links of the T800 processors are connected to switches in such a way, that every communication network of degree four can be realized. A detailed description of the communication hardware can be found in the paper by Funke et al. [31].

Using this machine we are in the position to use the *DeBruijn network* as defined below.

The DeBruijn network was first described by DeBruijn [18]. It is easily defined with the help of the following permutations:

For all $n \in \mathbb{N}$ and all $v = v_n \ldots v_1 \in \{0,1\}^n$ we define

$$\begin{array}{rccc} s: & \{0,1\}^n & \rightarrow & \{0,1\}^n \\ & v_n \ldots v_1 & \rightarrow & v_{n-1} \ldots v_1 v_n \end{array}$$

to be the *shuffle* function and

$$\begin{array}{rccc} e: & \{0,1\}^n & \rightarrow & \{0,1\}^n \\ & v_n \ldots v_1 & \rightarrow & v_{n-1} \ldots v_1 \bar{v}_n \end{array}$$

to be the *exchange* function. Here $\bar{v}_n$ denotes the complement of v_n.

Definition 3.1 *(DeBruin network)*

The DeBruijn network of dimension n, $DB(n) = (V_n, E_n)$, is defined as:

$$\begin{array}{rcl} V_n & = & \{0,1\}^n \\ E_n & = & \{\{v, s(v)\}, \{v, e(v)\} \mid v \in V_n\} \end{array}$$

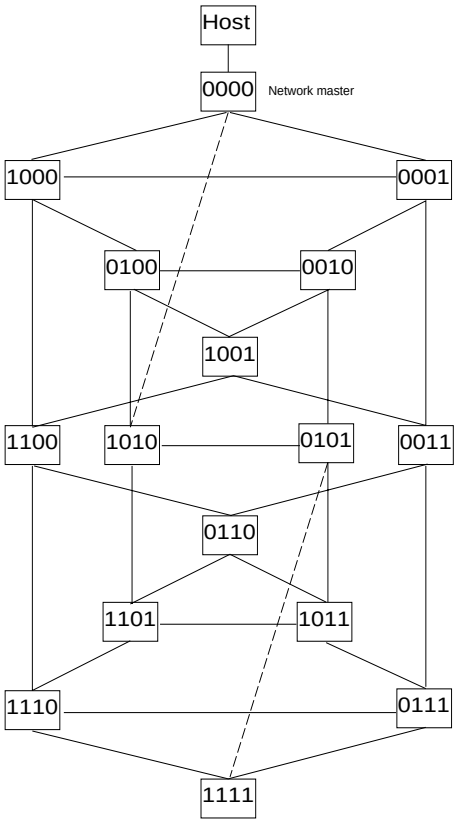


Figure 3.1: Debruijn network of dimension four: $DB(4)$

The DeBruijn network has two nodes of degree two, 0^n, and 1^n. A host processor is connected to 0^n, which is responsible only for the input and the output. 0^n becomes the network master. A DeBruijn network of dimension four is presented in Figure 3.1

Moreover, the DeBruijn network has two nodes, $1010\ldots$ and $0101\ldots$, which are connected by two edges (In Figure 3.1 only one edge is included). One of these edges may be deleted. Instead of this edge we insert two new edges from $1010\ldots$ to 0^n and from $0101\ldots$ to 1^n into the network. We get a communication structure, in which one processor, 1^n, is connected with three other processors. All the other processors have four neighbors.

The modifications described above do not influence the performance of our distributed algorithm. Therefore, we will simply use the notion "DeBruijn network" for the modified DeBruijn network.

The possibility to set up arbitrary topologies in the Parsytec SC 320 allows us to build other networks besides the DeBruijn network, such that a comparison of different communication networks is possible with the same machine.

Other than the DeBruijn topology, we also use the following topologies:

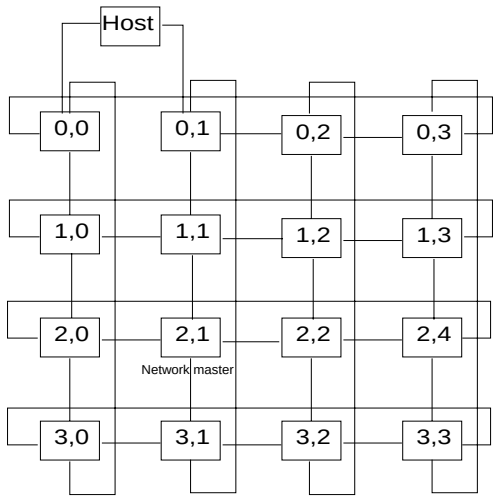


Figure 3.2: A 4×4 - torus

Definition 3.2 *(Grid, Torus)*

The $n \times m$ - grid $G(n,m) = (V_G, E_G)$ is defined as follows:

$$\begin{aligned} V_G &= \{(i,j) \mid 0 \leq i < n,\ 0 \leq j < m\} \\ E_G &= \{\{(i,j),(i',j')\} \mid |i-i'| + |j-j'| = 1\} \end{aligned}$$

The $n \times m$ - torus $T(n,m) = (V_T, E_T)$ is defined as:

$$\begin{aligned} V_T &= V_G \\ E_T &= E_G \cup \{\{(0,j),(n-1,j)\},\{(i,0),(i,n-1)\} \mid 0 \leq i < n,\ 0 \leq j < m\} \end{aligned}$$

To connect the host processor to the torus, the edge between the processors $(0,0)$ and $(0,1)$ is deleted. The corresponding links are connected to the host processor. Using the grid topology, the host processor is connected to the processor $(0,0)$. A 4×4 - torus is shown in Figure 3.2, a 4×4 - grid is shown in Figure 3.3. Both networks are modified in order to connect the host processor.

3.1.1 Comparing DeBruijn Network, Torus and Grid

In the DeBruijn network the diameter Δ grows logarithmically in the number of processors of the network. Thus, the average distance Δ_Φ between two processors grows logarithmically as well.

The communication networks used in modern parallel computers are often grids or toruses. In these networks the diameter, as well as the average distance between two processors, is of order $\sqrt{N}$, where N is the number of processors used. Thus, the DeBruijn network is superior in terms of the diameter and the average distance, however, grids and toruses are used, since they have a clear structure and are easily scalable.

Table 3.1 shows the diameters and the average distances Δ, resp. Δ_Φ, of our slightly modified DeBruijn network (upper part), torus (medium part) and grid (lower part).

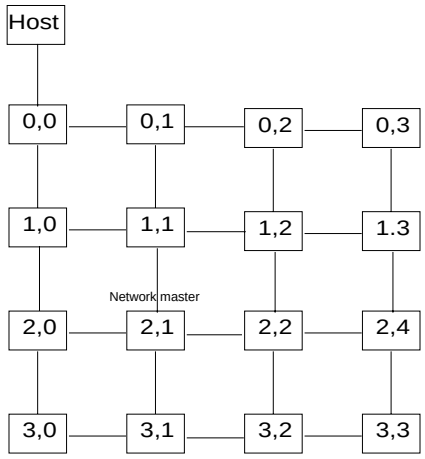


Figure 3.3: A 4×4 - grid

3.2 Related Work

In this section, we will present a short overview of other publications regarding parallel game tree search. A list of the most important results is presented in Table 3.4 on page 50.

3.2.1 Parallel Static Evaluation/Move Generation

The static evaluation function, as well as the move generation, allow parallelization. For example, all features of the static evaluation of a position may be computed in parallel, and then be combined to the final value, or all legal moves of a position may be generated in parallel. Chess machines like *BELLE* or *DEEP THOUGHT* use these possibilities with special hardware. Even a chess program like *CRAY BLITZ* implicitly uses parallel static evaluation and move generation by running the program on vector processors. A description of these machines can be found in [16], [34] and [37]. Such a parallelization leads to a fine grain parallelization, which is most useful when special hardware is used.

3.2.2 Parallel Window Search

Baudet [8] describes a parallel window search, for which the processors are searching the game tree with disjoint search windows. He notices a maximum speedup between 5 and 6 independent of the number of processors searching random trees. Observation 2.4 on page 40 shows, that by using the state-of-the-art heuristics to sort moves, the game trees searched are only slightly larger than the minimal game tree on the average, i.e. the sequential algorithm searching a b/d - uniform game tree visits

$$c \cdot (b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1\ )$$

leaves of the game tree for some small constant c. In our algorithm, $c \in [1, 2]$ for almost all game trees that are searched. A parallel window search, using n processors and disjoint search

DeBruijn	4	5	6	7	8	9	10
N	16	32	64	128	256	512	1024
Δ	4	5	6	7	8	9	10
Δ_Φ	2.0	2.7	3.4	4.2	5.0	5.8	6.7

Torus	4x4	4x8	8x8	8x16	16x16	16x32	32x32
N	16	32	64	128	256	512	1024
Δ	4	6	8	12	16	24	32
Δ_Φ	2	3	4	6	8	12	16

Grid	4x4	4x8	8x8	8x16	16x16	16x32	32x32
N	16	32	64	128	256	512	1024
Δ	6	10	14	22	30	46	62
Δ_Φ	2.5	3.9	5.3	7.9	10.6	16.0	21.3

Table 3.1: Distances in some DeBruijn, torus and grid networks

windows, visits at least

$$\underbrace{(n-1)\cdot b^{\lfloor \frac{d}{2} \rfloor}}_{n-1\ Failsearches} + \underbrace{b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1}_{minimalgametree}$$

leaves of the game tree. It follows immediately, that the parallel window search is not suited for well ordered game trees.

3.2.3 Game Tree Decomposition

The Algorithm of Finkel and Fishburn

The first algorithm to search game trees by decomposing the tree into disjoint parts and searching these parts in parallel, is due to Finkel and Fishburn [29, 30]. Their concept is based upon a game tree decomposition and a static distribution of the processors by mapping a processor tree onto the upper part of the game tree: The root of the game tree is assigned to the root of the processor tree. Every inner processor of the processor tree generates the successors of the position that is assigned to it, and keeps them in a queue to assign them successively to its successors in the processor tree. The leaf processors search the subproblems assigned to them using the sequential $\alpha\beta$-algorithm. They then return the values they have determined to their fathers in the processor tree. A new subproblem, rooted at a brother of the root of the former subproblem, then, is assigned to them, if such a subproblem exists. If only the minimal tree of a b/d - uniform game tree has to be searched, then this algorithm achieves a speedup of $O(\sqrt{n})$ using n processors. In simulations, a speedup of 5.31 was achieved using 27 processors arranged in a processor tree of depth 3 and width 3.

Mandatory Work First

Akl, Barnard and Doran [3] describe the "Mandatory Work First"-algorithm. In a first phase this algorithm searches only nodes, which have to be searched by a sequential search algorithm, which uses only direct cutoffs. Afterwards, those subproblems are searched that are not yet cut off. This succession is guaranteed by using a priority queue for processes. Akl et al. conjecture that the speedup achievable by this algorithm is, at most, 6 independent of the number of processors.

The PV-Split-Algorithm

Marsland and Campbell [45, 46] describe the PV-Split-algorithm. The algorithm makes use of the fact, that game playing programs using state-of-the-art heuristics for move ordering, in general, search very well-ordered game trees. Therefore, along the leftmost variation of the game tree, all processor are first used to search the left subtree in parallel and then to search all the right subtrees in parallel. The subproblems itself are then searched as proposed by Finkel and Fishburn [30]. Experiments by Marsland and Campbell [45] show, that the PV-Split-algorithm is up to 41% faster than the algorithm of Finkel and Fishburn [30]. Speedup measurements of Marsland and Popowich [46] resulted in a speedup of 3.2 with 4 processors using the PV-Split-algorithm.

The methods described so far are mainly of theoretical interest. They are lacking dynamic load balancing possibilities, which allow a decomposition of the game tree in accordance to its unregular structure. The PV-Split-algorithm still uses the fact that the game trees occurring in practice are very close to the minimal game tree. However, this fact is used only along the leftmost variation of the game tree. The algorithm does not make use of the fact that the right subtrees are very well-ordered, also. The PV-Split-algorithm, however, is the basis for most of the implemented parallel game tree search algorithms.

Applications of the PV-Split-Algorithm

The PV-Split-algorithm has been implemented in several game playing programs. Newborn [54] achieved a speedup of 5.03 with 8 processors. He used a uniform distribution of the successors of a type-1 node onto the processors. This distribution is corrected if a processor becomes idle, by redistributing the moves not yet searched onto the processors uniformly again. Marsland, Olafsson and Schaeffer [47] describe an implementation in the chess program *SUNPHOENIX*, which achieves a speedup of 3.75 with 4 processors. For their implementation, the authors expected a speedup of, at most, 4.4, independent of the number of processors, mainly because the load distribution is poor.

Schaeffer [64] describes the Dynamic PV-Split-algorithm. In this algorithm a control processor is allowed to dynamically change the processor tree. The processors start working according to the PV-Split-algorithm, which allows the use of parallelism along the leftmost variation of the game tree. A processor, which is assigned a subproblem with a root v, now starts the PV-Split-algorithm. Thus, parallelism can be used to evaluate the subtrees along the leftmost variation of the subtree with root v. No other parallel evaluation of subtrees is allowed. Therefore the Dynamic PV-Split-algorithm includes dynamic load distribution, however, this load distribution remains incomplete and is organized by one central processor, which is a bottle neck.

Hyatt [36] and Hyatt, Suter and Nelson [38] resp., describe a parallel algorithm based on the PV-Split-algorithm, which is called EPV-Split-algorithm. It uses the following strategy to shorten the idle times caused by processors waiting for some results of their brothers in the processor tree: If a processor becomes idle at a time, at which his father is not able to send it another subproblem, then it stops the searches of all its active brothers. All the processors are restarted in one of the active subtrees two levels deeper in the tree. This rigorous load distribution technic results in an improvement of the speedup from 4.57 to 5.93 using 16 processors, as compared to the basic PV-Split-algorithm.

Dynamic PV-Split-Algorithms

Ferguson and Korf [28] describe a parallel game tree search algorithm, which is called "Bound and Branch Algorithm". It enables dynamic load distribution by allowing the use of parallelism at a node only after a bound is computed for it. This algorithm was implemented in a checkers program. At the same time we developed the "Young Brothers Wait Concept" and used it in a chess program of minor playing strength. The results are described in the literature [21, 22, 23]. In both programs important heuristics for move ordering were not implemented. Otto and Felten [56] implemented a dynamic parallel game tree search algorithm in their chess program *WAYCOOL*, which allows parallelism at a node of the game tree only if a transposition table entry was found for that node. If in the transposition table a move is found for some node, then the corresponding successor is searched first. Parallel evaluation of successors of this node is allowed only after the search of the first successor is finished. If there is no transposition table entry available for some node, then all the successors of this node may be evaluated in parallel. Otto and Felten achieved a speedup of 101 with 256 processors, as compared to the suboptimal sequential $\alpha\beta$-algorithm. The results of this work are improvements of the parallel game tree search algorithm described earlier [24, 26].

Parallel SSS*

The SSS*-algorithm is a Branch & Bound algorithm for game tree search. It was developed by Stockman [70]. The SSS*-algorithm does not generate any node of the game tree, that is not generated by the $\alpha\beta$-algorithm, however, just as all Branch & Bound algorithms, the space requirement of this algorithm is enormous. Roizen and Pearl [61] show, that the SSS*-algorithm asymptotically has the same time complexity as the $\alpha\beta$-algorithm. Vornberger and Monien [75] describe problems arising in an implementation of a distributed SSS*-algorithm. Kraas shows by simulations [41], that the SSS*-algorithm parallelizes well, if the list of living subproblems used in this algorithm is kept on a single processor. These simulations showed a speedup of roughly 19 with 40 processors. For massive parallel systems, however, the single processor organizing the list of living subproblems is a bottle neck.

The Algorithm of Hsu

Another approach using a distinguished processor was proposed by Hsu [34]. A special processor searches the game tree to a depth d. All nodes of level d are queued in two different FIFO-queues, which are realized by a bus system. The first queue only contains the nodes of the minimal game tree, the second queue only contains the nodes, which are not in the minimal game tree. The nodes in these queues are searched by slave processors to a depth of d' with the sequential

algorithm. Subproblems from the first queue are searched first. The slave processors return the results of their searches to the distinguished processor searching the upper part of the game tree.

The search of the game tree to depth d uses the "Delayed Branching Scheduling Strategy", which guarantees that the parallel version will find any direct cutoff. In this algorithm there are two bottlenecks: First the distinguished processor, which has to search the upper part of the game tree such fast, that all slave processors are kept busy. Second the bus system, which realizes the queues for subproblems. Hsu [34] shows, that a fast chess machine like *DEEP THOUGHT* is able to search the upper part of the game tree fast enough. Simulations showed a speedup of roughly 350 with 1000 processors. The average processor work load was 90%. At the time of writing the machine had not yet been realized.

Theoretical Results

Independent of each other, three papers concerning a theoretical analysis of parallel game tree search algorithms were published within the last few years: Althöfer [6] presents a game tree search algorithm and shows that it achieves average speedup linear in the number N of processors, if the depths of the game trees to be searched are at least $\Omega(N \cdot log\ N)$. Karp and Zhang [39] show for their algorithm a worst case speedup linear in the number N of processors, if the depths of the game trees to be searched are at least $\Omega(N)$. Harting [33] improves the result of Karp and Zhang for binary bivalued game trees by presenting an algorithm, which achieves a speedup of $(\frac{1}{76})^k \cdot N^k$ with N^k processors in game trees of depth N for all $k \in \mathbb{N}$.

Concluding:

Table 3.4 contains a summary of the speedups achieved in some of the above mentioned implementations in the lines marked with "SPE". Speedups, which were measured by simulations are marked $*$. The lines marked "N", contain the numbers of processors used to achieve the speedups. The lower lines show the resulting efficiencies.

Lit.	[2]	[30]	[46]	[47]	[54]	[64]	[38]
	Random	Checkers	chess	chess	chess	chess	chess
SPE	6*	5.31*	3.2	3.75	5.03	4.78	5.93
N	35	27	4	4	8	7	16
EFF	0.17	0.37	0.8	0.94	0.63	0.68	0.37

Lit.	[28]	[22]	[56]	[41]	[34]	[26]	hier
	Dame	chess	chess	chess/Random	chess	chess	chess
SPE	12	11.5	101	19*	350*	126.4	344.49
N	32	16	256	40	1000	256	1024
EFF	0.375	0.72	0.39	0.48	0.35	0.49	0.34

Figure 3.4: Results of parallel game tree search algorithms

The results of this work differ in, at most, one of the following aspects from all the other publications mentioned above:

1. The results in this work are not achieved by simulation, but by the use of real parallelism.
2. The algorithm parallelized is the Scout-algorithm, which is superior to the pure $\alpha\beta$-algorithm.
3. The sequential algorithm uses all state-of-the-art heuristics and, therefore, searches game trees very efficiently.
4. The parallel algorithm runs efficiently, even on massive parallel systems.
5. The parallel algorithm is implemented in a chess program, which has been shown to be good. The version presented in this work successfully played several tournaments. A successive version finished second at the World Computer Chess Championships in 1992.
6. The results of our algorithm were achieved on a loosely coupled system, which is a parallel architecture realizable nowadays.

3.3 Dynamic Game Tree Decomposition

Our parallelization of the Scout- or the $\alpha\beta$-algorithm respectively is based on a decomposition of the game tree to be searched, and on a parallel evaluation of the resulting subproblems. The sequential Scout-algorithm is a depth first search algorithm. It visits the nodes of the tree to be searched from left to right. To do so, an actual variation is stored in memory. All the nodes to the left of this variation have been visited, or, visiting these nodes has been shown to be superfluous. The nodes to the right of this variation have not yet been visited. The idea of the parallelization of such a tree search is now to make available as much of the right brothers of the nodes on the actual variation for parallel evaluation as possible. By this, several processors may start a tree search on a subtree of the whole game tree. These processors build up actual variations for themselves, and thus make available the right brothers of the nodes on their actual variations for parallel evaluation.

With this we get a concept for a parallel algorithm. The efficiency of such an algorithm depends heavily on efficient solutions for the following problems:

1. Establishing a master-slave relationship:
   Every processor searching a subproblem generates subproblems for itself, which may be searched in parallel by other processors. Thus, it is a potential master. An idle processor is a potential slave. In order to make an idle processor busy it has to receive a subproblem from one of the working processors for parallel evaluation. A master-slave relationship has to be established. The efficiency, which is achieved for the subalgorithm of establishing a master-slave relationship will determine the *processor work load* in massively parallel systems.
2. Distribution of information in the distributed algorithm:
   The sequential $\alpha\beta$-algorithm allows many cutoffs in the game trees, that occur in practical applications like a chess program. The number of cutoffs, and thus the efficiency of the sequential search, is heavily influenced by the quality of the first successors of the inner game tree nodes. The sequential search algorithm uses the information computed during the search of the left parts of the tree to improve the ordering of successors of inner game tree nodes, as well as to tighten the bounds which allow the cutoffs. In our concept for

a parallel game tree search this successive improvement of the information available for move ordering and the tight $\alpha\beta$-bounds can be realized by communication between the processors.

If this communication does not suffice to supply the processors with information similar to the information available to the sequential algorithm, then the parallel algorithm may loose a lot of cutoffs which would have been computed by the sequential one. The parallel algorithm visits more nodes than the sequential one. *Search overhead* arises.

If the communication for the distribution of information increases too much, then the processor work load will decrease. A possible small improvement of the search efficiency by the distribution of information can not compensate for a very poor processor work load. There is a *tradeoff* between processor work load and search overhead.

3. Restricting the use of parallelism to avoid search overhead:
   Game trees occurring in practice are very similar to the minimal game tree, i.e. many cutoffs occurring in the minimal game tree will also occur in the game tree that has to be searched. Even if the available information about cutoff bounds etc. is distributed optimally, in some situations it may not make any sense to use parallelism. The parallel algorithm may visit only nodes, which the sequential one would not search. In situations like this, one would prefer an idle processor to remain idle for a short time and then to begin working on a subproblem which is useful to evaluate, rather than to begin working on a subproblem immediately, which is irrelevant with high probability. Here the tradeoff between processor work load and search overhead can be observed once again: If parallelism is used too carefully, the processor work load and the search overhead may both decrease. If parallelism is used too rashly, the processor work load may increase, but so does the search overhead.

For establishing the master-slave relationships we were able to use results we received in other tree searching algorithms [20, 49]. In this work we will present methods for an efficient sharing of information in a distributed system. Furthermore, we will investigate some methods, which reduce the search overhead but decrease the processor work load, and methods, which increase the processor work load but increase the search overhead too.

3.4 Basic Version of Our Parallel Scout-Algorithm

3.4.1 Start of a Master-Slave Relationship

In the initial status of the algorithm all processors are idle. The host processor reads the root position and sends it to a processor called *network master*. In the following, this processor behaves just like any other processor in the network.

A processor, which receives a subproblem, is alone responsible for the correct evaluation of this subproblem. It starts the search of this subproblem as in the sequential algorithm, i.e. it does a depth first search in the corresponding subtree. By this, an actual variation of subproblems is built and other subproblems are generated, which are stored for later evaluation according to the method of depth first search. These subproblems may be sent to other processors for parallel evaluation.

An idle processor starts its search for work by randomly choosing an arbitrary processor in the network. It then sends a

$$REQUEST = (s, e)$$

to it. Here s and e are the processor numbers of the sender and the receiver of the message. The search for work is described in Figure 3.5.

```
procedure start search for work();
/* Let #proc be the number of processors in the network,
   let p be the own processor number */
begin
    e:= random(0, #proc − 1);
    send a request (p, e) to processor e;
end;
```

Figure 3.5: Starting a search for work

A processor, which receives a request, searches along its actual variation for free subproblems, which are not yet under consideration. In order to do so it runs the algorithm described in Figure 3.6. For the moment, we define "free" as follows:

Definition 3.3 *(Free node)*

A successor $v.j$ of a node v of the actual variation is called free, if the following conditions are satisfied:

1. *$v.j$ is not yet searched by any processor*
2. *$v.j$ is not completely evaluated*
3. *$j > 1$.*

A change in this definition will be made in Section 3.7. We will, at that point, define a node to be free only if it is free according to the definition above and the node $v.1$ is completely evaluated. This will lead to an implementation of the Young Brothers Wait Concept.

If an idle processor has sent a request, then it waits until it gets a

$$SUBPROBLEM = (s, e, v, \alpha, \beta, u, o, r, c)$$

to evaluate or a

$$CANCELLATION = (s, e, n).$$

If it gets a subproblem $(s, e, v, \alpha, \beta, u, o, r, c)$, then it runs the initialization described in Figure 3.7 and starts the search in level r. A *master-slave relationship* has been established. Here s is the sender of the subproblem, and e the receiver. The node v is the root of the subproblem in the game tree to be searched, $[\alpha, \beta]$ is the search window of the father of v, u, and o are the bound variables of the father of v. r contains the depth of the node v, and c contains the hash value of the father of v. With the help of the search window and the bound variables, the receiver of

```
procedure react upon a request(s: processor);
var i: integer;
    found: boolean;
begin
    found:= false;
    if not idle then
    /* Let (v_r, ..., v_t) be the actual variation */
    begin
        i := r;
        while (i ≤ t) and not found do
            if ∃ a free successor of v_i then found:= true
                                         else i:= i+1;
    end;
    if found then
    /* Let [α_i, β_i], r ≤ i ≤ t be the search windows in level i,
       let val_i, low_i and high_i, r ≤ i ≤ t be the bounds for level i,
       let v_i.j be the leftmost free successor of v_i,
       let h be the hash function for the transposition table,
       let p be the own processor number */
    begin
        send a subproblem (p, s, v_i.j, α_i, β_i, low_i, high_i, i + 1, h(v_i)) to processor s;
        slave(i):= slave(i) ∪ {s};
        code(s):= h(v_i);
    end
    else
    /* Let n be the number of messages received after the last iteration */
    begin
        send a cancellation (s, e, n) to processor s;
    end
end;
```

Figure 3.6: Algorithm reacting upon a request

```
procedure initialize subproblem((s, e, v, α, β, u, o, r, c): subproblem);
begin
    master:= s;
    code:= c;
    α_{r-1} := α; β_{r-1} := β; low_{r-1} := u; high_{r-1} := o;
    t := r − 1;
    downwards(t, v, [low_{r-1}, high_{r-1}]);
end;
```

Figure 3.7: Initializing a subproblem

```
procedure react upon a cancellation((s, e, n): cancellation);
begin
    delay(n);
    start search for work;
end;
```

Figure 3.8: Algorithm reacting upon a cancellation

a subproblem is able to start a research at node v by itself. The variable c is used as a code of the subproblem with root v. The receiver of the subproblem returns this code together with the result to the sender of the subproblem after it has finished the evaluation of the subproblem. If the processor gets a cancellation (s, e, n), it again chooses a processor randomly and starts searching for work by sending a request again, as described in Figure 3.8. The variable n of the cancellation contains heuristic information about the amount of communication which occurred at the sender of the cancellation in the moment of sending it. If information n indicates that the communication at the sender of the cancellation was too large, the process of searching work is delayed for a short time. The amount of communication occurring at the sender of the cancellation is interpreted as an approximation of the amount of communication occurring in the whole network. If the sender of a cancellation is overloaded, the receiver believes that a large amount of communication is necessary all over the network. It reduces its own communication by delaying its search for work for a short time.

3.4.2 End of a Master-Slave Relationship

A processor p, which solved a subproblem with a root in level $r+1$ by itself or with the help of other processors, sends a

$$RETURN = (p, e, x, r, pv, c),$$

to its master e. This return message contains the value or the bound x computed for the root of its subproblem as well as the principal variation computed for the corresponding subtree. To avoid that the receiver of a return message reacts upon a return for a subproblem which is no longer active, the slave sends as a part of the return message the code c to its master which

```
procedure react upon a return((s, e, x, k, pv, c): return);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search windows in levels i,
   let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables for the levels i.
   Then we have t ≥ k. */
var cutoff, improvement: boolean;
begin
    if (s ∈ slaves(k)) and (c = code(s)) then begin      /* slave is valid */
      cutoff:= false; improvement:= false;
      if v_k is a MAX-node then begin
        if x ≥ β then cutoff:= true;
        if (x > low_k) then improvement:= true;
        val_k := max(val_k, x); low_k := max(low_k, x); high_k := low_k + 1;
      end
      else begin                                          /* v_k is a MIN-node */
        if x ≤ α then cutoff:= true;
        if (x < high_k) then improvement:= true;
        val_k := min(val_k, x); high_k := min(high_k, x); low_k := high_k − 1;
      end;
      if cutoff then begin                                /* stop searching below v_k */
        send cutoff(k);
        principal variation below v_k := pv;
        t := k − 1;
      end
      else
      if improvement then begin                           /* narrow windows */
        principal variation below v_k := pv;
        send window(k);
        update windows(k + 1, [low_k, high_k]);           /* see Figure 3.11 */
        if cutoff occurred in update windows then
          /* let the cutoff be in level i, r ≤ i ≤ t */
          if zero window search at node v_i then          /* initialize research at v_i */
            /* downwards as described in Figure 2.4 on page 18 */
            if v_{i−1} is a MAX-node then downwards(t, v_i, [max(u_{i−1}, α_{i−1}), β_{i−1}])
                                     else downwards(t, v_i, [α_{i−1}, min(o_{i−1}, β_{i−1})]);
      end;
      slaves(r) := slaves(r) \ {s};                       /* delete slave s */
    end
end;
```

Figure 3.9: Reacting upon a return

```
procedure send cutoff(k: level);
/* Let (v_r, ..., v_t) be the actual variation,
   let p be the own processor number */
var i, j: integer;
begin
    for i := k to t do
    begin
        for all s ∈ slaves(i) do
            send the cutoff message (p, s, code(s)) to processor s;
        slaves(i) := ∅;
    end
end;


procedure send window(i: level);
/* Let p be the own processor number */
begin
    for all s ∈ slaves(i) do
        send the window message (p, s, [α_i, β_i], [low_i, high_i], code(s)) to processor s;
end;
```

Figure 3.10: Sending of cutoffs and window messages

it received together with the subproblem. Afterwards the master-slave relationship is resolved. The slave is again idle. The receiver of a return message runs the algorithm described in Figure 3.9.

First, the receiver of a return (s, e, x, k, pv, c) checks if processor s is a slave in level k, and whether the code c received equals the code of s or not. If this is not the case, then the return is considered to be too old and is rejected. The receiver then checks if the value x computed by s allows narrowing the search window at the node v_k and updates the bound variables of level k.

3.4.3 Improvements of Search Windows

If x causes a cutoff at the node v_k, then all the slaves in levels $k, \ldots, t$ are informed by a cutoff message

$$CUTOFF = (s, e, c)$$

that they may stop working on their subproblems. Here s is the sender of the cutoff, e is the receiver, and $c = code(e)$ is the code of the slave e at the master s. The variation pv sent by s in the return message is saved as the principal variation at v_k, and the search is continued at v_{k-1}. The procedure "send cutoff" is described in Figure 3.10.

If x does not cause a cutoff, but causes an improvement of the minmax value val_k of level k, then pv is saved as the principal variation below v_k. Each slave in level k is informed by a window

```
procedure update windows(k: level; [α, β]: search window);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search windows of levels i,
   let val_i, low_i and high_i, r ≤ i ≤ t the bound variables of the levels i. */
var i : integer;
    cutoff, improvement: boolean;
begin
    cutoff:= false;
    i:= k;
    repeat
        improvement:= (α_i < α) or (β_i > β);                /* check new window */
        α_i := max(α_i, α);                                  /* update new window */
        β_i := min(β_i, β);
        if v_i is a MAX-node then
        begin
            if low_i ≥ β then cutoff:= true;
            low_i := max(low_i, α);
            if v_i.1 is completely evaluated then high_i := low_i + 1;
        end
        else                                                 /* v_i is a MIN-node */
        begin
            if high_i ≤ α then cutoff:= true;
            high_i := min(high_i, β);
            if v_i.1 is completely evaluated then low_i := high_i − 1;
        end;
        if improvement and not cutoff then send window(i);
        if not cutoff then i := i + 1;
    until (i > t) or cutoff or not improvement;
    if cutoff then                                           /* stop searching below v_i */
    begin
        send cutoff(i);
        t := i − 1
    end
end;
```

Figure 3.11: Update of the search windows

message

$$WINDOW = (s, e, [\alpha, \beta], [u, o], c)$$

about the narrowing of the search window. With these messages the new contents of the search window variables α and β as well as the contents of the bound variables u and o of level k are transmitted to the slaves in level k. The algorithm of the procedure "send window" is described in Figure 3.10.

By the narrowing of the search window of v_k a narrowing of the search windows of the nodes $v_{k+1}, \ldots, v_t$ of the actual variation below v_k may be possible. The procedure "update windows" realizes this narrowing of the search windows. This is described in Figure 3.11.

The procedure "update windows" runs along the actual variation and checks if the new window $[\alpha, \beta]$ is an improvement of the actual upper or lower bound. If so, the lower bound at a MAX-node is increased and the upper bound at a MIN-node is decreased, and thus the search window is narrowed. In the case of a narrowing of the search window in level i, all slaves in level i are informed by a call of the procedure "send window". These slaves then run the algorithm described in Figure 3.12.

An improvement of the search window of a node v, which is searched by a zero window search, causes a cutoff at v. In this case a research is initialized for v, as if the new bound would have been computed by the zero window search itself. Thus the new window stops the search of the subtree with root v, the receiver of the window message or the return message resp. starts the research by itself, if necessary.

If a processor receives a cutoff message, then it immediately stops the computations running at its slaves. The processor becomes idle. This algorithm is described in Figure 3.12.

3.4.4 The Distributed Algorithm

Figure 3.13 shows the simplified process structure for our distributed Scout-Algorithm. This process structure already allows to access the distributed transposition table, which will be described more detailed in Section 3.6. Besides the processes for demultiplexing, routing, buffering and multiplexing, which are combined to a single routing process, every processor runs a working process. This tree searching process calls three sequential processes in a loop. The tasks of these processes can be described as follows:

<u>Iteration:</u>

- Generation of a node or
- Static evaluation of a leaf node or
- Evaluation of a subsearch

The process "Iteration" results from a change of the recursive sequential algorithm from Figure 2.2, into the iterative one, which is described in Figures 2.3 and 2.4. In the parallel algorithm an iteration may be done only if the next iteration is not going to perform an "upwards" over a level, in which still some slaves are active. In Figure 3.14 this is expressed in procedure "Iteration" by the condition "iteration allowed".

```
procedure react upon a window message((s, e, [α, β], [u, o], c): window);
/* Let (v_r, ..., v_t) be the actual variation,
   let [α_i, β_i], r ≤ i ≤ t be the search windows of the levels i,
   let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables of the levels i. */
begin
    if (master = s) and (code = c) then                  /* valid message */
    begin
        α_{r-1} := α; β_{r-1} := β;                      /* initialize window of the father */
        low_{r-1} := u; high_{r-1} := o;
        if v_r is a MAX-node then                        /* update new search window */
          update windows(r, [α, high_{r-1}]);
        else
          update windows(r, [low_{r-1}, β]);
        if cutoff then
          /* Let the cutoff be in level i, r ≤ i ≤ t */
          if zero window search at node v_i then         /* research */
            downwards(t, v_i, [α_{i-1}, β_{i-1}]);       /* see Figure 2.4 on page 18 */
        end
    end
end;

procedure react upon a cutoff((s, e, c): cutoff);
/* Let (v_r, ..., v_t) be the actual variation */
begin
    if (master = s) and (code = c) then                  /* valid message */
    begin
        send cutoff(r);
        idle:= true;                                     /* stop searching the subproblem */
    end
end;
```

Figure 3.12: Reacting upon a window resp. cutoff message

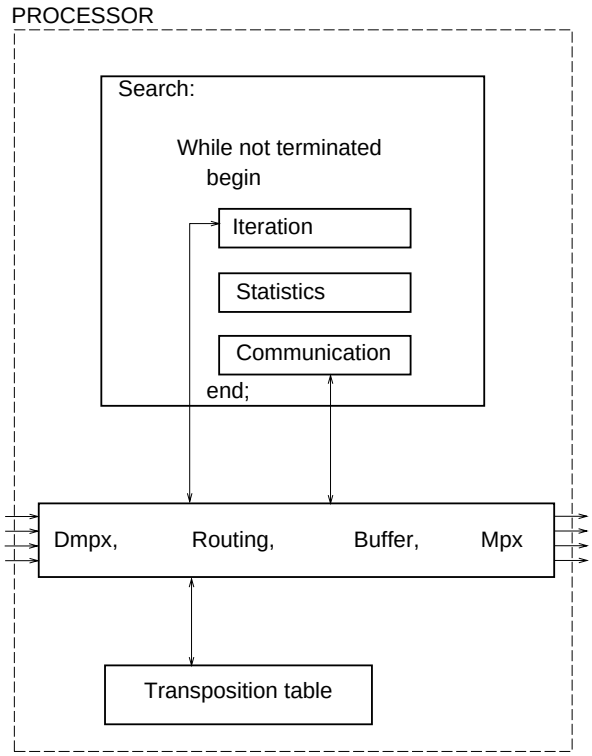


Figure 3.13: Structure of the processes running on a processor

Statistics:

- Several tasks for computing the statistics

Communication:

- If the subsequent iteration makes the sending of a message necessary, then send this message.
- While there are still some messages in the buffer, react upon the next message.

The process "Communication" is described more detailed in Figure 3.14.

Through this process structure, it becomes apparent that incoming messages are reacted upon only between two iterations of the game tree search, i.e. a processor reacts upon a message only if the last iteration has been finished. Thus, the actual variation of the receiver of a message is defined uniquely, whenever a message is read.

Delay by Communication

Furthermore, the description of the process "Communication" shows that incoming messages are read until the buffer for incoming messages is empty. This guarantees that incoming messages are

```
procedure Iteration;
begin
    if iteration allowed then iterate;
end;

procedure Communication;
/* Let p be the own processor number. */
begin
    /* Send the messages, which were caused by the subsequent iteration */
    if idle then
    begin
        if not request sent then
          start search for work;
        if subproblem finished then
          send return (p,master,val_r, r, pv_r,code) to the master;
        if subproblem (s, e, v, α, β, r, c, nfs) received then
          initialize subproblem(s, e, v, α, β, r, c, nfs);
    end
    else
    begin
    /* Let val_i, low_i and high_i, r ≤ i ≤ t be the bound variables for the levels i,
       a cutoff or an improvement may have occurred in level k */
       if cutoff then send cutoff(k) else
       if improvement then send window(k);
    end;
    /* Receive messages */
    while message received do
    begin
        if request (s, e) received then
          react upon a request(s);
        if return (s, e, x, k, pv, c) received then
          react upon a return((s, e, x, k, pv, c));
        if window (s, e, [α, β], [u, o], c) received then
          react upon a window((s, e, [α, β], [u, o], c));
        if cutoff (s, e, c) received then
          react upon a cutoff((s, e, c));
        if cancellation (s, e, n) received then
          react upon a cancellation((s, e, n));
    end
end;
```

Figure 3.14: Communication process of the parallel Scout-algorithm

read as fast as possible. However, without any restriction regulating the number of messages to be read in situations of communication overload, in the worst case this algorithm runs an endless loop. But even if this loop terminates at some time, a large amount of communication may cause the process "Communication" to run for a very long time. During this time, the processor is not able to continue its tree search. A large amount of communication comes up if the processor work load is bad, since many messages for the searches for work of all the idle processors then have to be communicated. In such phases it may happen that the very few processors working on subproblems are not able to split the subproblems into smaller subproblems fast enough, since they are not able to perform enough iterations. As a consequence the processor work load remains poor and the number of messages remains large. It would help if processors could delay their searches for work in phases, when the amount of communication in the network is high. However, it seems to be senseless to compute the global amount of communication in the network, whenever a processor is going to start a search for work. Therefore, a processor reacting upon a request for work by sending a cancellation , informs the receiver of the cancellation about the amount of communication in its communication process at the time of reading the request. The receiver of a cancellation views the communication load at the sender of the cancellation as an approximation for the global communication load. If the communication load at the sender of the cancellation was large, then it delays its search for work for a short time, thus reducing the global communication load. This results in a dynamic, self-regulating system: The larger the communication load, the more processors delay their searches for work for a short time, thus reducing the communication load. If the communication load is small, then the search for work is continued immediately and the communication load is increased.

Delay by Routing

Routing processes, (de)multiplexer, and the transposition table process run in time-sharing to the game tree searching process "Iteration" on a single processor. In order to guarantee a fast routing of messages, these processes have high priority compared to the tree searching process. This means, that the game tree searching process is stopped by the scheduler, whenever one of the communication processes is able to run, and is restarted only if these processes come to some send or receive constructs, which are not executable at the moment. With this we immediately get the

Observation 3.1 *(CPU-delay)*

A large amount of messages, which have to be routed, or messages for the transposition table will delay the execution of the game tree searching process.

Three examples

Figure 3.15 shows the parallel evaluation of the two nodes $v.2$ and $v.3$ by two processors P_1 and P_2. After the evaluation of the subtree rooted at $v.1$, P_1 receives a request for work from P_2. At this time, let v be the only node of the actual variation of P_1. Let $[\alpha, \beta] = [1, 9]$ be the search window and let $[u, m, o] = [5, 5, 6]$ be the bound variables of the node v. The node $v.2$ is free, because it is not searched by any processor. Since v is the only node of the actual variation, P_1 chooses $v.2$ as a free successor of v for transmission. P_1 sends the subproblem $(P_1, P_2, v.2, 1, 9, 5, 6, Tiefe(v.2), h(v))$ to P_2. Then, P_2 initializes the subproblem $v.2$ as well as

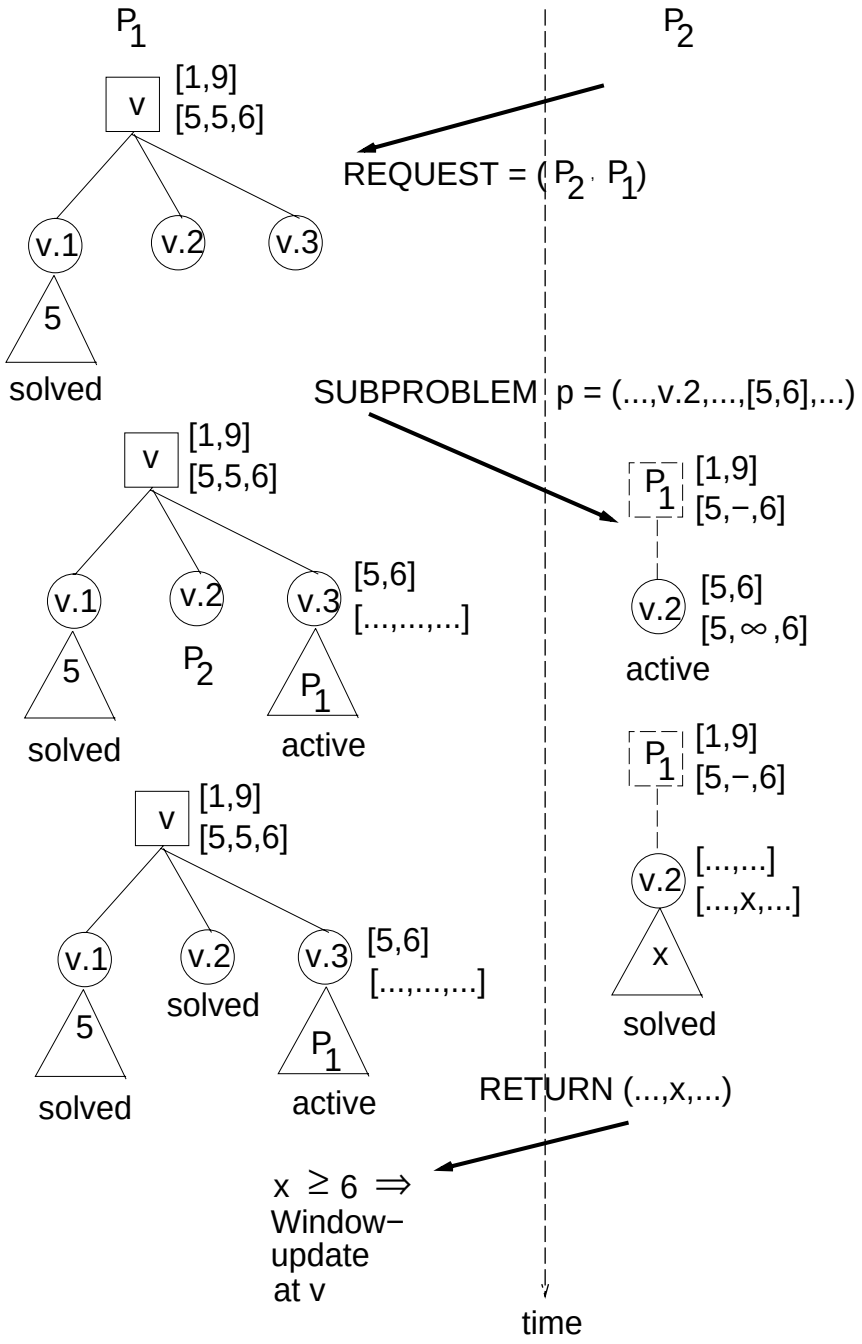


Figure 3.15: Parallel search of a game tree without any new windows

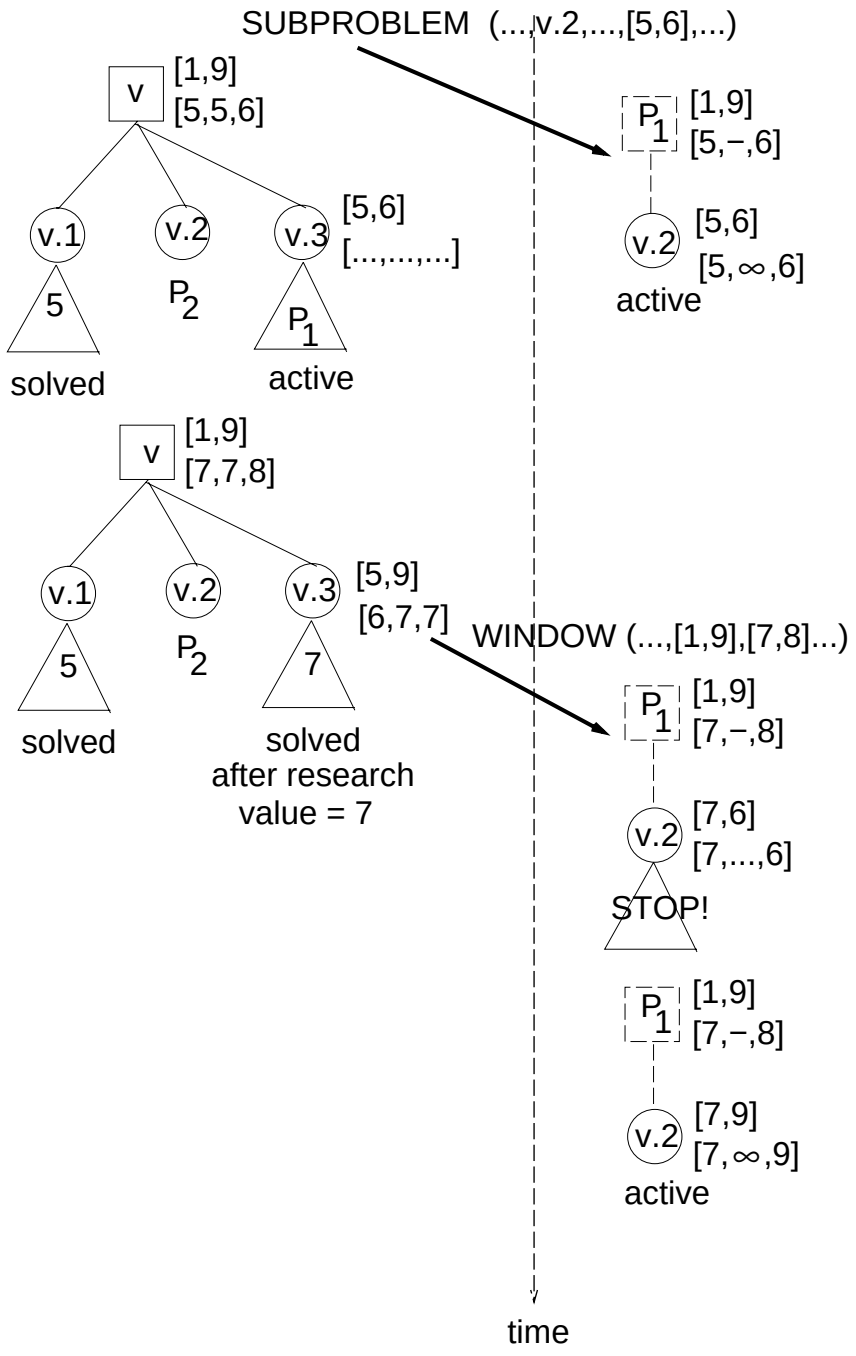


Figure 3.16: Window message for a zero window search

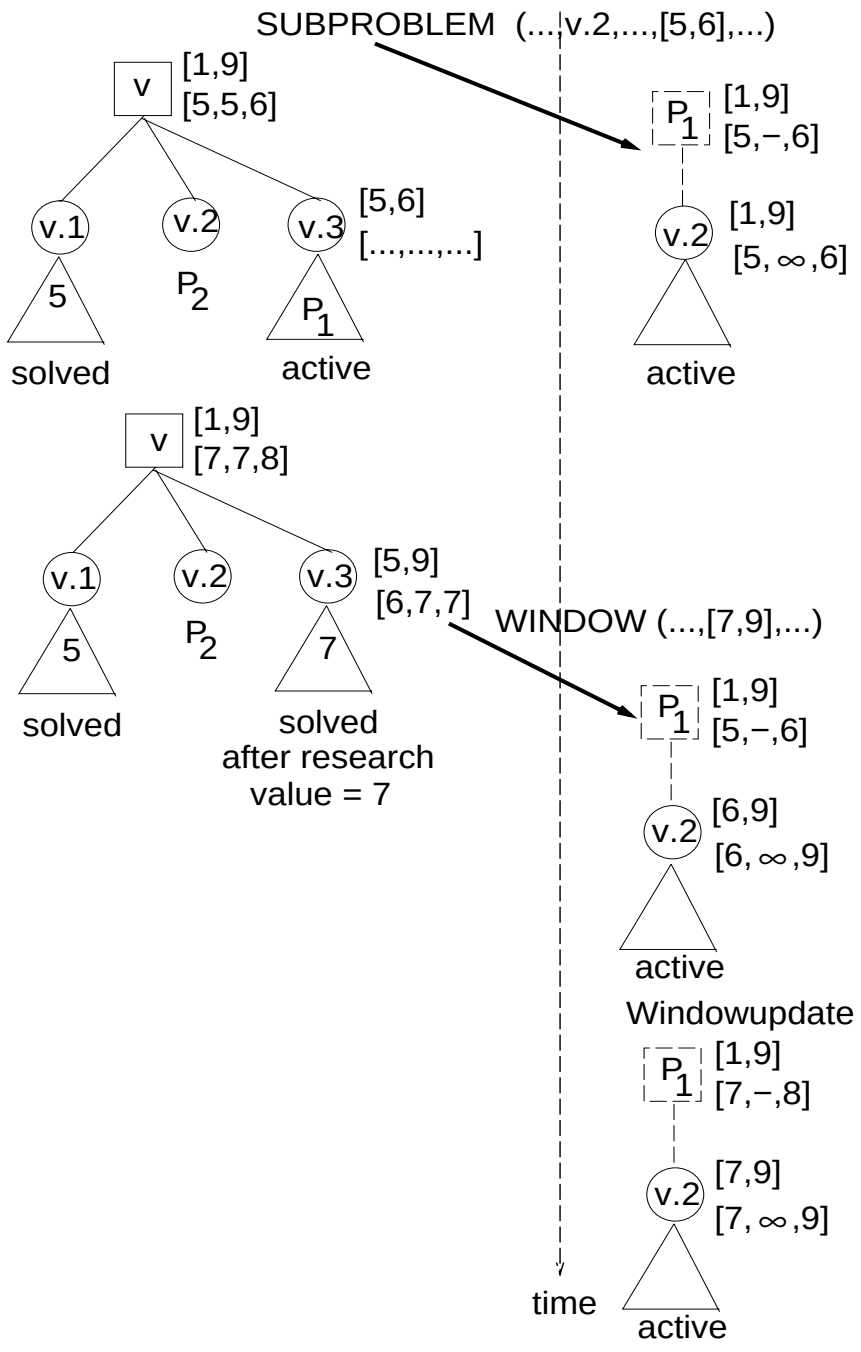


Figure 3.17: Window message for a research

the search window and the bounds of the father of $v.2$. With the help of these variables, P_2 is able to start a research at $v.2$ on its own, if necessary. P_2 now searches the subtree rooted at $v.2$, P_1 searches the subproblem rooted at $v.3$. After $v.2$ is completely evaluated resulting in a value x, P_2 returns x to P_1. The value x may influence the minmax value of the MAX-node v only if $x \geq 6$. In this case P_1 calls "update windows" for the nodes of its actual variation, as described in Figure 3.11.

Figure 3.16 shows a situation in which the processor P_1 finishes the evaluation of the node $v.3$ earlier than P_2 its search below $v.2$. The value of the node $v.3$ improves the minmax bound m of the node v from 5 to 7, such that a window message is sent to P_2. The node $v.2$ actually is searched by P_2 with a zero window search and a search window of $[5, 6]$. The window message received indicates, that the value of $v.2$ may influence the value of v only if it is greater than or equal to 8. The node $v.2$ now is searched by P_2 with the search window $[7, 9]$, as if the search were started without zero window search after the computation of $F(v.3) = 7$.

Figure 3.17 shows a similar situation. Here P_2 receives the window message containing the new bound 7 at a time, when it already is researching the subtree rooted below $v.2$ by itself with a search window of $[6, 9]$. The new bound of 7 now is used to narrow the search window to $[7, 9]$. Then the search windows of the actual variation below $v.2$ are narrowed, until for the first time a cutoff occurred or a search window is not improved by the new bounds.

3.4.5 Thoughts on the Distribution of Work Load

The algorithm described above implicitly contains a good load distribution algorithm: Idle processors try to get subproblems by communicating very much. With this, the time necessary to get a new subproblem is kept short. Every processor, which gets a subproblem and then starts evaluating it, automatically becomes a potential master for other idle processors. Every processor may be master or slave of any other processor. The only exception from this rule is the network master, which keeps the root of the whole game tree, and thus cannot become idle before the whole game tree search is finished.

By the random choice of a possible master, of course, it is not guaranteed that the processors are distributed uniformly over the game tree according to the size of the subproblems. On the other hand, the size of a subproblem is not predictable at the beginning of its evaluation. The size of a subproblem rooted at v with a search window $[\alpha, \beta]$ at v is determined by

1. the size of the search window $[\alpha, \beta]$,
2. the depth r of v in the game tree,
3. the move ordering in the subtree rooted at v,
4. the location of the value $F(v)$ with respect to the bounds α and β, and
5. the number of the search extensions in the subtree below v.

Reliable information for the last three items is not available before a subproblem $(G(v), [\alpha, \beta])$ is solved. Therefore, even a more-or-less centralized distribution of the processors over the subproblems would have to rely on some guesses about the size of the subproblems. By their arbitrary searches for work and by the random choice of a potential master, processors, which get only small subproblems, more often ask other processors for work than processors, which get

a large subproblem for evaluation. By this, after a while a distribution of the processors over the subproblems is achieved as one would like to have: as long as large subproblems are available for parallel evaluation, more and more processors get such a subproblem, and thus work in the large subtrees of the game tree. Of course, no one guarantees that the processors are evaluating relevant subtrees.

In many cases, it makes sense to delay the use of parallelism. In order to do so, it may help to keep some subproblems in the queues, instead of sending them away for parallel evaluation, until the probability is high enough that the subproblems are relevant for the minmax value of the root of the game tree. Since a processor should be able to evaluate a subproblem without getting any commands where to search or where to use parallel evaluation, etc., such a decision to delay the use of parallelism should be possible on the basis of local heuristics and local information alone. We will see later that local heuristics are sufficient to achieve such an effect.

In this section we discussed the basic version of our distributed search algorithm. In the next sections we will describe how the heuristics necessary for a good move ordering and the detection of transpositions are implemented in our distributed algorithm.

3.5 Local Killer and History Heuristics

Every processor keeps its own local killer lists. Information from these killer lists is not exchanged between the processors.

Therefore, the contents of a killer list depends upon the subproblems, which the processor searched before. Killer lists are an important heuristic used for the move ordering. They are useful because their contents, the killer moves, may change when the search process travels through the game tree from left to right. However, these changes occur only rarely. Thus, one can expect, that good killer moves for subtrees not far from each other will only rarely differ. Good killer moves for subtrees far from each other, however, may often be distinct. In the sequential algorithm this fact is considered in the algorithm to update the killer lists in Figure 2.8 on page 30. An idle processor in our distributed algorithm, however, not necessarily gets a subproblem close to its former subproblem. By the random choice of a potential master the new subproblem may be very far from the former subproblem, and, therefore, may require completely different killer moves. These new killer moves now have to be found again by searching a part of the new subproblem and by updating the killer list, in order to make the killer heuristic work as well as in the sequential case.

Occasional exchanges of information between the killer lists of neighbored processors or exchanges of information between two processors establishing a master-slave relationship do not improve the behaviour of our distributed game tree search algorithm.

On the other hand the data of Table 3.2 shows, that the local killer lists may be used successfully. The data was computed during an 8 ply search on the 24 positions of the Bratko Kopec set using 256 processors.

In the sequential algorithm, the nodes of the brute force tree, which are reached via killer moves, nearly cause 42% of all the cutoffs achieved by the search of the right successors. In the quiescence search this percentage clearly is smaller (16%), since the killer moves usually are not searched. The parallel algorithm using 256 processors computes a percentage of 34% of all the cutoffs, which are not caused by the first successor, by searching a killer move (14%

1 processor		256 processors	
BF	QS	BF	QS
41.57%	16.27%	34%	14%

Table 3.2: Cutoffs by killer moves in % of the number of cutoffs by right sons

in the quiescence search). The above described data, however, is somewhat misleading to the disadvantage of the parallel algorithm: If two right successors of a node will cause a cutoff, one of which is a successor reached via a killer move, then in the parallel algorithm it may happen, that the value of the successor, which is not reached via a killer move, is computed faster than the value of the killer. In this case the cutoff is not counted as a cutoff by a killer, although the killer move may be ordered to the left of the non killer move.

Observation 3.2 *(Local Killer Lists)*

34% *cutoffs achieved by killer moves indicate, that in the local killer lists enough useful information is contained.*

Another method, which collects information about the profitability of moves is the history heuristic. Information contained in the history tables change similar to the information of the killer lists. Therefore, for the history heuristic an exchange of information was omitted, also: every processor keeps its own local history table.

3.6 The Distributed Transposition Table

3.6.1 Local, Global or Distributed Hash Tables ?

The sequential algorithm uses a transposition table to save the results already computed for some nodes of the game tree. These results may then be requested from the table instead of computing them again by large searches. Usually the transposition table is realized as a hash table for positions. In many cases, by the use of the transposition table, it may be avoided that equal positions which by some transposition of moves may be reached twice in the game tree, are evaluated twice. On the other hand, the transposition table combined with the method of iterative deepening is of decisive importance for a good move ordering and thus for the efficiency of the search.

Marsland and Popowich [46] show, that the use of local transposition tables result in an increase of the search overhead. This increase of the search overhead is caused by the fact, that the processors have access only to the information in the transposition table collected by themselves. In Table 3.3 these results are contained. Marsland and Popowich investigated five versions of a 4 processor program:

1. TR-: No access to any transposition table.
2. GTR: Access to a global transposition table of 8192 entries for every node of the game tree. The global transposition table is kept on one processor.

		time	leaves	Spe	SO (%)	CO (%)
1 Proc		1275	52705	1.0	0.0	0.0
	TR-	532	64470	2.4	22.3	26.7
	GTR	872	56202	1.5	6.6	61.0
4 Proc	GTRP	420	54558	3.0	3.5	21.4
	LTR	457	57930	2.8	9.9	23.4
	LTRP	402	55224	3.2	4.8	20.4

Table 3.3: Results for local and global transposition tables

3. GTRP: As GTR, however, access to the table is allowed only for the nodes of the game tree, which have to be searched to a depth > 2.

4. LTR: Access to a local transposition table of 4096 entries each.

5. LTRP: As LTR, however, access to the local tables is allowed only for the nodes of the game tree, which have to be searched to a depth > 2.

The percentage of the nodes searched more by the 4 processor version than by the sequential version increases, if only local transposition tables are used, from 6.6% to 9.9%, if every node is accessed and from 3.5% to 4.8% if only partial access is allowed. This loss of information results in worse move ordering and minor detection of transpositions. The effect of this loss already becomes apparent if only 4 processors are used. For large numbers of processors, the information contained in a local table of a single processor is much smaller than in the 4 processor version. Thus, local transposition tables are of no use for massive parallel systems.

Additionally, the data of Table 3.3 makes clear, that a global transposition table kept on one processor produces a bottleneck, which decreases the efficiency of the whole system. Thus, the sum of the idle times and the communication delays (column "KO") increases from 23.4% of the whole running time to 61.0% for the versions LTR and GTR. Allowing only partial access this difference decreases. The distinguished processor keeping the transposition table is overloaded such heavily by communication tasks even in a system with a small number of processors, that the time necessary for serving accesses to the global table becomes very long. By this, however, all processors accessing the global table are delayed. For large numbers of processors, this delay would be fatal. Thus, a global transposition table is of no use for massive parallel systems too.

Hybrid versions of the above described methods were implemented in the chess program SUN PHOENIX of Schaeffer [64]: Positions which are located in the upper part of the game tree are stored in a global table, positions of the lower part of the game tree are kept in local tables. The average speedup of a 7 processor version is increased from 4.41 to 4.78 compared to the version with local tables only.

In order to get the advantages of the global table and to communicate without any bottleneck the implementation of a distributed transposition table seems useful. A distributed transposition table was described by6 Otto and Felten [56]. It was implemented in the distributed chess program WAYCOOL. In this approach every processor keeps a part of the transposition table. Any accesses as well as new information to be stored into the table now have to be routed to the processor, which keeps the corresponding entry. Answers to transposition table accesses have to be routed back to the requesting processor.

3.6.2 Implementation of a Distributed Transposition Table

Every processor in a distributed system consisting of n processors keeps a transposition table of the same size max as in the sequential algorithm. Logically, these subtables were viewed as one large table of size $max \cdot n$. The hash function now returns an index from $\{0, \ldots, n \cdot max - 1\}$. If a position v of the game tree has a hash value

$$h(v) = (ix(v), s(v)) \in \{0, \ldots, n \cdot max - 1\} \times \{0, 1\}^{30},$$

then we split $ix(v)$ into a processor number

$$p(v) := ix(v) \bmod n$$

and an index into the subtable of $p(v)$

$$i(v) := \lfloor \frac{ix(v)}{n} \rfloor.$$

With this we are able to increase the transposition table size with the size of the parallel system. The more processors are used for the game tree search, the larger game trees we hope to be able to search. The search of large game trees, however, makes desirable a large transposition table, since otherwise too many information may be lost by collisions. Thus, one may expect a transposition table, which grows with the size of the parallel system, to be helpful for the game tree search.

Store and Read

Two operations have to be implemented for the distributed transposition table: Store and read. If a processor P evaluates a subproblem rooted at a node v, and the solution shall be stored into the entry $ix(v)$ of the transposition table, then P does the following: it computes the processor number $p(v)$ and the index into the subtable of $p(v)$, $i(v)$, and sends a message

$$NEWENTRY = (p_1, p_2, i, s, x, k, z, t, c)$$

to processor $p(v)$. Here p_1 is the own processor number and $p_2 = p(v)$ is the processor number of the receiver. x, k, z, and t are the results computed for v, c is the actual time stamp and $i = i(v)$ the index of the entry at the processor $p(v)$. When a processor gets such a new entry, it decides just as in the sequential algorithm, whether the contents of the received messages are to be stored in the entry with index $i(v)$.

If a processor P starts the evaluation of a node v, and wishes to read information possibly contained in the transposition table for v, then it computes $p(v)$ and $i(v)$ and then sends a TR-request

$$TR\text{-}REQUEST = (p_1, p_2, i)$$

with $p_2 = p(v)$ and $i = i(v)$ to $p(v)$. p_1 is the own processor number. If a processor receives a TR-request (q_1, q_2, i), then it sends the contents of the entry i of its subtable with an

$$ANSWER = (p_1, p_2, s, x, k, z, t, c).$$

Here p_1 is the own processor number, $p_2 = q_1$ and $(s, x, k, z, t, c) = T_i$ is the content of the entry i of its subtable. The receiver of an answer message decides just as in the sequential algorithm, whether the information received can be used and how it can be used.

The procedures, which describe the communication necessary to implement the distributed hash table, are contained in Figure 3.18.

```
procedure store DTR-entry(ix : index; (s, x, k, z, t, c) : entry);
/* Let n be the number of processors,
   let p be the own processor number */
var q : processor; i : index;
begin
    q := ix mod n; i := ix div n;
    send a new entry (p, q, s, x, k, z, t, c) to processor q;
end;


procedure read DTR-entry(ix : index; var (s', x', k', z', t', c') : entry);
/* Let n be the number of processors,
   let p be the own processor number */
var q : processor; i : index;
begin
    q := ix mod n; i := ix div n;
    send a TR-request (p, q, i) to processor q;
    /* If p = q, then this is a cheap local operation */
    wait for an answer (q, p, s, x, k, z, t, c) of q;
    (s', x', k', z', t', c') := (s, x, k, z, t, c);
end;


procedure react upon new entry(ix : index; (s, x, k, z, t, c) : entry);
begin
    store TR-entry(ix, (s, x, k, z, t, c));          /* see fig. 2.6 */
end;


procedure react upon TR-request((p1, p2, i) : TR-request);
/* Let p be the own processor number */
var (s, x, k, z, t, c) : entry;
begin
    read TR-entry(i, (s, x, k, z, t, c));            /* see fig. 2.7 */
    send an answer (p, p1, s, x, k, z, t, c) to processor p1;
end;
```

Figure 3.18: Accesses to the distributed transposition table

```
procedure update and get(v : node);
var (s, x, k, z, t, c) : entry;
begin
    Compute h(v) = (ix(v), s(v)) from h(father(v)) and the move (father(v), v).
    parbegin
        read DTR-entry(ix(v), (s, x, k, z, t, c));
        generate the position v;
    parend
end;
```

Figure 3.19: Using the waiting time

Use of waiting times

In the above described protocols, the operation of storing new information is not critical: The information to be stored is available to all the processors as soon as the receiver of a new entry reacted upon it. In comparison to the sequential algorithm, this operation may be slightly delayed by the time necessary to route the message from the sender to the receiver. However, this delay is negligible.

Much more critical is reading the distributed transposition table: If, for position v, an entry is available in the transposition table, then this entry determines, which successors of v have to be searched, and if some, which successor has to be searched first. Thus, a processor which wants to read an entry of the transposition table, may continue its search only after it got the information of this entry. The waiting times caused by the routing of the messages will delay the work of the requesting processors significantly. Thereby, the requesting processor should be allowed to do some useful work while waiting for the answer message. Here the helpful observation is, that the information of a transposition table entry only rarely allows stopping the search of a node. In most of the cases the information from the transposition table for a node v are sufficient only to narrow the search window or to determine the first successor of v. Then the search has to be continued at some successor of v. In order to use the waiting times sensibly, processors first compute $h(v)$ without having generated v before. Two processes are then invoked on this processor: The first process sends a TR-request and waits for an answer, the second process generates the position v by initializing the corresponding data structures. These data structures include a representation of the board, some lists of pieces, for every square of the board a list of attacking pieces, etc. and have a size of roughly 1kByte. The contents of these data structures are not recomputed as a whole but can efficiently be updated from the data structures of the father of v. The request to the distributed transposition table for a node v and the generation of the position v are done in parallel. The idea to access the transposition table while the search proceeds has already been suggested by Marsland and Campbell [45] in 1982. Figure 3.19 contains the algorithm.

The costs and gains of a distributed transposition table are investigated in Section 4.5.

3.7 The Young Brothers Wait Concept

The game trees occurring in practice are very similar to the minimal game trees, because of the successful use of the move ordering heuristics. Observation 2.4 on page 40 shows that the game trees searched on the average are a factor of 1.25 times the minimal game tree. The parallelization of the game tree search should consider this special structure of the game trees to be searched. Thus, it seems to make sense to develop a parallelization, which produces no search overhead at least in a minimal game tree. In those parts of the tree, which exactly correspond to a minimal game tree, search overhead will be avoided too.

In some earlier papers [21, 22, 23, 24, 25, 26], we proposed a method to reduce the use of parallelism, which allows to search a minimal game tree without any search overhead. An additional advantage of our method is that the reduction of the use of parallelism can be done by using only information locally available. The method to reduce the use of parallelism is called *Young Brothers Wait Concept.* (YBWC) It may be stated as follows:

> **YBWC:**
>
> The search of a successor $v.j$ of a node v in the game tree must not be started, before the leftmost brother $v.1$ of $v.j$ is completely evaluated.

To implement the YBWC we define a node to be YBWC-free using the Definition 3.3 on page 53 as follows:

Definition 3.4 *(YBWC-free)*

A successor $v.j$ of a node v of an actual variation is called YBWC-free, if the following conditions are satisfied:

1. *$v.j$ is free and*
2. *$v.1$ is completely evaluated.*

The YBWC is easily implemented by replacing the word "free" in Figure 3.6 on page 54 by "YBWC-free".

The algorithm using the YBWC has some advantages:

1. It is easy to show that any minimal game tree can be searched in parallel without search overhead. Proof for this is given in [21]. It is based on the observation that in a minimal game tree all the cutoffs, which are computed in the subtrees below the right sons of the root of the game tree, are caused by the value of the leftmost successor of the root. Using the YBWC this value is known at the time when the use of parallelism is allowed to evaluate the successors of the root.

2. Even in non minimal game trees the use of the YBWC may avoid search overhead. The left son $v.1$ of a node v, which is searched with a search window $[\alpha, \beta]$ sometimes may improve the bound for the right sons of v. This is the case, if $\alpha < F(v.1) < \beta$. The search window for the right sons of v may then be narrowed to $[\alpha, F(v.1)]$ or $[F(v.1), \beta]$ respectively.

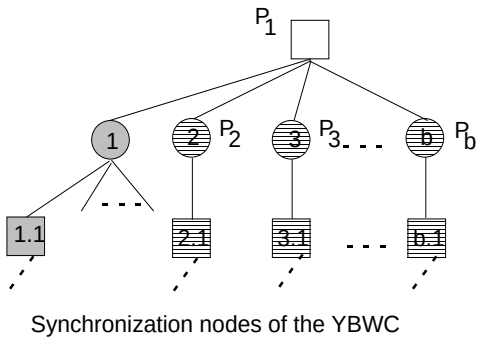


Figure 3.20: Synchronization nodes caused by the YBWC

3. The decision, whether or not the successors of a node v should be evaluated in parallel can be made on the basis of information available at the node v. The processor responsible for the evaluation of v thus is able to decide, whether to allow the use of parallelism, without communicating with other processors.

However, there is a severe disadvantage of the use of the YBWC:

Synchronization nodes

This disadvantage can be made clear with the help of Figure 3.20: At the beginning of the evaluation of a game tree only a single node, the root of the game tree, is alive. The network master starts the search by generating the leftmost successor of the root. Now two nodes are alive, however, the use of parallelism is not allowed at both. Thus, the network master generates left successors until it reaches a leaf node. After the static evaluation of the leaf, the network master moves upward in the game tree reaching the father of the leftmost leaf. Now, for the first time, it is allowed to use parallelism. The only subproblems, which may be distributed over the processors, are the brothers of the leftmost leaf. However, these subproblems usually are very small and are few compared to the number of processors used. The situation becomes better and better the closer the network master gets to the root of the game tree. The subproblems transmitted become larger, and are itself branched into smaller subproblems, which enables the use of more and more processors. The YBWC, thus, not only caused a sequential moving down the tree from the root to the leftmost leaf, but, for every inner node of the leftmost variation, causes a point of time, at which all but one processor must be idle. We will call these nodes *synchronization nodes.* The number of these synchronization nodes is increased by the use of the iterative deepening method. It grows quadratically in the search depth. The same effect as described for the synchronization nodes happens in a weakened manner at the leftmost variations of all the subtrees below the type-1 nodes.

These synchronization nodes require the possibility to quickly redistribute work from one processor to all the other processors. Therefore, the use of the YBWC is possible only if a fast load balancing or a fast load distribution is guaranteed between the processors. In our algorithm this is achieved by the method to search for work, which is described in Section 3.4.

In the literature, for every parallel game tree search algorithm of practical relevance, different

methods for delaying the use of parallelism are proposed [46, 47, 56, 41, 34]. These methods vary from reducing the parallelism only at the leftmost variation as in the PV-Split-algorithm [46, 47], to a reduction of parallelism depending on the results obtained for a node from the transposition table [56], or even to an administration of a global list of living subproblems on a distinguished processor [41], or to an administration and sequential evaluation of the upper part of the game tree on a single processor [34]. The YBWC is a method which allows a decision based upon whether to use parallelism based on locally available information only.

In the following we will describe two modifications of the YBWC: the first modification at some nodes of the game tree allows a more extensive use of parallelism taking the risk of additional search overhead. The second modification at some nodes of the game tree reduces the use of parallelism more strictly than the pure YBWC, with the aim to reduce the search overhead. Both modifications make some assumptions about the structure of the game tree to be searched, and thus come to different decisions about the use of parallelism.

3.7.1 The YBWC for Type-1 and Type-2 Nodes

The mapping *type* from Definition 2.5 on page 39 identifies the nodes of the minimal game tree. Hsu [34] implicitly extended the above mentioned mapping to the nodes, which are not in the minimal game tree. The extension can be defined as follows:

Definition 3.5 *(Expected type of nodes)*

$$type(v.j) := 2 \quad \textit{if } type(v) = 2 \textit{ and } j > 1$$

We call the type of a node v defined with the help of the above extension the expected type of v.

With this extension the mapping *type* is defined for every node of any game tree. The extension may be viewed as follows: If the left son of a type-2 node v fails to produce a cutoff at v, then a cutoff at v is no more likely. We expect a cutoff to occur at the right sons of v.

We now have two disjoint classes of type-3 nodes:

1. Type-3 nodes, which are part of the minimal game tree.
   All successors of these type-3 nodes are lying in the minimal game tree, i.e. these successors have to be searched anyway.

2. Type-3 nodes, which are not part of the minimal game tree.
   At these nodes we do not expect a cutoff.

At the type-3 nodes, at which no cutoff occurs, the only use of the YBWC is that before parallelism is invoked the result of the first successor is computed, which may provide a better bound for the evaluation of the right successors.

YBWC-1-2:

Thus, we get the first modification of the YBWC, the YBWC for type-1 and type-2 nodes (YBWC-1-2) by ignoring the rule for the YBWC at type-3 nodes.

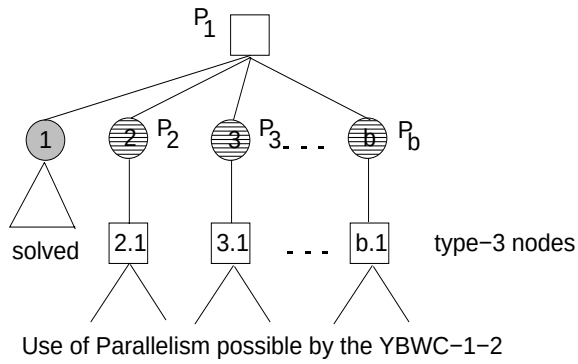


Figure 3.21: Using parallelism more rashly with the YBWC-1-2

The YBWC-1-2 is easily implemented by replacing the word "free" in Figure 3.6 on page 54 by the notion "YBWC-1-2-free" defined below.

Definition 3.6 *(YBWC-1-2-free)*

A successor $v.j$ of a node v of the actual variation is called YBWC-1-2-free, if the following conditions hold:

1. *$v.j$ is YBWC-free or*
2. *$type(v) = 3$ and $v.j$ is free, where $type(v)$ is the expected type of v.*

The YBWC-1-2 allows a more extensive use of parallelism . For example, in the situation described in Figure 3.21 parallelism may be used directly: The left son of the root, 1, is completely evaluated. The right successors $2, \ldots, b$ of the root are distributed over some processors by the network master. These nodes are type-2 nodes, since we expect the left son of the root to become the best successor of the root, and therefore a cutoff to occur at the right sons. The processors, which search the nodes $2, \ldots, b$, generate the left successors $2.1, \ldots, b.1$ von $2, \ldots, b$. These are the first successors of type-2 nodes and therefore are type-3 nodes. The pure YBWC at this moment would not allow to evaluate the successors of $2.1, \ldots, b.1$ in parallel, since the first successor is not yet evaluated. In the less restrictive version of the YBWC-1-2 all successors of the nodes $2.1, \ldots, b.1$ may be searched in parallel.

It is clear, that using the YBWC-1-2 we take the risk of increased search overhead to improve the work load of the processors.

3.7.2 The Extended YBWC for Type-2 Nodes

In the above section, we discovered a class of nodes, at which the immediate use of parallelism may be superior to the delay caused by the YBWC. In this section we will have a look at a class of nodes, at which the use of parallelism possibly should be delayed further, even if no cutoff occurred after the evaluation of the left successor. By the definition of *type* from the previous section we expect a cutoff at all the type-2 nodes. However, it is not clear whether a cutoff will ever occur at such a node. If the evaluation of the first successor does not cause a cutoff, then

the YBWC allows a parallel evaluation of all the right successors. This may result in search overhead, if one or more of the right successors cause a cutoff or at least an improvement of the bounds. Because of good move ordering, however, the probability is high, that such an improvement of the bounds or a cutoff resp. will happen after the evaluation of one of the first of the right successors.

Thus, we will define promising successors in the following way:

Definition 3.7 *(Promising successors)*

1. *The first successor is promising.*
2. *Every successor, which is reached via a positive capture move, is called promising. Here a "positive capture" is defined as on page 31.*
3. *Every successor, which is reached via a killer move, is called promising.*
4. *All the other successors are not promising.*

With the help of the promising successors we will now define the notion of *YBWC*-free*, which leads to an extension of the YBWC.

Definition 3.8 *(YBWC*-free)*

A successor $v.j$ of a node v of the actual variation is called YBWC-free,, if the following conditions hold:*

1. *$v.j$ is YBWC-free and*
2. *If $type(v) = 2$, then all the promising successors of v are completely evaluated. Here $type(v)$ is the expected type of v.*

The YBWC* is easily implemented by replacing the word "free" in Figure 3.6 on page 54 by "YBWC*-free".

> **YBWC*:**
>
> The YBWC may be extended to the YBWC* by allowing the use of parallelism at type-2 nodes only if all the promising successors are completely evaluated.

It is clear, that this more careful use of parallelism will decrease the processor work load. It should, however, decrease the search overhead, as well.

3.7.3 Combining the YBWC-1-2 and the YBWC*

The changes of the YBWC, which are necessary to define the YBWC-1-2, rely only on the type-3 nodes, the changes necessary for the YBWC* rely only on the type-2 nodes. Thus, combining both modifications to the YBWC-1-2* is possible. A definition of *YBWC-1-2*-free* is straightforward.

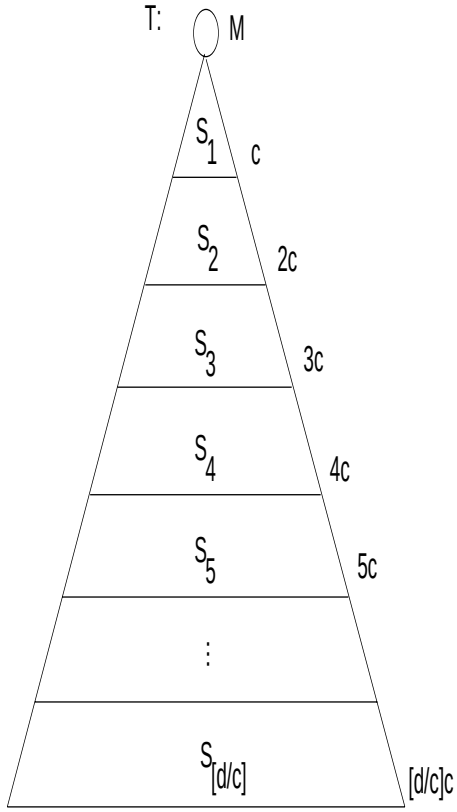


Figure 3.22: Layers for local search for work

3.8 Load Distribution Strategies

In Section 3.4, we described our method of how idle processors try to establish a master-slave relationship. One part of the method is the random choice of a potential master, to which a request for work is then sent. Any processor may be chosen as a potential master. In the next section we will discuss, why such a global load distribution is necessary, if processors at any time are allowed to keep only one subproblem.

3.8.1 Necessity of a Global Search for Work

Let $G = (V, E)$ be a processor network, let M be the network master of G. Let us assume, that the request of an idle processor v can reach only processors having a distance of $\leq c$ from v for some $c << max\{\Delta_G(M, u) \mid u \in V\} =: d$. Then let T be the breadth first search tree of G with root M, as described in Figure 3.22. If all processors of levels $(k-1) \cdot c + 1, \ldots, k \cdot c$, $k \in \{1, \ldots, \lceil \frac{d}{c} \rceil\}$, in T are combined to a layer S_k, then the root of any subproblem, which is received by a processor of layer S_i, has depth at least i in the game tree to be searched. This is easily seen by recognizing, that any processor, which received a subproblem rooted at w, generates subproblems for transmission only in the levels below w, because the subproblem rooted at w is the only subproblem the processor is working on. The above statement, then, easily follows from the fact, that a request for work of a processor in layer S_i is not able to

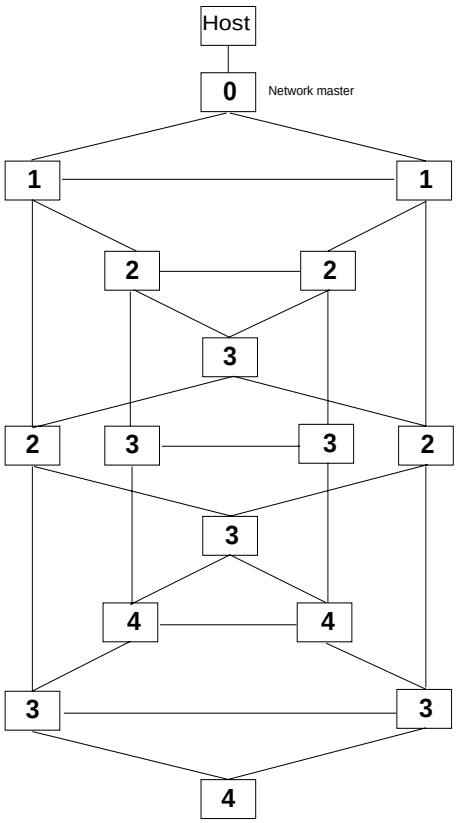


Figure 3.23: Local load distribution with $c = 1$ in the $DB(4)$

reach any processor of the layers S_j, $j < i - 1$. Thus, processors, which are located in the deep layers of T, only get very small subproblems. They are idle more often than processors in the higher layers. A uniform load distribution is hardly realizable. Figure 3.24 shows the situation described for $c = 1$ in the unmodified $DB(4)$. Six processors will receive subproblems of depths ≥ 3 only, three processors subproblems of depths ≥ 4 in the game tree. We get the following:

Corollary 3.1 *(Local load distribution)*

Under the assumption, that any processor at any time may only work on one subproblem, a load distribution, which is restricted to the close surroundings of the processors, causes a nonuniform and therefore a bad processor work load.

With this it is clear, that a good processor work load will not be achievable by a local search for work only. In the following section we present a method for load distribution, which in situations of good processor work load very often is successful using a local search for work, but, if necessary, switches to a global search for work automatically.

3.8.2 Local Search for Work

For our algorithm, we do not like the idea of allowing processors to work on several subproblems at a time, e.g. in time sharing. In this case masters may become idle, because they have to wait for a return of a slave, which searches the critical subproblem slowly, in time sharing with other subproblems. Thus, an idle processor is not able to speed up its search for work by sending several requests for work at the same time. Such a parallel search for work would cause the processors to get more than one subproblem.

Therefore, every processor may send a second request only if it received a cancellation for the former request. Any request as well as any cancellation on the average have to be routed over the average distance between two processors in the processor network, because a potential master is chosen randomly. The CPU-time for the routing may be saved, if processors no more choose a processor randomly from the whole network but randomly choose one of its neighbors in the processor network. However, restricting all the searches for work in this way results in a bad processor work load as stated in Corollary 3.1.

This bad processor work load can be avoided by supporting the above described local load distribution strategy by a global load distribution strategy: The destination processor of a request passes the request on to one of its neighbors (randomly chosen) instead of sending a cancellation.

With this all, the cancellations would be superfluous. A request would be passed on, until it eventually comes to a processor, which is able to send a subproblem to the initiator of the request. The cancellation messages, however, additionally are used to control the communication load in the parallel system: With a cancellation, the receiver gets some information about the communication load at the sender of the cancellation. This local communication load is interpreted as an approximation for the global communication load in the system. With the help of this information the receiver of a cancellation computes the length of a delay, after which it is allowed to start another search for work. The use of this method was established in Section 3.4.4. In a version where processors do not send cancellation messages, idle processors are lacking information about the communication load in the network.

For this reason, we allow passing on a request only for c times, for some constant c. If a processor gets a request for work, which has been passed on less than c times, then it passes the request on to a randomly chosen *neighbor* of itself . When it gets a request, which has been passed on c times, then it sends a cancellation message to the initiator of the request. This cancellation then contains information about the communication load at the sender. Since a request for work is passed on only to direct neighbors, processors will not be overloaded by additional routing tasks.

The above two proposals to speed up the establishment of a master-slave relationship can be implemented as follows:

N1:

Processors send their requests to a randomly chosen *neighbor* instead to a processor somewhere in the network.

N2:

If a processor P_1, which is not allowed to submit any subproblem, gets a request initiated by a processor P_2, which is passed on less than c times, then it passes this

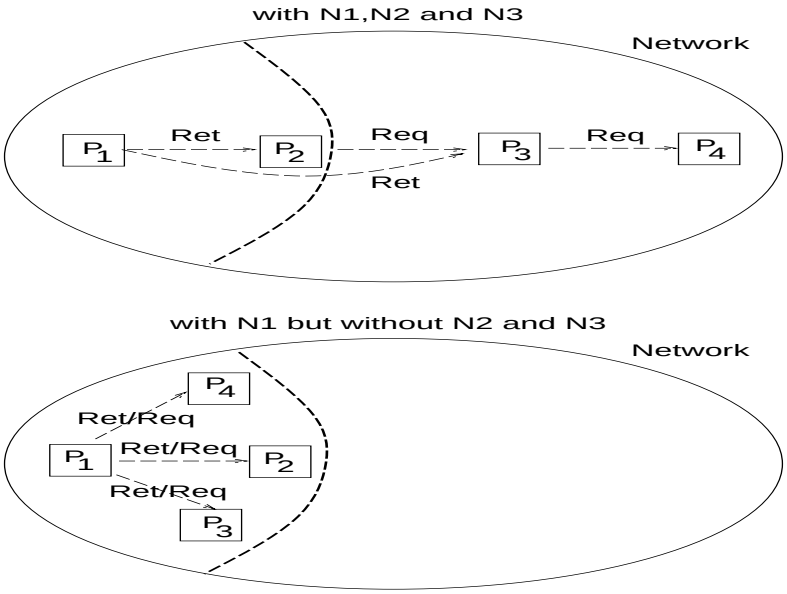


Figure 3.24: Local and global search for work

request on to a randomly chosen neighbor of itself. If it gets a request, which has been passed on c times, then it sends a cancellation message to the initiator of the request.

We call c the *radius constant* for N2. In our program $c = 8$. N1 restricts the search for work to the direct neighbors of the processors, N2 allows a search for work in a surrounding of radius $c+1 = 9$. If c is small compared to the maximal distance of a processor from the network master, then we have to deal with the consequences of a local load distribution as described in Corollary 3.1.

In the next section, we will introduce a method which, in conjunction with N1 and N2, allows a global search for work, without causing additional communication overhead.

3.8.3 Global Search for Work With Returns

The searches for work may be speeded up, if a request for work is sent to a processor, which is allowed to transmit a subproblem. Unfortunately, the information, whether a processor is allowed to transmit a subproblem or not, is not available to any other processor in the network. However, in many cases an idle processor is able to send its request to a processor, which at least is known not to be idle. After a processor finished working on a subproblem, it sends a return to its master. Afterwards it is idle and starts searching for work. The only processor, which at the moment of sending the return is known not to be idle, is its master. Thus, sending a request may be saved, if a master interprets a return message as a request for work additionally. With this, the search for work is started by sending a return message. We get the method

N3:

A return message additionally is interpreted by the receiver as a request for work.

A processor, which has finished the evaluation of a subproblem, begins its search for work with the return. Processors which lie between the sender and the destination of the return, route the message to its destination without reacting upon the request for work contained implicitly.

This method saves a request for work, since the return has to be sent anyway. Additionally, the method N3 combined with the method N2 guarantees a global load distribution, even if the maximal distance of a processor from the network master is large compared to the radius constant $c = 8$. This is made clear in Figure 3.24. In a version with N1 but without N2 and N3, requests and returns of a processor P_1 are not able to leave the direct neighbors in a version with N1 and N2 but without N3 they will not leave a surrounding of P_1, which is specified by the radius constant c of N2. In a version with N1, N2 and N3, however, a former master P_2 of P_1 may, after getting a return form P_1, pass a request on to a processor P_3 outside the surrounding of P_1. P_3 will become master of P_1 and after getting a return from P_2 it may pass on a request to P_4, etc. We get the following

Corollary 3.2 *(Use of N1)*

A global load distribution may be realized even if N1 is used by using N2 and N3 or N2 with the radius constant c large enough respectively. Using N1 without both N2 and N3 or without N2 with a radius constant large enough will result in local load distribution.

There is another reason, why the use of N3 makes sense: A processor will get its subproblems from the same master as long as its master has some subproblems to transmit, when it receives a return. If its master is not allowed to transmit any subproblem, then, in most cases, the evaluation of the masters subproblem is already finished or will be finished soon. Thus, processors searching for work according to the method N3 often stay in the same subtree of the whole game tree, until this subtree is completely evaluated. This has some positive influences on the killer lists and the history tables used to obatin a good move ordering. The probability to find a good killer move in the killer list is higher, if the former subproblem of the processor lies close to the current subproblem in the game tree. This improvement of the move ordering causes a decrease of the search overhead.

3.8.4 Combining Local and Global Search for Work

Since the methods N3 and N1 contradict each other, we will use the following precedence.

1. If a processor did not work on a subproblem before or did not need to send a return for the last subproblem or got a cancellation, then it starts its search for work according to rule N1.
2. If a processor just finished working on its subproblem, then it starts its search for work according to N3.
3. N2 is used in any case.

The effect of the use of the methods N1, N2 and N3 in a 256 processor DeBruijn network are described in Section 4.7. The effects of N1 in a 1024 processor grid are described in Chapter 5 on page 136.

Chapter 4

Performance on up to 256 Processors

This chapter is divided into three parts: In Section 4.1 we define the main measures necessary to illustrate the efficiency of our distributed algorithm. In Sections 4.2 and 4.3 we describe the performance of the best version on up to 256 processors. The last part of this chapter contains the descriptions of experiments we made to investigate how some parts of the distributed program contribute to the overall efficiency of the program.

4.1 Measures

In order to describe the efficiency of a parallel algorithm one usually looks at the speedup, which the parallel algorithm achieves compared to the best sequential one. Unfortunately, for the game tree search problem, the best sequential algorithm is not known. The Observations 2.4 and 2.5 show that our sequential algorithm traverses the game trees to be searched in a very efficient manner. All the domain dependent parts of the program are used by the sequential algorithm in exactly the same way as by the parallel algorithm. The parallel version differs from the sequential one only by the configuration description. Therefore, we use the single processor version of our algorithm as a basis for the speedup obtained by the use of parallelism.

Thus, the *speedup* of a parallel evaluation of a problem P is the quotient of the time needed by the sequential algorithm for P, $t_1(P)$, and the time needed by the parallel algorithm using n processors for P, $t_n(P)$.

For the following definitions, let $\mathcal{P}$ be a set of positions. A parallel version, a search depth and a parallel evaluation of the positions in $\mathcal{P}$ should be fixed.

Definition 4.1 *(Speedup, Efficiency)*

The speedup of a parallel evaluation with n processors on a set $\mathcal{P}$ of positions is defined to be

$$SPE(n) = \frac{\sum_{P \in \mathcal{P}} t_1(P)}{\sum_{P \in \mathcal{P}} t_n(P)}.$$

With this notion the efficiency is defined to be

$$EFF(n) := \frac{SPE(n)}{n}.$$

The speedup of a parallel version is influenced by several factors: One of these is the processor work load:

Definition 4.2 *(Processor work load)*

For a problem P, let $w_n(P)$ be the time spent by n processors to execute their processes "iteration" during the evaluation of P. Then the average (processor) work load is defined to be

$$LOAD(n) = 100 \cdot \frac{\frac{1}{n} \cdot \sum_{P \in \mathcal{P}} w_n(P)}{\sum_{P \in \mathcal{P}} t_n(P)}.$$

The average work load describes the percentage of the time necessary to execute the tree searching process with respect to the total evaluation time. However, in addition, $w_n(P)$ contains the time, which is used during the parallel tree search for routing messages. This was explained in more detail on page 61. A good average work load only guarantees, that the processors were not idle for long periods of time.

A second factor influencing the speedup, is the number of game tree nodes, which are generated by the sequential and by the parallel algorithm.

Definition 4.3 *(Search overhead)*

For a problem P, let $k_1(P)$ be the number of nodes, generated by the sequential algorithm and let $k_n(P)$ be the number of nodes generated by n processors evaluating P. Then we define the average search overhead to be

$$SOVD(n) = 100 \cdot (\frac{\sum_{P \in \mathcal{P}} k_n(P)}{\sum_{P \in \mathcal{P}} k_1(P)} - 1).$$

The search overhead is negative, if the parallel algorithm generates fewer nodes than the sequential one. It represents the percentage of nodes, which the parallel algorithm generates more (less) than the sequential one. The nodes, which are visited several times, e.g. for a research, are counted when they are visited for $k_1(P)$, as well as for $k_n(P)$.

The sequential and the parallel algorithm do not generate the nodes of a game tree with the same speed. With the help of the following definition we want to quantify the loss of performance described in Observation 3.1 on page 63:

Definition 4.4 *(Performance)*

The average performance is defined to be

$$PERF(n) = 100 \cdot \frac{\frac{\sum_{P \in \mathcal{P}} k_n(P)}{\sum_{P \in \mathcal{P}} w_n(P)}}{\frac{\sum_{P \in \mathcal{P}} k_1(P)}{\sum_{P \in \mathcal{P}} t_1(P)}}.$$

Here $\frac{\sum_{P \in \mathcal{P}} k_n(P)}{\sum_{P \in \mathcal{P}} w_n(P)}$ is the number of nodes generated by a processor in the n processor system on the average per second of pure working time. $\frac{\sum_{P \in \mathcal{P}} k_1(P)}{\sum_{P \in \mathcal{P}} t_1(P)}$ is the number of nodes, which are

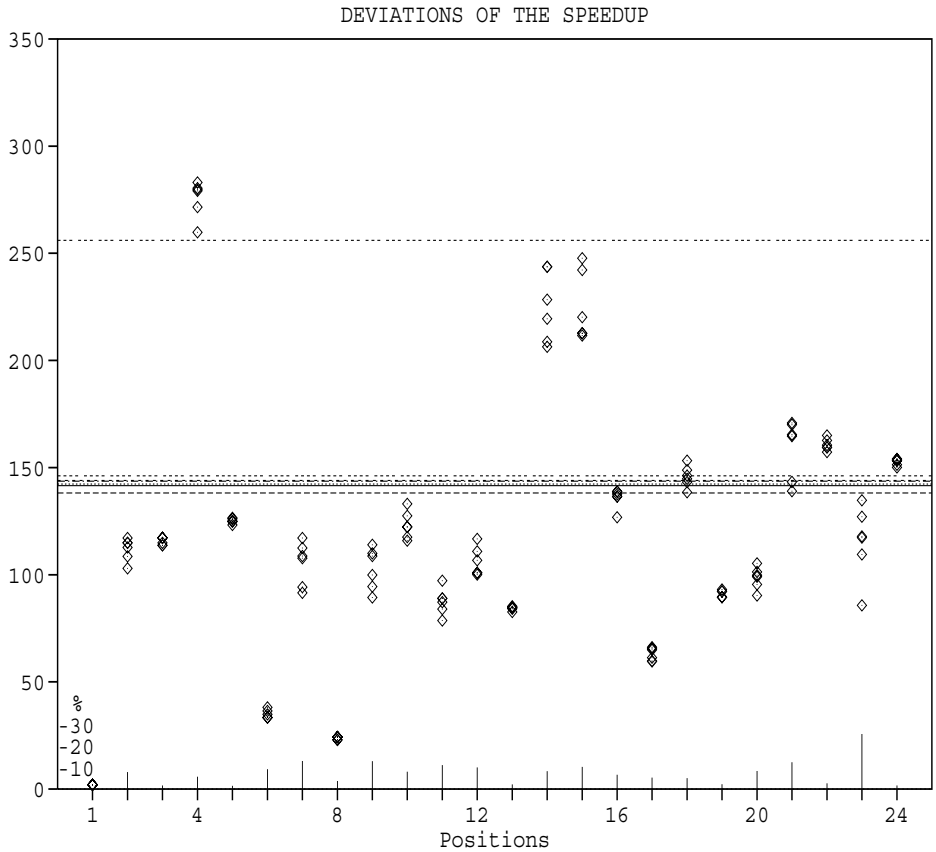


Figure 4.1: Deviation of the speedup

generated by the sequential algorithm per second. In the sequential case $t_1(P) \approx w_1(P)$. The average performance thus contains the percentage of the number of nodes searched per second in the parallel version with respect to the number of nodes generated per second in the sequential version.

If the average work load, the search overhead, and the performance are the only factors influencing the speedup, then the *calculated speedup*

$$SPE^* = \underbrace{n}_{Proc.} \cdot \underbrace{\frac{PERF}{100}}_{performance} \cdot \underbrace{\frac{LOAD}{100}}_{workload} \cdot \underbrace{\frac{100}{100 + SOVD}}_{work^{-1}} = n \cdot \frac{PERF \cdot LOAD}{100 \cdot (100 + SOVD)}$$

of a parallel execution with n processors should equal the measured speedup. For all the experiments considered in this work, SPE^* indeed is an excellent approximation of the measured speedup. Thus, we believe that no other factors have any influence on the speedup.

4.2 Measuring of the Nondeterministic Behaviour

The distributed algorithm presented in this work is highly nondeterministic, if more than one

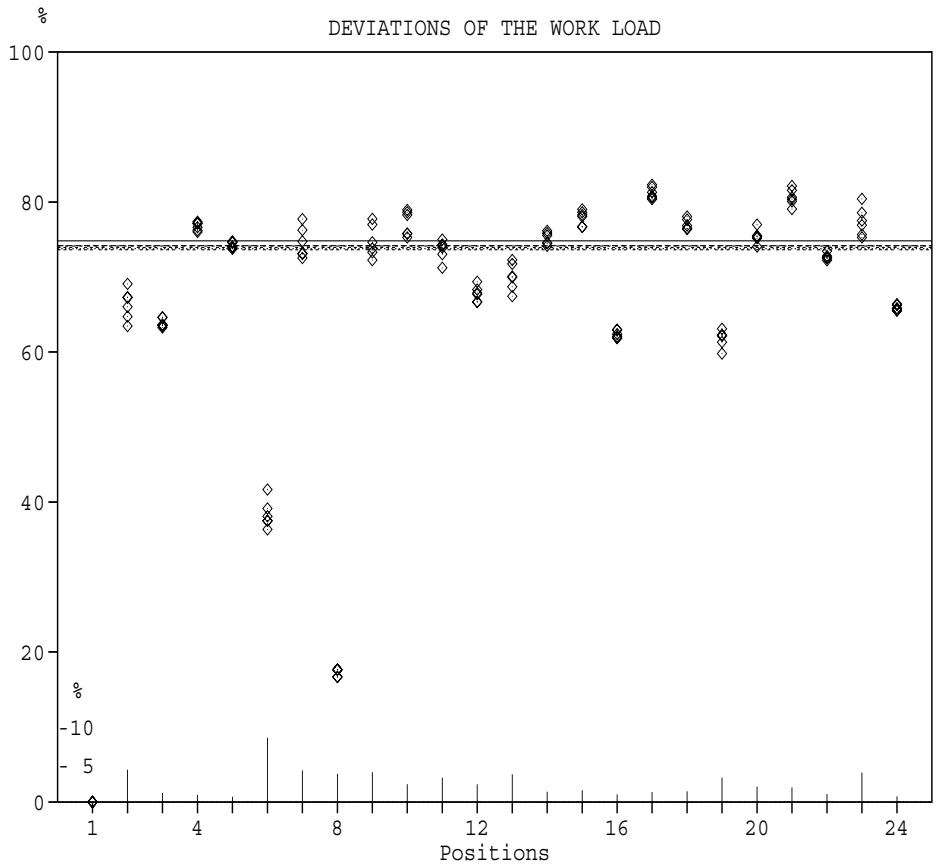


Figure 4.2: Deviation of the work load

processor is used. By this nondeterminism a different behaviour may occur in different evaluations of the same game tree. Of course, the result computed, the minmax value of the root of the game tree is always the same, but the subtrees searched to get this result may differ. Different best moves may be computed, also, for the root, if two or more successors of the root have the same minmax value, and this value is greater than or equal to all the other minmax values of its brothers. The last case, however, has not been observed in any of the experiments evaluating the positions of the Bratko Kopec test.

The following characteristics of our program cause nondeterministic behaviour:

1. The computer architecture.
   If two messages arrive at a processor at the same time via two different channels, then the processor by specification decides nondeterministically, which message is accepted first.

2. The random choice of the master.
   Here nondeterminism was introduced, because in earlier experiments [20] the load distribution obtained by a random choice of a master was better than the load distribution obtained by deterministic algorithm.

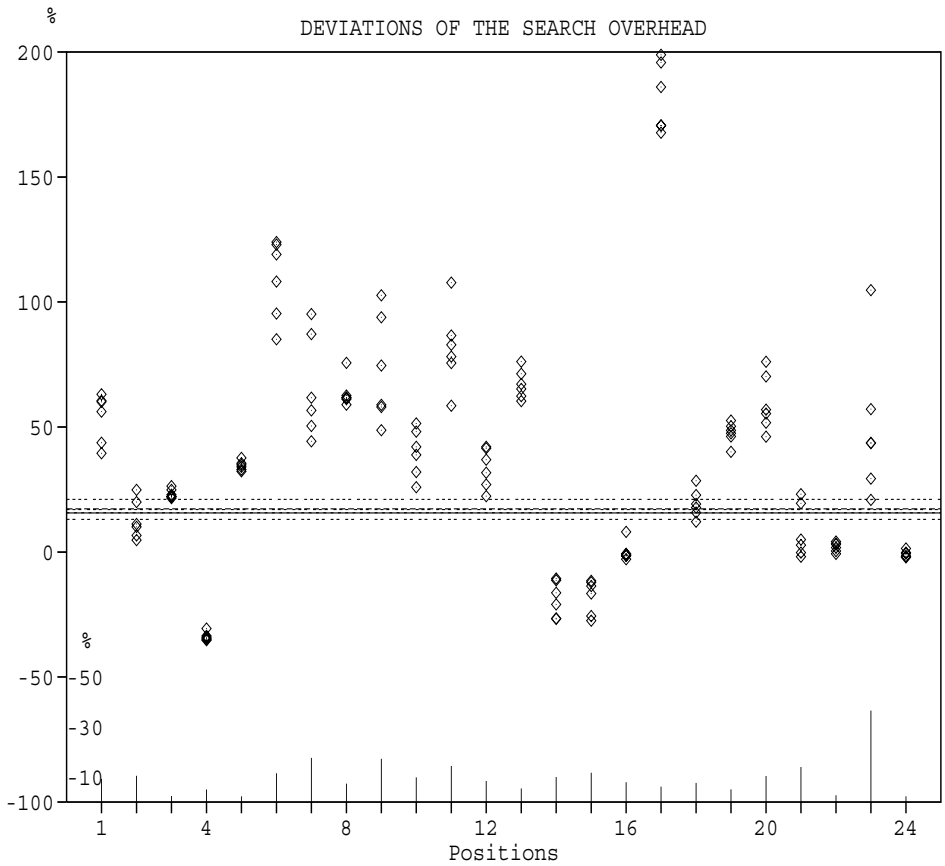


Figure 4.3: Deviation of the search overhead

The nondeterminism introduced by the architecture used, and the random choice of a master, causes two characteristics of the program:

1. The distributed transposition table.
   During their search, all processors have access to the distributed transposition table. Here, the search behaviour may differ from test to test, if in some test a result can be used from the transposition table, which was stored before the TR-request was served, and in another test the TR-request reaches the processor earlier than the message, which stores the entry required.

2. Different results from the move ordering mechanisms.
   Besides the transposition table, the killer and the history heuristic are important heuristics for the move ordering. Killer lists as well as history tables are kept local on every processor. Therefore, their contents depend upon the subproblem solved by the corresponding processors so far. By the randomized choice of potential masters, it is determined nondeterministically, which parts of the game tree a processor has to search. Therefore, the contents of the heuristics is determined nondeterministically, also. As a result, the move ordering is nondeterministical, which may have positive as well as negative consequences

for the complexity of the search.

In order to investigate the deviation of the parallel running times, six 8-ply searches have been performed by the 256 processor version on the 24 positions of the Bratko Kopec test set. Figure 4.1 shows the running times of the 256 processor version for these six runs, presented as the speedup compared to the sequential version (diamonds). Some positions show varying results. The greatest deviations can be seen for the position $B23$: the minimal speedup is 85.74, the maximal speedup is 134.74. The mean value of the speedups for position $B23$ is

$$\frac{109.48 + 117.90 + 85.74 + 134.74 + 127.06 + 117.33}{6} = 115.38,$$

and the maximal deviation from this mean value in percent compared to the mean value is

$$\frac{115.38 - 85.74}{115.38} \cdot 100\% = 25.69\%.$$

A maximal deviation of more than 10% can be observed for the positions $B07$, $B09$, $B11$, $B12$, $B15$ and $B21$, as well. Other positions, such as $B03$, $B05$, and $B24$, show a uniform behaviour having maximal deviations of less than 2%. The maximal deviations from the mean value of the speedups in percent compared to the average speedup can be found in Figure 4.1 by looking at the small vertical lines on the x-axis. The superlinear speedup for position $B04$ is not caused by one hyperfast run. The six values measured for the speedup vary between 259.77 and 279.75 with a mean value of 275.60. This corresponds to a maximal deviation from the mean value of 5.74% only.

The variation of the speedup as an average over the whole test set is described by the six horizontal lines lying close together. The average speedup varies between 146.13 and 138.12. The mean value of these average speedups is

$$\frac{146.12 + 141.53 + 138.12 + 144.00 + 142.32 + 143.62}{6} = 142.62.$$

Thus, the maximal deviation of this average speedup from the mean value is 3.2%. The tests resulting in a speedup of 146.12 and 138.12 ,respectively, seem to have a small probability to occur. All the other values are very close to the mean value.

In Figures 4.2 and 4.3 the deviations of the work load and the search overhead respectively measured in the above described tests are shown. Only very small deviations can be observed for the processor work load even for single positions. These result in a small deviation of the average work load. The mean value of the six average work load measured is

$$\frac{73.62 + 73.93 + 74.81 + 74.18 + 74.13 + 73.77}{6}\% = 74.07\%.$$

The maximal deviation from the mean value is very small: 0.9%. Similarly, the average performance is nearly a constant. The mean value is

$$\frac{87.73 + 87.54 + 87.34 + 87.73 + 87.91 + 88.11}{6}\% = 87.73\%,$$

the maximal deviation is 0.45%. For the search overhead, however, the correlation with the parallel running times measured is obvious: positions showing varying running times have varying search overhead too. The inverse is also true, if the positions $B01$ and $B06$ having parallel

running times of 1 and 20 seconds respectively are ignored. Thus, even for the average search overhead of the six tests greater deviations have been measured:

$$\frac{13.03 + 16.98 + 21.06 + 15.66 + 17.30 + 15.71}{6}\% = 16.62\%.$$

The maximal deviation from the mean value is 26.7%. This number, however, describes the deviation of the search overhead. The maximal deviation of the sizes of the game trees searched by the parallel algorithm can be calculated as

$$(\frac{100 + 21.06}{100 + 16.62} - 1) \cdot 100\% = 3.81\%.$$

Thus we established the following:

Corollary 4.1 *The deviations of the parallel running times are mainly caused by the deviations of the search overhead.*

The nondeterminism in the program causes the transposition table,, the killer lists and the history heuristic to have different contents during different runs on the same problem. These heuristics heavily influence the move ordering, which itself has large influence on the search overhead. Thus, the above corollary is not astonishing.

The above presented data, however, shows a feature of the parallel algorithm, which decreases the deviations of the running times: If the search overhead increases, then the work load increases too. Thus, for example, for the first run the average search overhead is 13.03% and the average work load is 73.62%, for the third run, the average search overhead of 21.06% causes the average work load to increase to 74.81%. This results in a smaller deviation of

$$\frac{146.13 - 138.12}{142.62} \cdot 100\% = 5.62\%$$

for the running times than one would have expected from the deviation of the sizes of the game trees searched:

$$\frac{121.06 - 113.03}{116.62} \cdot 100\% = 6.89\%.$$

The increase of the average processor work load can not compensate for the increase of the search overhead. However, it decreases the effects of the deviations of the search overhead. As a concluding observation we have:

Observation 4.1 *Measuring the deviations shows, that the Bratko Kopec set of test positions contains enough positions, so the deviations of the parallel running times only have minor influence on the average speedup of the whole test set.*

But the measurements show, that two different versions of the program may not be assessed by a few results, obtained by solving single positions, if these results differ only in a few percent of the efficiency. In many cases, this makes it difficult to assess success or failure of a modification of the distributed algorithm. The assessment of every modification by measuring the deviations as above would be too complex. Thus, in this book, measuring of the deviation of two different versions of the program was used only, if the two versions could not be compared by a single run, because of the above described deviations. Since the deviation of the average speedup is quite small, most of the comparisons in this work could be made by running each of the positions of the Bratko Kopec set once.

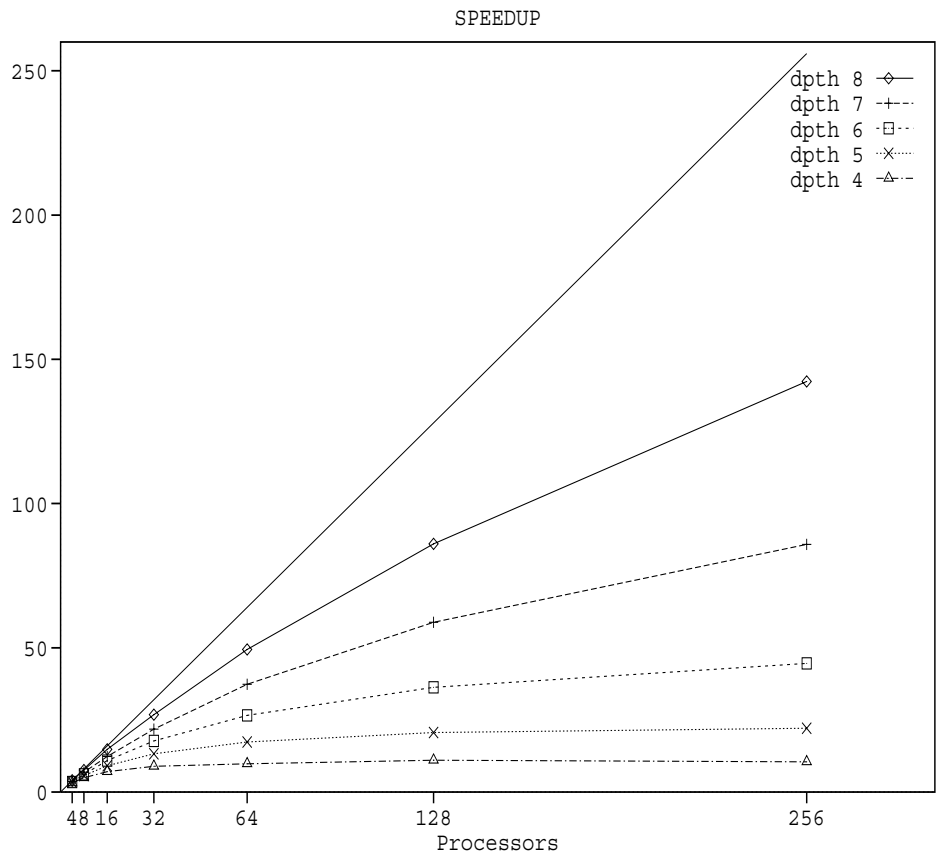


Figure 4.4: Speedups for several search depths

4.3 Behaviour of the Best Distributed Version

4.3.1 Speedups for Several Search Depths

DeBruijn Networks with 4 - 256 Processors

In this section, the behaviour of our distributed algorithm running with 4 to 256 processors will be investigated. All the results are obtained from searches of the 24 positions of the Bratko Kopec set of test positions to varying search depths. The processor network used for all processor numbers considered is the corresponding DeBruijn network. The program uses the YBWC-1-2* and the combination of N1, N2 and N3 described in Section 3.8 on page 83. This version will be denoted as *standard version*. For the 256 processor version we discuss the data resulting from the run, which comes closest to the mean values calculated in Section 4.2. This is the fifth run in Section 4.2 with $SPE = 142.32$, $LOAD = 74.13\%$, $SOVD = 17.30\%$, and $PERF = 87.91\%$.

We will analyze the speedup measured by considering the average work load, search overhead, and performance of the program and presenting reasons for these data. The equality of the calculated speedup SPE^* and the measured speedup shows, that only these three factor influence

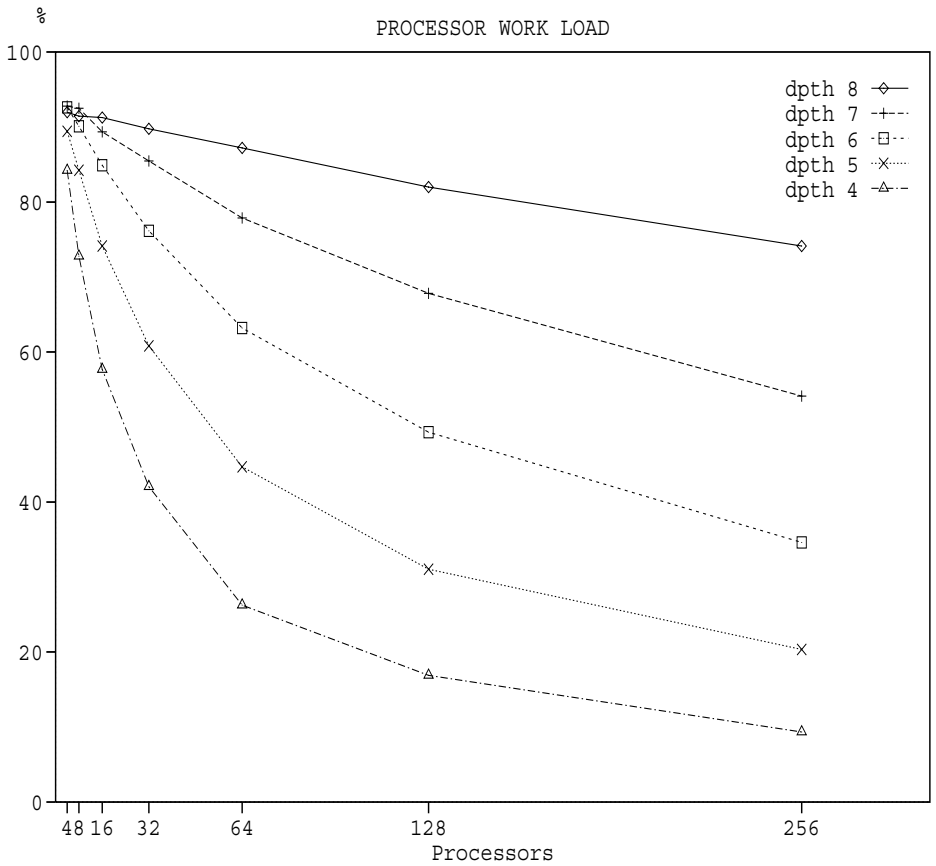


Figure 4.5: Processor work load for several search depths

the speedup.

Figure 4.4 shows the measured speedup for processor numbers from 4 to 256 and search depths from 4 to 8. As expected the speedup increases with increasing search depths, i.e. with increasing running times, for every processor number considered. The reason becomes apparent with Figures 4.5 and 4.6: With increasing search depth, the average work load increases and the average search overhead decreases.

4.3.2 Reasons for the Speedups Measured

The results for the speedup presented above are due to an increase in the average work load and a decrease in the average search overhead with increasing search depths. Searching larger game trees makes the load distribution easier, such that the increase in the average work load is as expected. The decrease in the search overhead in large game trees, however, is not such obvious at once. We will try to explain this behaviour in the following.

1. Overload of the transposition table of the sequential version.
   The transposition table used by the sequential version has 160000 entries. It is overloaded

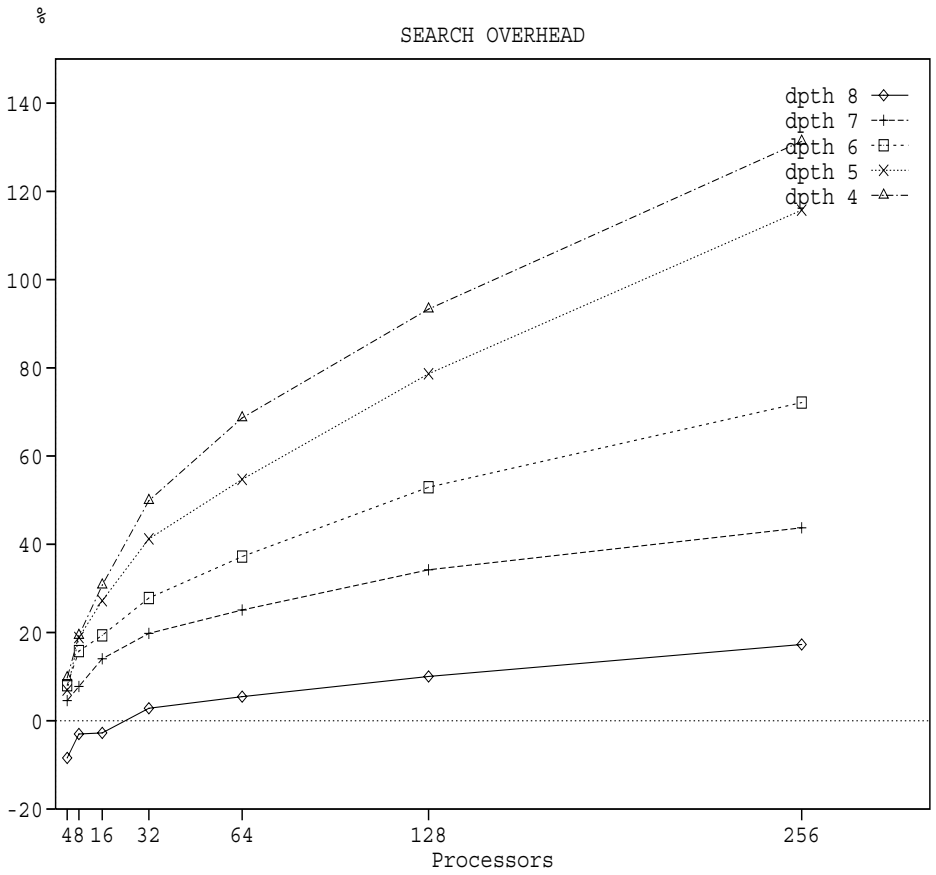


Figure 4.6: Search overhead for several search depths

when searching to search depths of 7 or 8 as stated in Observation 2.8 on page 42. Here, the parallel algorithm benefits from a larger and therefore not overloaded transposition table. In Section 4.5, we will investigate, how much the parallel algorithm benefits from the large transposition table during a search to a search depth of 8. We expect, that a parallel search to a search depth of 7 benefits from the large transposition table too, but the overall effect should be much smaller in this case.

2. Rash use of parallelism because of the bad work load in small game trees.
   In small game trees the processor work load is poor. Some processors may remain idle during the whole computation. Therefore, the type-3 nodes do not suffice to keep all the processors busy. Parallel evaluation is used more often to search the successors of type-2 nodes. If one of these successors causes a cutoff, the evaluation of all the right brothers is superfluous. This happens more often during the search of small game trees than during the search of large game trees, because the average work load is poor.

3. Rash use of parallelism because of a bad approximation of the real type of nodes in small game trees.
   During the evaluation of the small search depths, results are stored into the transposition table, as well as the killer lists and the history tables, which have been computed without

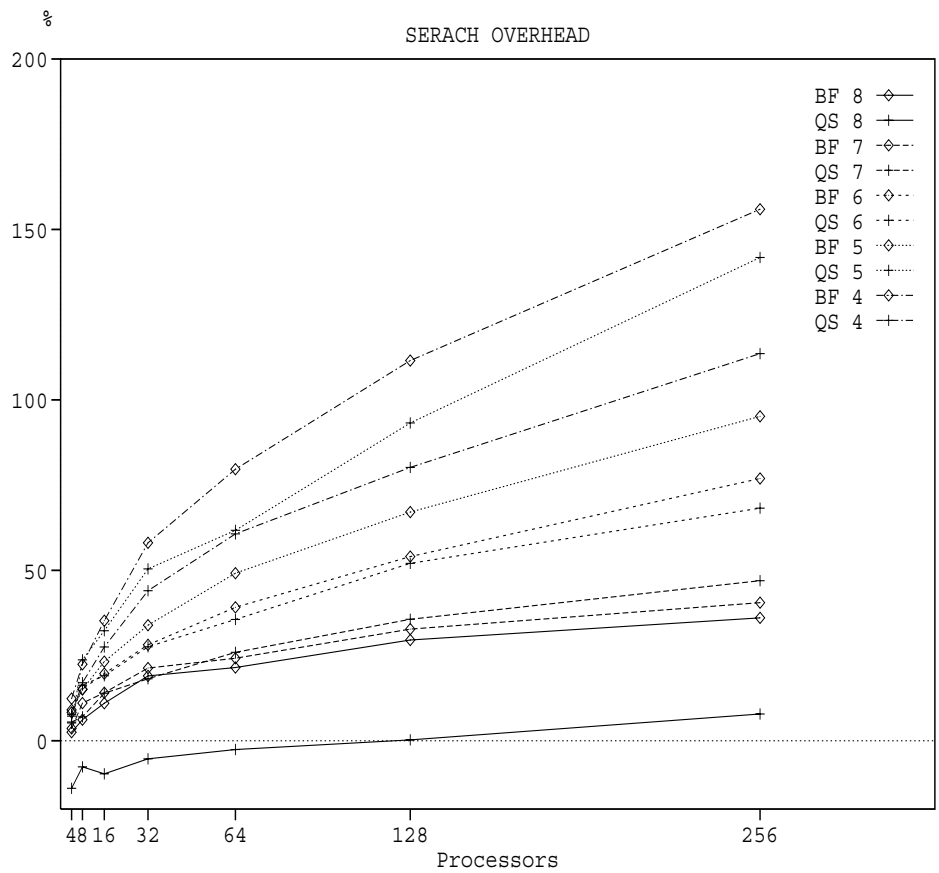


Figure 4.7: Search overhead in the BFS and QS for several search depths

quickly. These results allow a better move ordering during the searches to larger search depths and thus speed up these searches. Better move ordering can be expressed by a more exact approximation of the real types of nodes by the expected types of nodes.

Table 4.1 shows the percentage of the nodes, for which the expected type differs from the real type, in the brute force tree of the game tree (BF) and in the quiescence search (QS) for a run with 256 processors. The expected type of nodes gets better and better with increasing search depths. Thus, the probability to use parallelism in a sensible way increases with the search depth.

Corollary 4.2 *The average search overhead decreases with increasing search depth, since the transposition table of the sequential version is overloaded and because parallelism is used more rashly in small game trees.*

Figure 4.7 shows the values measured for the average search overhead for the 4- to 8-ply searches with up to 256 processors. The search overhead for every search depth is separated into the search overhead in the brute force tree (BF) and the search overhead of the quiescence search (RS). It turns out, that the search overhead of the quiescence search is larger than the search

	BF		QS	
Tf	type 2	type 3	type 2	type 3
4	1.60 %	3.07 %	2.48 %	4.53 %
5	1.85 %	3.19 %	2.04 %	2.83 %
6	0.87 %	1.47 %	1.77 %	2.25 %
7	0.74 %	1.09 %	1.42 %	1.34 %
8	0.49 %	0.63 %	1.16 %	1.14 %

Table 4.1: Percentage of nodes having the wrong expected type

overhead in the upper part of the game tree if the search depth is odd, and smaller, if the search depth is even. The reason for this effect is a different ratio of the inner type-2 nodes to the type-2 leaves. During a search to a search depth of d, a parallel evaluation of the successors of a node in level $\geq d$ may cause search overhead only in the quiescence search, since all the successors of this node are quiescence nodes. A parallel evaluation of the successors of nodes in the levels $< d$, however, may cause search overhead in the upper part of the game tree as well as in the quiescence search. Here, the search overhead of the quiescence search mainly is caused by the fact, that for the nodes, the parallel algorithm visits more in the upper part a quiescence search has to be done too. Thus, the search overhead of the quiescence search nodes caused by this effect is proportional to the search overhead in brute force search.

In practice the game trees to be searched are very similar to the minimal game tree. The minimal game tree of a b/d - uniform game tree contains $b^{\lceil \frac{d}{2} \rceil} - 1$ type-2 nodes in level d. If one now looks at the ratio of the number of type-2 leaves to the number of inner type-2 nodes of the minimal tree of a b/d - uniform game tree, then one gets

$$\frac{b^{\frac{d+1}{2}} - 1}{2 \cdot \sum_{i=1}^{\frac{d-1}{2}} (b^i - 1)} \approx \frac{b}{2}, \text{ if } d \text{ is odd and}$$

$$\frac{b^{\frac{d}{2}} - 1}{(b^{\frac{d}{2}} - 1) + 2 \cdot \sum_{i=1}^{\frac{d-2}{2}} (b^i - 1)} \approx 1, \text{ if } d \text{ is even.}$$

In the minimal game tree of a b/d - uniform game tree of odd depth d the fraction of the type-2 nodes of level d is larger than if d is even. One can expect this to be similar in game trees searched in practice. Therefore, the search overhead measured for the quiescence search is larger than the search overhead in the brute force tree, if the search depth is odd. We get the following

Corollary 4.3 *The fact that the search overhead of the quiescence search is greater compared to the search overhead of the brute force tree, if the search depth is odd, is simply caused by the the location of the type-2 nodes and the method to measure the search overhead.*

The above considerations, however, do not give any reasons for the fact, that the search overhead of the quiescence search is smaller than the quiescence search in the brute force tree, if the search depth is even. A reason for this effect will be presented in the next section.

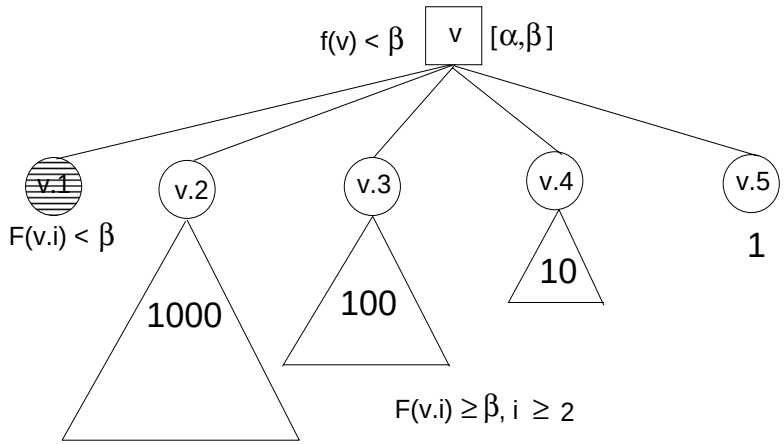


Figure 4.8: Negative search overhead in the quiescence search

4.3.3 "Superlinear" Speedup in the Quiescence Search ?

The quiescence search starts below the leaves of the brute force tree. In Observation 2.1 on page 35 we described the differences between the inner nodes of the brute force tree and the nodes of the quiescence search. At a quiescence node immediately after the computation of the static evaluation, or perhaps the attempt to generate quiescence moves, it is decided, whether this quiescence node becomes a leaf or not. In contrary, a node of the brute force tree of depth d is a leaf, if it is lying in level d of the tree. Even more than for subtrees of the brute force tree, approximations about the complexity of a quiescence search tree are very uncertain.

Therefore, situations as described in Figure 4.8 may surface in the quiescence search: The quiescence search node v is searched with the search window $[\alpha, \beta]$ The player to move is not in a dangerous situation, thus the value of the static evaluation $f(v) \leq \alpha$ is computed for v, which, unfortunately, does not suffice to produce a cutoff. Let us assume, that the left successor of v does not cause a cutoff at v. The other successors $v.2, \ldots v.5$. are generated. Now, $v.5$ leads to a quiet position immediately, such that $v.5$ becomes a leaf of the quiescence search. On the other hand, $v.3$ and $v.4$ do not lead to a quiet position. However, they do not create new threads, such that the quiescence search terminates after generating 100 or 10 further quiescence nodes respectively. The move from v to $v.2$, though, creates new threads, such that the quiescence search below $v.2$ has to generate 1000 quiescence search nodes. Let all the successors $v.i$, $i \geq 2$ of v have minmax values $F(v.i)$, which allow a cutoff at v. Then the complexity of the search of the sequential algorithm in this situation heavily depends on the ordering, in which the successors $v.2, \ldots, v.5$ of v are generated. If $v.2$ is generated as the second successor, then the sequential algorithm has to generate 1000 further nodes to get the cutoff at v. It is of value to note that any parallel evaluation of the successors $v.2, \ldots, v.5$ of v will stop after one round (generation of four nodes), because it then has computed the cutoff at v. Thus we derive the following

Observation 4.2 *Parallel evaluation of the nodes of the quiescence search is superior to the sequential quiescence search, if cutoffs can be computed in quite small, right subtrees and the subtrees to the left of this subtree are quite large.*

Even if during a parallel evaluation of a node in the brute force tree a similar effect may arise, situations similar to the one described above are much more likely in the quiescence search,

because of the characteristics of the evaluation of the quiescence search nodes. This effect decreases the search overhead in the quiescence search, if the search depth is even. Together with the use of the large transposition table during an 8-ply search the search overhead of the quiescence search is negative with up to 64 processors, it is 0.26% with 128 processors and 7.90% only with 256 processors, although the search overhead of 36.05% in the brute force tree causes the generation of 35.70% more leaves of the brute force tree, at which a quiescence search has to be performed.

The advantage of a parallel search of the successors of a quiescence search node suggest a different sequential evaluation of a quiescence search node:

Corollary 4.4 *Observation 4.2 suggests, that at the nodes of the quiescence search a sequential breadth first search is superior to the usual depth first search.*

The directional depth first search, obviously, is not optimal for the quiescence search. The nonoptimality of directional searches was proven analytically for a special class of OR-trees by Althöfer [5]. For AND/OR-trees or game trees we are not aware of any similar analytical results.

An implementation of the breadth first search suggested above would have required a completely different memory organization. Thus, for this thesis, no attempts were made to use breadth first search in the quiescence search. Presently, we do not know of a chess program using a breadth first search in the quiescence search.

4.3.4 Measuring the Performance

Figure 4.9 shows the loss of the performance measured for several processor numbers and several search depths as a function of the average distance of the processors in the corresponding network.

	Search depth			
	8	7	6	5
TR-Msg/nodes	0.28	0.21	0.31	0.20
QS/BF	1.58	1.05	1.19	0.98

Table 4.2: Number of transposition table messages

The performance loss for the 6-ply search is larger than for the other search depths. The reason for this becomes clear, is one compares the number of transposition table messages during a 6-ply search with the number of messages necessary during the other searches. Table 4.2 in the first line shows the number of transposition table messages, which have to be send per game tree node, for the searches to search depths $5, \ldots, 8$. This number is smaller for the searches to the odd search depths, and greater for the even search depths. The reason for this, again, is the different structure of game trees of odd and even depth.

Let us, idealized, assume that a b/d - uniform game tree has to be searched. The minimal tree of a b/d - uniform game tree of depth d contains

$$\sum_{i=0}^{d-1}(b^{\lceil \frac{i}{2} \rceil} + b^{\lfloor \frac{i}{2} \rfloor} - 1) = O(b^{\lceil \frac{d-1}{2} \rceil})$$

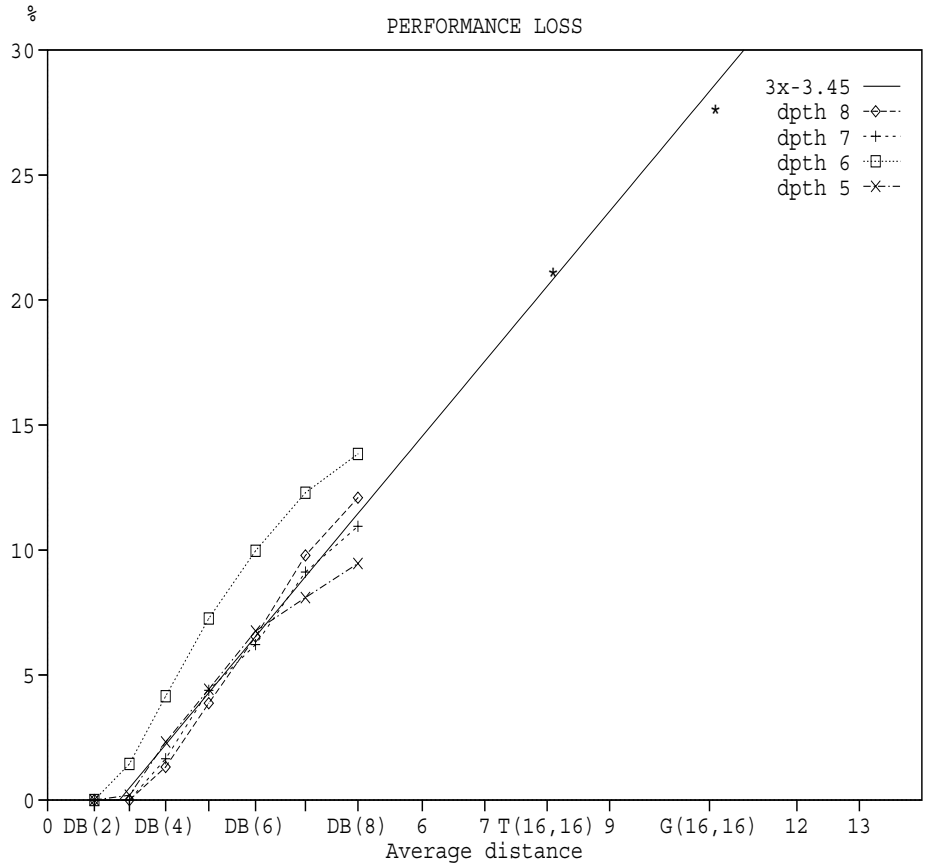


Figure 4.9: Loss of performance for several search depths

inner nodes and

$$b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1 = O(b^{\lceil \frac{d}{2} \rceil})$$

leaves. Since the game trees to be searched are very similar to the minimal tree, even in practice at a search to an odd search depth the ratio between nodes, which require a transposition table access (inner nodes), to nodes, which do not require such an access (leaves). This results in a smaller communication load during a search to an odd search depth.

The second line of Table 4.2 shows the ratio of quiescence search nodes to nodes of the brute force tree for a parallel search with 256 processors. This immediately makes clear, that the number of transposition table messages per node of the game tree searched is smaller, and thus the loss of performance is smaller during an 8-ply search compared to the 6-ply search. During the 6-ply search the quiescence search has roughly 1.19 times the size of the brute force search, during the 8-ply search this factor grows to 1.58. Since the nodes of the quiescence search do not require a transposition table access, the communication load is smaller during an 8-ply search.

Linear Approximation of the Performance Loss

The performance losses during the searches to the search depths of $5, 7$ and 8 are almost equal. The measurements are presented in Figure 4.9 as a function of the average distance of the processors in the network used. The straight line

$$f(x) = 3.00 \cdot x - 3.45$$

put into the diagram, is the solution of a linear approximation of the measures obtained from 8-ply searches using the DeBruijn networks with 16 to 256 processors as well as the performance losses measured for the 16×16 - Torus and the 16×16 - Grid. The latter are marked by a star, and are used to get a measurement for networks having a relatively high average distance. The behaviour of our program using the grid and the torus networks is discussed in more detail in Section 4.8. The residuum

$$\begin{aligned} r \quad = \quad & \underbrace{(1.38 - 2.55)^2}_{DB(4)} + \underbrace{(3.87 - 4.65)^2}_{DB(5)} \\ + \quad & \underbrace{(6.54 - 6.75)^2}_{DB(6)} + \underbrace{(9.78 - 9.15)^2}_{DB(7)} + \underbrace{(12.09 - 11.55)^2}_{DB(8)} \\ + \quad & \underbrace{(21.06 - 20.55)^2}_{T(16,16)} + \underbrace{(27.54 - 28.35)^2}_{G(16,16)} \\ \approx \quad & 3.63 \end{aligned}$$

of this linear approximation shows that the data measured correlates with the linear approximation very well. A linear approximation for the data of the 7-ply searches results in the function $f(x) = 3.01 \cdot x - 3.74$.

Corollary 4.5 *The very well correlated approximation of the measurements for the 7- and the 8-ply searches by the functions*

$$f(x) := 3.01 \cdot x - 3.74 \text{ or } f(x) := 3.00 \cdot x - 3.45,$$

respectively, indicates that the loss of performance in a wide range linearly depends on the average distance x of the processor network used.

Corollary 4.5 holds, at least, if the search depth is large enough to guarantee a sufficient processor work load and the average distance of the processor network used is not too large. Here, a large advantage of the DeBruijn topology compared to the grid topology becomes apparant: the loss of performance in a 16×16 - grid is 27.56% for an average distance of 10.6. If the performance loss depends solely on the average distance of the prcessors , then, with the same performance loss as for the 16×16 - grid, one could run a DeBruijn network having $2^{14} = 16384$ processors, since the average distance between two processors in the $DB(14)$ is just 10.53.

The reasons for performance loss are investigated in Section 4.4.

4.3.5 Speedups Under Tournament Conditions

The speedups of the 8-ply searches are precisely listed in Figure 4.3. However, not all of these speedups are relevant for practical use.

	processors							
	1	4	8	16	32	64	128	256
SPE (Dpt 8)	1.00	4.11	7.62	14.79	26.83	49.43	86.03	142.32
SPE↑	1.00	3.48	6.88	12.33	21.83	49.43	86.03	142.32
(Dpt,time)	(6,767)	(6,221)	(7,612)	(7,342)	(7,193)	(8,557)	(8,320)	(8,193)
SPE↓	1.00	3.42	6.13	10.91	21.83	37.34	58.82	142.32
(Dpt,time)	(5,163)	(5,48)	(6,125)	(6,70)	(7,193)	(7,113)	(7,72)	(8,193)

Table 4.3: Speedups under tournament conditions

If our algorithm is used in practice, then we have to fulfill the timing constraint, which is usual for chess games as well as for computer chess games. In a tournament game, these timing constraints usually allow an average of 3 min. = 180 sec. per move. By the use of opening book and endgame databases, as well as the use of the opponents time, the time, which may be used by the computer per move on the average, varies between 180 and 300 seconds depending mostly on the opponent. The use of the opening book and the endgame databases allow the computer to play a move directly (1 second). The opponents time is used in the following way: If a computer has computed a move, then it starts another computation, the so called *Permanent Brain*, during the time the opponent is thinking about his move. This is done by expecting the best opponents move, which it has computed before together with its last move, as an answer to its own move. It then starts a game tree search on the position resulting after an update of the expected answer. For example, if the principal variation of the d-ply search is $(v_0, v_1, \ldots, v_d)$, then the computer delivers the move from v_0 to v_1 as its best move, expects the opponent to play the move from v_1 to v_2, and then starts a game tree search on the game tree rooted at v_2. If the opponent actually plays the move to v_2, then the opponents thinking time was used for relevant work. If the opponent plays to a position $v_2' \neq v_2$, then the computer rejects the results computed in the permanent brain and starts a search at position v_2'. Experience shows, that when playing against medium and good human opponents, on the average roughly 50% of the opponents moves were predicted correctly, in a game against another computer slightly more than half of the moves ($55\% - 60\%$). Thus, we may argue that roughly half of the opponents time can be used for relevant computations.

If we allow an average time of 193 seconds for every position, then we have a more severe timing constraint than used in tournament games. The second line of Table 4.3 contains the speedups and the corresponding search depths and average computation times which are closest from above to the limit of 193 second. The third line contains the speedups of those search depths, which are closest to 193 seconds from below. The 256 processor version computes the 8-ply searches with an average computation time of 193 seconds per position, such that for this processor number an 8-ply search comes closest to the conditions during a tournament game. The other results indicate that even the 128 processor version is able to perform an 8-ply search under tournament conditions in most of the positions. Networks with 64 to 16 processors, however, allow only 7-ply searches, networks with 8 or 4 processors resp., are able to perform 6-ply searches and the sequential algorithm is not even able to search deeper than 5 plies.

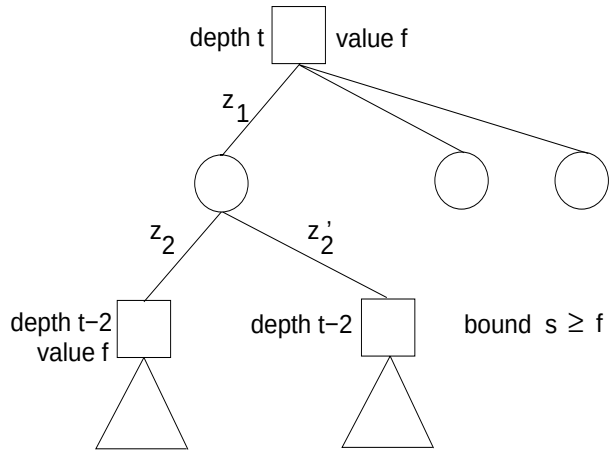


Figure 4.10: Saving the smaller search depths

Saving the Smaller Search Depths

The above described results were obtained by computing every single test position using the method of iterative deepening to the search depth required. Thereby, for every single position during an 8-ply search, all the game trees with search depths 1 to 8 were evaluated one after the other. Figure 4.4 on page 92 shows that the evaluations of these small game trees are done very inefficiently, at least if a large number of processors is used.

The following consideration shows that during a game the smaller search depths can be saved. The situations described below are elucidated in Figure 4.10 :

Let the computer have played a move z_1, which was computed to be the best move of a d-ply search. Afterwards, the computer started its computation in the permanent brain expecting its opponent to play a move z_2. Then the current root position has been searched (at least) to a search depth of $d-2$. Since the current root position has been on the principal variation of the previous d-ply search, the best move as well as its minmax value of a $d-2$-ply search is known. The computer may start its computation with a $d-1$-ply search. If the opponent plays z_2, then the searches to the small search depths $1, \ldots, d-2$ were saved.

If the opponent does not play z_2 but instead plays $z_2' \neq z_2$, then the computer has to restart its search on a position after the moves z_1 and z_2'. This position has been searched with a search depth of $d-2$ as well, however, in general only a bound instead of the minmax value has been computed during this search. Therefore, the best move in this position, in general, has not yet been computed. Yet this move has been sufficient to refute the move z_2' compared to the move z_2. In this case, the computer accepts the results of the $d-2$-ply search, as if they were the exact minmax value and the best move for the search depths $1, \ldots, d-2$, and again starts its search with a search depth of $d-1$. In phases of the game when the computer is in time trouble, the last case is somewhat critical, because the computer may not be able to finish the $d-1$-ply search started immediately in time. In this case, the only move the computer is able to play may be the cutoff move from the previous $d-2$-ply search. However, this move has refuted the opponents move z_2', and thus, in the worst case, is something like a "lost chance". If one wants to be somewhat more careful, then the last case may be handled is a way that the search at the

position after the moves z_1 and z_2' is started with a search depth of $d-2$.

In both cases the transposition table, as well as the killer and the history heuristic, all contain the information of the search depths 1 to $d-2$ or similar ones, such that an iterative deepening started with a search depth of 1 would not make sense. The computer starts with a $d-1$-ply search. We obtain the following

Corollary 4.6 *If our game tree search algorithm is used during a game, in which a move has been computed for a position v with a search depth of d, then the iterative deepening process for the next position to be searched may be started with a search depth of$d-1$.*

Under the assumption that all moves played by the computer have been the result of searches al to the same search depth d, then we may conclude that only for the first position to be searched in a game the iterative deepening process has to be started with a search depth of 1, all the other positions may be searched starting with a search depth of $d-1$.

Corollary 4.6 has some consequences for the speedups described above. For instance, the 256 processor version may save the searches to depths $1-6$ for every position. If we subtract the running times for the searches to depths $1-6$ of the sequential as well as the 256 processor version, then this results in a total time of 4226 seconds for the 256 processors and a total time of 641820 seconds for the sequential version running the 7- and the 8-ply searches for the 24 positions of the Bratko Kopec set. This corresponds to an average running time of 17 seconds per problem for the 256 processors, which is below the time allowed in a tournament game. The speedup increases to 151.87. This calculation shows that the speedup during a game seems to be larger than the speedup measured for the evaluation of single positions.

4.3.6 Efficiency

Figure 4.11 shows the efficiencies, which result from the speedups presented in Figure 4.3.2, for various processor numbers. The efficiency is shown as a function of the average running time per problem of the test set. We investigate the search depths from 3 to 8, for the 256 processor version additionally 9*-ply search, as described in Section 2.5.1. With this we get some data for the 256 processor version lying above the interval from 180 to 300 seconds, which we marked by the vertical lines. We note the following

Observation 4.3 *The efficiency is strictly increasing even for running times larger than the time allowed in a tournament.*

Therefore, especially for the networks with a large number of processors, a much better efficiency could be expected, if e.g. the threefold running times were allowed. On the other side the same increase of the efficiency can be expected, if the speed of the processors is increased by a factor of three. If this increase of the processing power is the same for the computing power and the communication power, then the expected efficiency of the new processors can be calculated by shifting the vertical lines, indicating the tournament speed, to the right, since the new processors evaluate a position in e.g. 200 seconds, for which the old processors need 600 seconds.

From Observation 4.3 we obtain the interesting

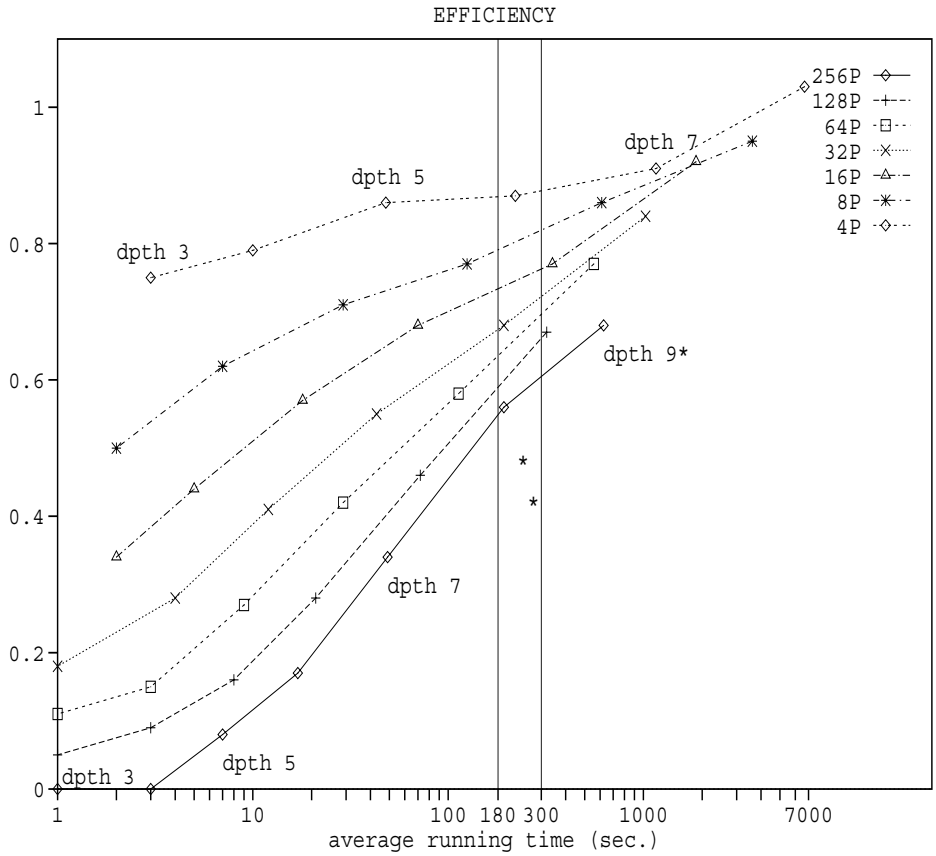


Figure 4.11: Efficiency as a function of the running time

Corollary 4.7 *A system of* 256 *processor, each three times as fast as the T800 processor is, if an average running time of* 200 *seconds is allowed per position, more than three times faster than a system consisting of* 256 *T800 processors.*

The reason for this increase of the efficiency is the better work load, the new system would be able to achieve in the same average running times. This increase of the work load is not compensated by a larger search overhead or a larger performance loss.

Example

Corollary 4.7 enables the forecast of an implementation of our chess program on the successor of the T800 processor, the T9000 processor. We may assume a three times greater computation - and communication power for the T900 compared to the T800. *ZUGZWANG-9000*, thus, would be able to do a 9*-ply search, which is done by the actual implementation in an average of 626 seconds, in tournament speed. The 9*-ply search is done by the actual implementation with an efficiency of 0.68, which results in a speedup of 174.56. This speedup is the result of an average work load of 83.46%, an average search overhead of 8.72% and a performance of 88.84%. The search overhead is smaller than for the 8-ply search, because during the 9*-ply search the

	processors						
	4	8	16	32	64	128	256
EFF* (250 sec.)	0.87	0.79	0.74	0.69	0.64	0.61	0.58

Table 4.4: Efficiency under tournament conditions

transposition table of the multiprocessor system is even more superior to the transposition table of the sequential version. The odd search depth of 9 for 19 of the 24 positions causes a small increase of the performance. The calculated speedup is

$$SPE^* = 256 \cdot \frac{88.84 \cdot 83.46}{100 \cdot 108.72} = 174.59.$$

Since we assumed that the T900 is three times as fast as the T800, we get a "speedup" of

$$3 \cdot 174.59 = 523.77$$

compared to a single T800 processor. Thus, under tournament speed, the 256 T9000 processors would be 3.68mal faster than the 256 processor system consisting of T800 processors, although every T9000 processor was assumed to be three times as fast as the T800.

If we now assume, that the system of T9000 processors would be connected via routing hardware, then we may assume a performance of 100% and get a speedup of

$$SPE^* = 256 \cdot \frac{100 \cdot 83.46}{100 \cdot 108.72} = 196.52$$

compared to a sequential T9000 processor. So, a system of 256 T9000 Transputers with routing hardware would be

$$3 \cdot 196.52 = 589.56$$

times faster than a single T800 Transputer. The T9000 system under tournament conditions, would thus be 4.14 times faster than the T800 system.

Scalability

By the data of Figure 4.11 the scalability of our distributed algorithm becomes apparent. If we approximate the efficiency between the points measured by a line, then we get approximations of the efficiency at the average running time of 250 seconds, which are presented in Table 4.4.

The efficiency decreases when increasing the number of processor from 4 to 8 by 0.08. Doubling the number of processors from 8 to 16, 16 to 32 and 32 to 64 causes the efficiency to decrease by 0.05 only. Increasing the number of processors from 64 to 128 and 128 to 256 resp. causes the efficiency to decrease only by 0.03. This shows that our algorithm is able to efficiently use massively parallel systems even under the timing constraints usual in practice.

Corollary 4.8 *The distributed game tree search algorithm proposed is well scalable.*

No	k_1	t_1	k_{256}	t_{256}	LOAD	SOVD	SPE
B01	661	2	950	1	0.00	43.72	2.00
B02	3498480	5972	4366554	58	67.24	24.81	102.97
B03	5882296	11255	7148739	96	63.54	21.53	117.24
B04	61791734	120014	40390075	428	77.10	-34.64	280.41
B05	32303334	60498	43759864	485	74.64	35.47	124.74
B06	490573	801	1098715	24	41.67	123.97	33.38
B07	11114505	22613	17410531	208	73.08	56.65	108.72
B08	258323	412	419999	18	16.67	62.59	22.89
B09	20684171	38170	40103755	404	76.98	93.89	94.48
B10	12461216	24214	18458295	206	78.64	48.13	117.54
B11	6702872	13516	13923029	172	75.00	107.72	78.58
B12	5940915	11202	8405364	112	67.86	41.48	100.02
B13	8891954	18171	14862244	214	70.09	67.14	84.91
B14	33382037	72628	26388785	318	74.53	-20.95	228.39
B15	25835293	43595	21544484	198	78.28	-16.61	220.18
B16	8535001	16100	8458131	118	61.86	-0.90	136.44
B17	8865335	16551	25354928	270	80.37	186.00	61.30
B18	21710022	41682	25126519	280	76.43	15.74	148.86
B19	5193365	9496	7592005	103	62.14	46.19	92.19
B20	6744642	13273	10478861	133	75.19	55.37	99.80
B21	21963828	36551	23064988	222	80.63	5.01	164.64
B22	12275596	23428	12646076	147	72.79	3.02	159.37
B23	12352850	24523	15980182	193	75.65	29.36	127.06
B24	18694564	35562	18383955	231	65.80	-1.66	153.95
Σ	345573567	660229	405367028	4639			
Φ	14398898	27510	16890293	193	74.13	17.30	142.32

Table 4.5: 8-ply search with 256 processors

4.3.7 8-ply Search with the DB(8): Single Positions

Table 4.5 shows the data measures during an 8-ply search of the standard single processor version, as well as with 256 processors. The data for the 256 processor version stems from that run of Section 4.2, which was closest to the average speedup calculated in Section 4.2.

Of special interwest are the five positions $B04$, $B14$, $B15$, $B16$ and $B24$ having negative search overhead. The largest part of the negative search overhead in the brute force tree can be explained by the overloading of the transposition table of the sequential algorithm. The experiments described in Section 4.5 make clear the behaviour of our algorithm without this effect. The reason for negative search overhead in the quiescence search was presented in Section 4.3.3.

Furthermore, we note that not only the average running time of 193 seconds but also the maximal running time of 485 seconds (position $B05$) is not unusual to occur in a tournament game. During a game sometimes even longer computation times are used for a single move. The overall behaviour of the parallel algorithm is much more "uniform" than that the sequential one: If we delete the mating problem $B01$ as well as the endgame positions $B06$ and $B08$, which in a tournament game would have been searched to a larger search depth, then the maximal running

time of the sequential algorithm is 120014 seconds and the minimal one is 5972 seconds. This corresponds to a ratio of

$$\frac{120014}{5972} = 20.10$$

in contrary to a ratio of

$$\frac{485}{58} = 8.36$$

for the parallel algorithm. In other words, the parallel algorithm much more than the sequential one is able to search the positions arising in a game to some uniform search depth, even if some of the positions are very complex. One reason for this behaviour is that the work load becomes better for longer running times, and thus large trees are searched more efficiently than smaller ones.

Corollary 4.9 *If the search depth is fixed, then the parallel algorithm decreases the difference between the running times of the more complex and the less complex positions by evaluating the former ones more efficiently and the latter ones less efficiently.*

The most negative example of this test set is position $B17$ having a bad speedup of 61.30, although the work load is 80.37%. The reason for this is the enormous search overhead of 186.00%. This search overhead is caused by a very unfortunate 8-ply search: the best move of the 7-ply search is improved by three other moves during the 8-ply search. Since these improvements are computed in parallel, all the sons of the root are evaluated with bad search windows. One after the other the improvements are notified to the network master, which transmits the new bounds to the slaves in level one. Since all the successors of the root are evaluated with a zero window search, the first window messages causes researches with bad bounds. However, the game tree of position $B17$ is ordered very poorly not only in the upper level. Observation 2.4 on page 40 shows that position $B17$, when evaluated by the sequential algorithm, has the largest distance from the minimal game tree. Therefore, the game tree to be searched is sorted very badly. Consequently, the search overhead of 186.00% is large.

4.4 Measuring of the Performance Loss

In Observation 3.1, we noted that routing messages will cause performance loss. In this section we will study this performance loss more extensively.

We want to answer the question, which kind of messages causes which fraction of the performance loss. In order to do so we distinguish between three kinds of messages:

1. Messages, which are necessary for the implementation of the distributed transposition table.

2. Subproblems, which are sent for establishing a master-slave relationship.

3. All the other messages. This includes messages, which are used to implement the distributed game tree search algorithm, as requests for work, returns, window- and cutoff messages, etc.

Version	Sequential	Standard	TMD	SPD
PERF (%)	100.00	90.22	82.95	89.45

Version	Standard	TMD	SPD	Rest
PERF loss (%)	0.00	7.27	0.77	1.74

Table 4.6: Performance after doubling messages

In order to study the effects of the single classes of messages on the performance loss we subsequently doubled the messages of one kind, i.e. a subproblem is sent twice instead of once to the same processor, which then receives the same subproblem twice. One of these copies is deleted at once. For these tests we used the 128 processor network $DB(7)$. The results were obtained from an 8-ply search on the 24 positions of the Bratko Kopec set of test positions. Besides the doubling of messages, the standard version described in Section 4.3 was used.

Table 4.6 (upper table) shows the average performance of a processor measured for several version of the128 processor algorithm. In the column "Sequential" the performance of the single processor version is given. Column "Standard" contains the percentage for the standard version of the 128 processor system, in which no messages are doubled. We observe a performance loss of 9.78% compared to the single processor version. By doubling the transposition table messages the average performance of the processors drops to 82.95% (column "DTM"). Doubling the subproblems, however, results in a much better performance of 89.45% (column "DPM").

Table 4.6 (lower table) shows the performance loss compared to the 128 processor standard version, which results from doubling some messages. Doubling the transposition table messages results in a performance loss of 7.27%, doubling the subproblems results in the much smaller performance loss of only 0.77%. The performance loss of all the other messages is calculated from the performance loss of the standard version with respect to the sequential version and the above two numbers.

Corollary 4.10 *The transposition table messages in the $DB(7)$ are responsible for $\approx \frac{3}{4}$ of the performance loss. Roughly $\frac{1}{3}$ of the remaining performance loss is caused by sending subproblems.*

The above numbers are astonishing at a first glance. Transposition table messages as well as messages for the control of the distributed game tree search have a size of 20 Byte roughly and thus are quite small compared to the subproblems, which have a size of $\approx$ 1000 Byte. However, during the evaluation of the 24 positions the standard version sent 5.7 million subproblems only, compared to 109 transposition table messages. Table 4.7 show the number of different messages as well as the number of MByte to transmit. The messages for the control of the game tree search are divided into requests, cancellations, window messages, cutoff messages and returns. The remaining performance loss of 1.74% mainly is caused by requests, cancellations and returns, because window- and cutoff messages are only a small fraction of these kind of messages.

Although sending all the subproblems requires the transmission of 5720 MByte and the realization of the distributed transposition table requires the transmission 2268 MByte only, the transposition table messages contribute much more to the performance loss than the set of

	Subproblems	Transposition	Tree search control				
			B	A	F	C	R
Number $\times 10^6$	5.7	109.0	11.4	3.8	0.01	0.6	5.4
MByte (roughly)	5720	2268.8	182.1	61.5	0.28	17.3	215.6

Table 4.7: Amount of communication

subproblems sent. There are two reasons for this:

1. The transposition table messages are sent most frequently in those phases of the computation, in which the processor work load is very good, because only processors working on a subproblem access the distributed transposition table. During these phases, however, the probability that a processor, which has to route a message, must interrupt its tree searching process to route the message, is very high. The CPU-time, necessary to route the messages can not be used by the tree searching process. The performance loss is large.

   Subproblems most frequently are sent, when the processor work load is bad. Processors, which have to route a subproblem are idle by themselves with high probability. Routing a subproblem over an idle processor, however, does not cause a loss of performance.

2. The second reason comes from the hardware and the system software of the Transputer. The transmission of large packages is much cheaper per Byte sent than the transmission of small packages. Therefore, the time necessary for a Transputer to sent a special amount of data is influenced by the number of packages as well as by the size of the data.

4.5 Costs and Gains of a Distributed Transposition Table

In Section 3.6 we described our implementation of our distributed transposition table. This implementation allows the use of a transposition table, which grows with the number of processors. Collisions in this hash table become less and less probable, the large the table is. Thus, we obtain the following advantages:

1. Better detection of transpositions.

2. Better move ordering.

In Section 4.5.1, we will investigate which effects are caused by a decrease of the size of the transposition table for the speedup.

The distributed hash table, however, can not be used for nothing. Accesses to the hash table as well as answers from the hash table modules have to be routed along possibly long paths in the processor network. This delays the accesses and causes the following disadvantages:

1. CPU-time is used for routing messages.

2. The generation of the game tree nodes is delayed, because processors have to wait for answers from the transposition table.

The work done by the processors for routing messages as well as the waiting times can be measured as performance loss of the distributed system. These losses of performance will be quantified in Section 4.5.2.

4.5.1 Gains of a Distributed Transposition Table

In order to investigate, how the multiprocessor system benefits from the use of a large transposition table, we looked at the behaviour of our distributed algorithm using a smaller table with the same number of processors. We compare several versions of the 256 processor version, which differ by the size of the transposition table located at each processor. Using local transposition table modules with 625, 1250, ...40000, and 160000 entries per processor, we get 256 processor versions having access to a total of $\frac{1}{256}$, ..., $\frac{1}{4}$, and $\frac{1}{1}$ of the maximal transposition table size. The basis of all the data presented in this section are 8/ply searches of the 24 Bratko Kopec positions. The average running time of 233 seconds ($\frac{1}{256}$) to 193 seconds ($\frac{1}{1}$) are lying in the range of what we call tournament speed. All the data presented in this section has been measured separately. Thus, the measurements for the version of maximal transposition table size differs slightly from the data presented in the Sections 4.2 and 4.3. Tables 4.8 and 4.9, as well as Figures 4.12 and 4.13, show the data obtained for the 256 processor version having access to a transposition table of $\frac{1}{1}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{64}$, $\frac{1}{128}$ and $\frac{1}{256}$ of the size of the standard version table. Since the data for $\frac{1}{4}$ and $\frac{1}{1}$ are almost equal, we omitted a test with $\frac{1}{2}$.

Table 4.8 contains for growing table sizes the percentage of successful requests to the transposition table compared the the number of all transposition table requests. Three kinds of successful accesses have been distinguished:

1. *Value:* The answer contains an exact value computed with a sufficiently large search depth.
2. *Bound:* The answer contains a bound computed with a sufficiently large search depth.
3. *Move:* Only the move from the transposition table could be used.

The fourth line contains the sums of the columns, i.e. the percentage of all in some way successful requests. The fraction of the successful requests, from which an exact value could be obtained, remains constant even for a table size of $\frac{1}{128}$ and more. The fractions of the other two kinds of a successful request, however, are increasing monotonically. These last two kinds of successful requests are mainly important for a good move ordering but are not of such an importance for avoiding the search of subtrees already searched. If one looks at the total of all the successful requests, then this percentage grows from 14.80% for the version with $\frac{1}{256}$ of the size of the standard table size to more than 30% when using the full transposition table, i.e. it increases by more than a factor of two.

In Table 4.9 we list the different transposition table sizes, as well as the data measured for the speedup, the search overhead, the work load, and the performance.

The performance and the processor work load remain unchanged as expected. The small decrease of the work load from 76.92% ($\frac{1}{256}$) to 74.39% ($\frac{1}{4}$) may be caused by the decrease of the total average running time from 233 seconds to 193 seconds. Table 4.10 shows the improvements of the speedups of the different versions with respect to the weakest version (117.91 = 100%). The speedup improves by a large amount by doubling the size of the transposition table in the range from $\frac{1}{256}$ to $\frac{1}{16}$ in a 256 processor system. Then the improvement becomes smaller and smaller. Further doubling of the transposition table size only causes very small speedup improvements.

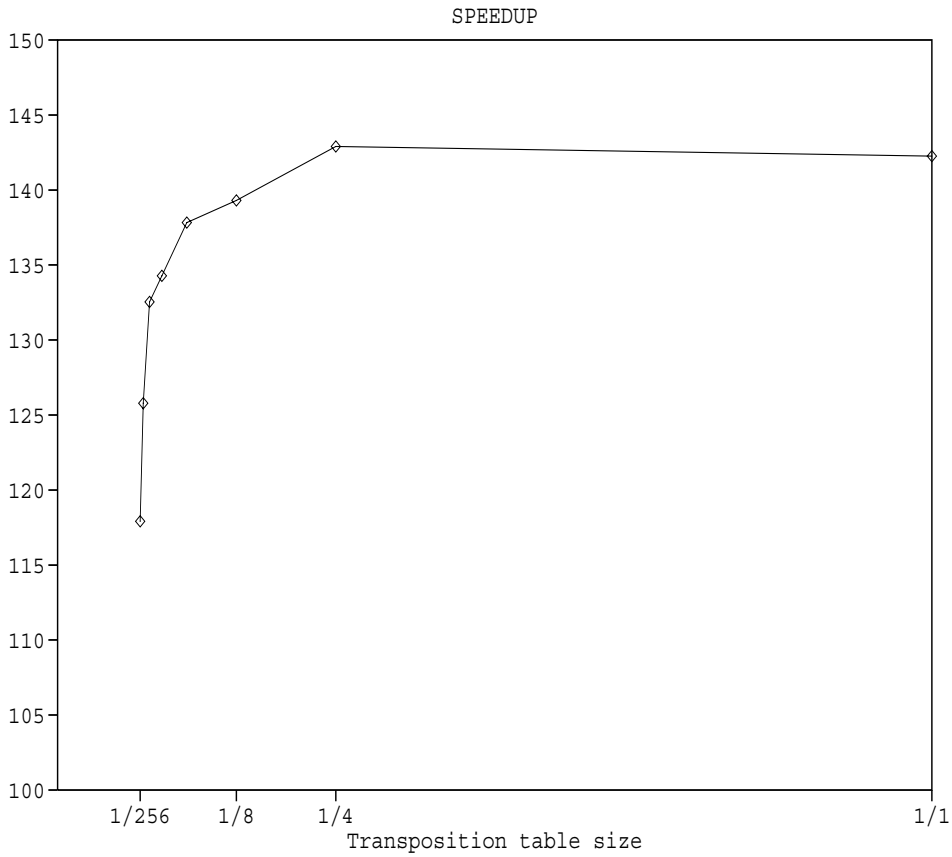


Figure 4.12: Speedups for varying sizes of the transposition table

Observation 4.4 *The versions with a table of size of* $\frac{1}{1}$ *or* $\frac{1}{4}$ *resp. of the maximal transposition table size are* $\approx 20\%$ *faster than the version with access to a table of size* $\frac{1}{256}$. *A speedup improvement of* 17% *roughly is even caused by a transposition table of size* $\frac{1}{16}$.

Similarly, the curves for the search overhead can be interpreted: The search overhead is greatly reduced by doubling the transposition table size in the range from $\frac{1}{256}$ to $\frac{1}{16}$. By doubling the size further only a small decrease of the search overhead is obtained. The reduction of the search overhead mainly is caused by a better move ordering, due to the larger transposition table, but not by a better detection of transpositions. The fraction of the positions, for which an exact value is found in the transposition table, does not increase significantly beyond a transposition table size of $\frac{1}{128}$. The fact that the version with $\frac{1}{1}$ of the full transposition table has a bit more search overhead than the version with $\frac{1}{4}$ of the full table, is due to the deviations observed in Section 4.2.

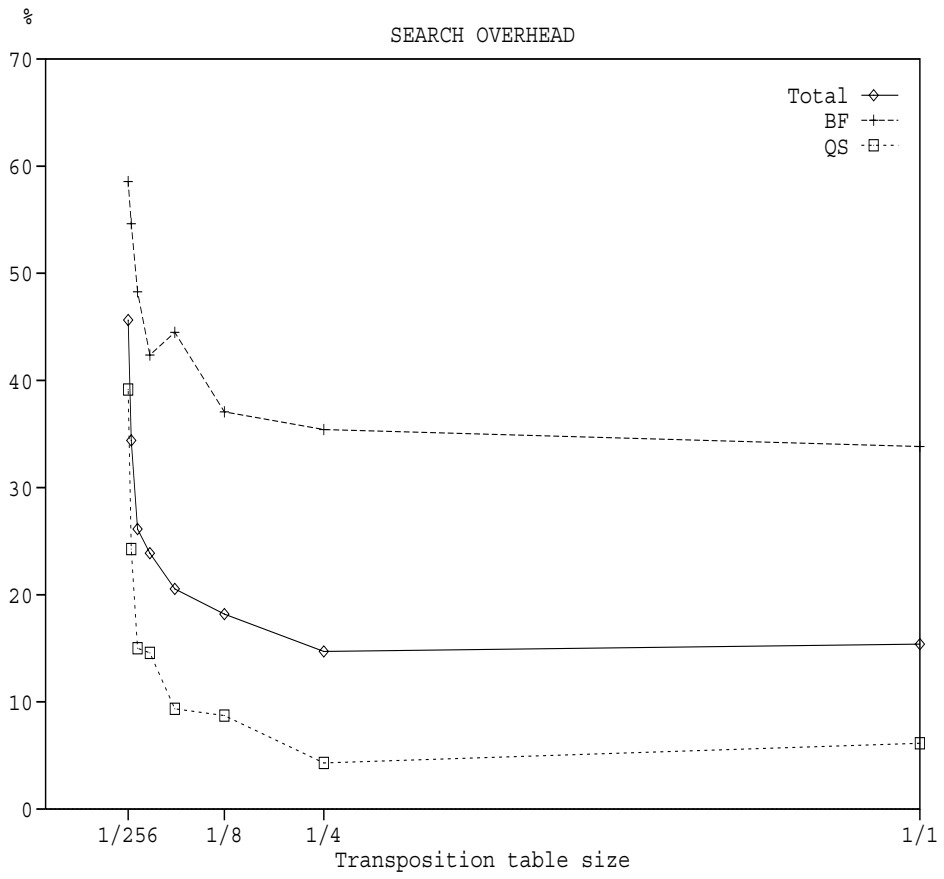


Figure 4.13: Search overheads for varying sizes of the transposition table

Selecting a Transposition Table Size

Observation 4.4 leads to the question of how large a transposition table should be chosen, to be of reasonable use compared to its costs (= size). This question will be answered by presenting the numbers of collisions, which occurred in the transposition table during the above described experiments.

In Table 4.11 the numbers of collisions are listed as a percentage compared to the number of all store operations for varying transposition table sizes.

From Observation 4.4 we get the following:

Corollary 4.11 *The version of our algorithm searching the game trees with an average percentage of collisions of* 23% *shows nearly the same behaviour as the version, which runs with the full transposition table achieving a percentage of collisions of* 2% *only.*

With this, the distributed transposition table of our 256 processor system seems to be oversized. In practice, that is during a whole game under tournament conditions, the large transposition table has some advantages. The table is not initialized between the computation of two successive

		$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{64}$	$\frac{1}{32}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{1}$
Value	(%)	0.0008	0.0011	0.0012	0.0013	0.0012	0.0012	0.0018	0.0012
Bound	(%)	10.57	14.47	17.54	19.68	20.72	21.60	22.31	22.62
Move	(%)	4.23	5.65	6.78	7.81	8.35	8.93	9.21	9.36
Total	(%)	14.80	20.12	24.32	27.49	29.07	30.53	31.52	31.98

Table 4.8: Successful accesses for varying transposition table sizes

	Elements	SPE	SOVHD	LOAD	PERF
$\frac{1}{1}$	40960000	142.26	15.39	74.57	86.97
$\frac{1}{4}$	10240000	142.90	14.70	74.39	87.15
$\frac{1}{8}$	5120000	139.31	18.19	74.68	87.15
$\frac{1}{16}$	2560000	137.83	20.54	75.53	87.34
$\frac{1}{32}$	1280000	134.28	23.87	75.31	87.15
$\frac{1}{64}$	640000	132.54	26.13	75.71	87.15
$\frac{1}{128}$	320000	125.79	34.40	76.00	87.73
$\frac{1}{256}$	160000	117.91	45.64	76.92	88.11

Table 4.9: Search behaviour for varying transposition table sizes

moves. The entries are marked with a time stamp, such that old entries may be detected and overwritten by newer ones, even if this new entries have been computed with smaller search depths. The larger the transposition table the more rarely the time stamp has to be changed, such that results computed earlier may overcome the computation of several moves in the hash table. This is very helpful, especially when searching game trees, in which many positions occur again, which have already been computed during earlier searches, e.g. in the endgame.

4.5.2 Costs of the Distributed Transposition Table

In this section we will investigate the costs of the distributed transposition table, which are caused by the waiting times for transposition table messages and the delay by routing these messages.

Waiting Times for Transposition Table Messages

In Section 3.6 we described, how the processors may use the time necessary to route the transposition table requests and the answers: During the time the messages are routed, the waiting

		$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{64}$	$\frac{1}{32}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{1}$
Speedup win	(%)	0.00	6.68	12.41	13.88	16.89	18.15	21.19	20.65

Table 4.10: Improvements of the speedup for varying transposition table sizes

	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{64}$	$\frac{1}{32}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{1}$
Collisions (%)	82.92	71.38	55.45	37.22	22.86	12.46	6.30	1.62

Table 4.11: Percentage of the number of collisions

Requests	Value	Bound	Move
100%	0.0012%	22.62%	9.36%

Table 4.12: Successful transposition table accesses

processor generates the position, for which it sent a TR-request, i.e. it updates the corresponding data structures. Generating the position makes sense when the result obtained from the transposition table does not suffice to stop the search below this position immediately.

Table 4.12 contains the percentage of the different kinds of successful requests to the transposition table for an 8-ply search of the Bratko Kopec positions with respect to the total number of requests. For this experiment we used 256 processors with the maximal transposition table size. The data shows, that only very rarely an exact value is delivered to some TR-request (0.0012%). In this case the search below the node requested is superfluous. In most of the successful cases a bound is delivered for the position requested. This bound may or may nor make necessary a search below the node requested. In general, however, the search must be continued. The search has to be continued anyway, if only the move of the transposition table entry can be used. Thus, generating the position requested is necessary for at least

$$100\% - 0.0012\% - 22.62\% \approx 77.38\%$$

of the requests. Statistical data for the number bounds causing a cutoff has not been measured.

The question is now, whether the time necessary to generate the position is long enough to completely cover the waiting time for the answer. One can expect the time necessary for the routing of messages to grow at least logarithmically in the number of processors. Therefore, the waiting time in large processor networks must be larger than the time to generate a position, which is a constant, independent of the number of processors. Table 4.13, however, shows, that the sum of all the waiting times not covered by the time to generate the positions during an 8-ply search of the Bratko Kopec positions with 256 processors is 1.13 seconds compared to the total time of 4639 seconds, and thus is negligibly small.

Observation 4.5 *Waiting times for transposition table messages in the $DB(8)$ are negligibly small.*

	Wait	Total
$DB(8)$ (Sec)	1.13	4639

Table 4.13: Sum of the waiting times for TR-requests

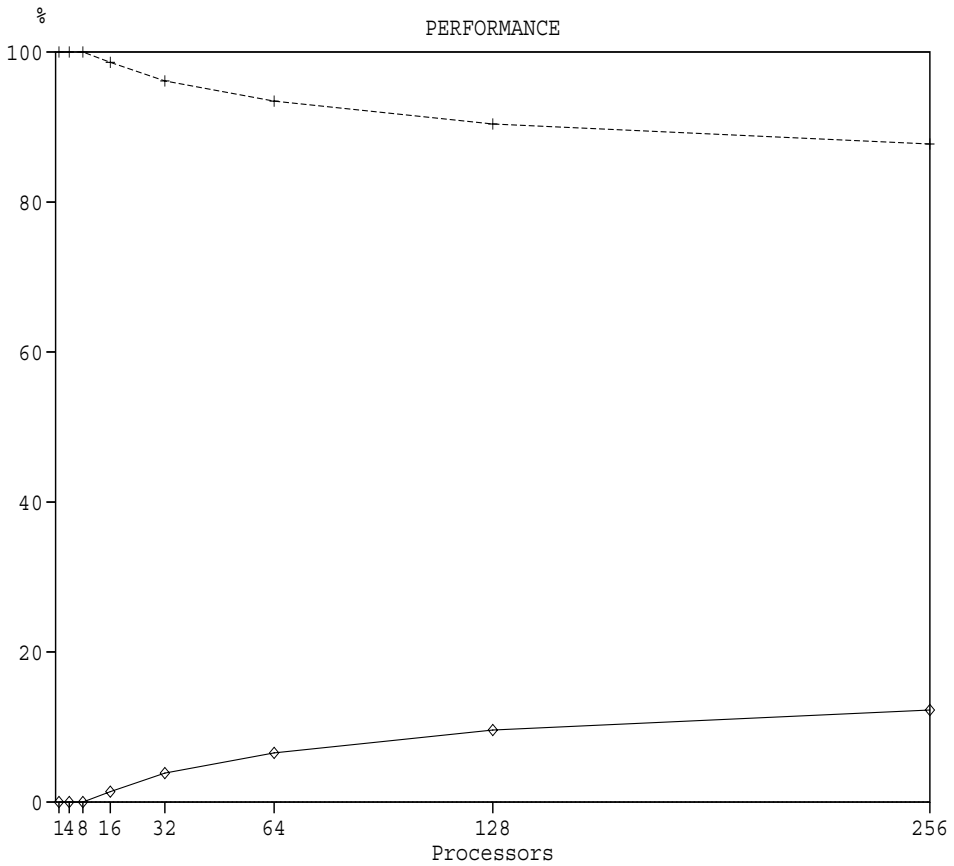


Figure 4.14: Performance loss as a function of the numbr of processors

Performance Loss by Routing tasks

Unfortunately, there is another reason which causes a loss of performance: Processors, which ly on the routing path from the origin of a messages to its destination are interrupted, because they have to route the message forward. The loss of performance caused by this effect is quite large.

Figure 4.14 and Table 4.14 show the performance losses, which are caused by routing all messages in processor networks of 4 to 256 processors. The basis of this data is an 8-ply search of the 24 Bratko Kopec positions . The topology used is the DeBruijn network.

The upper curve shows the performance obtained with the corresponding number of processors, the lower curve shows the performance loss compared to the sequential version. The performance losses for large systems is obvious. The 256 processor version, for example, spends roughly one eighth of the CPU power to route messages.

Figure 4.14 and Table 4.14 show the performance loss, which occurred by routing messages. In this context, however, we are interested in the performance loss caused by the transposition table messages only. Corollary 4.10 on page 108 states, that in the $DB(7)$ roughly $\frac{3}{4}$ of the total performance loss is caused by the transposition table messages. This fraction can be expected

	Processors							
	1	4	8	16	32	64	128	256
PERF (%)	100.00	100.00	100.00	98.62	96.13	93.46	90.40	87.73
Loss (%)	0.00	0.00	0.00	1.38	3.87	6.54	9.60	12.27
$\frac{3}{4}$ Loss (%)	0.00	0.00	0.00	1.04	2.90	4.91	7.20	9.20

Table 4.14: Performance loss by routing tasks

to remain unchanged for the other DeBruijn networks, since waiting times are of no influence for the performance loss. Thus, we obtain the performance loss as presented in the third line of Table 4.14 as the performance loss, which is caused by the transposition table messages only.

4.5.3 Conclusion

Our implementation of a distributed transposition table allows the multiprocessor system to access a data structure, which grows with the number of processors used. The analysis of costs and gains of this data structure leads to the following

Corollary 4.12 *The use of the distributed transposition table improves the speed of the* 256 *processor version by* 20% *approximately. In contrary to this the performance of the processors is decreased by* 9.2% *by the routing of the transposition table messages.*

The whole performance loss can be avoided, if for the routing of messages a special routing hardware can be used, as common in the modern parallel computers of today. With this hardware the little costs, which have to be paid for the use of the distributed transposition table, could be avoided.

The experiments, however, indicate that enlarging the transposition table will not have any influence on the behaviour of the distributed algorithm. A transposition table of the size as available in a 256 processor system, is sufficient to guarantee a good move ordering, and thus an efficient search, even if much more or much faster processors would have been used.

As a consequence of this the transposition table of a later version of *ZUGZWANG* than the one described in this work has been separated: the larger part of the table is used as a transposition table, the smaller one is used as a hash table for endgame databases. With this, access to endgame databases can be achieved during the game tree search without delay of the search in phases of the game, in which the entries of the endgame database are not yet relevant.

4.6 Decrease of the Search Overhead by the YBWC

In Section 3.7 we presented the YBWC as well as two modifications of the basic YBWC. We obtained the following versions:

YBWC The evaluation of a successor $v.j$ of a node v of the game tree may be started only if the leftmost successor $v.1$ of $v.j$ is completely evaluated.

YBWC-1-2 The YBWC is used only for type-1 - and type-2 nodes.

YBWC* Parallelism at a type-2 node may be used only, if every promising successor of this node is completely evaluated.

YBWC-1-2* The straightforward combination of the YBWC-1-2 and the YBWC*.

The Expected Types of Nodes Are Reasonable

The modifications of the YBWC are based on the idea, to guess with the help of the expected types of nodes for a game tree node, whether a cutoff will occur at this node. Thus, the success of both modifications depends besides others on how reasonable the expected type of a node is.

	BF		QS	
	Type 2	Type 3	Type 2	Type 3
$DB(8)$:YBWC-1-2*	0.49 %	0.63 %	1.16 %	1.14 %

Table 4.15: Percentage of the nodes with a wrong expected type

Table 4.15 shows, that during an 8/ply search with 256 processors a cutoff does not occur only at a small percentage of the nodes having expected type 2. Similarly the fraction of type-3 nodes, for which a cutoff is computed, is very small. Thus, an important condition for the use of the modified versions is fulfilled.

In this section, we will compare the versions YBWC-1-2*, YBWC-1-2 and YBWC with each other to investigate, how the first modification (* version) and the second modification (-1-2 version) of the YBWC behave. Here the YBWC-1-2* version is just the standard version discussed in Section 4.3. It will turn out, that the effects of the modifications are very small. At the end of this section, however, we will present a version, which we call YBWC-1. It corresponds to the PV-Split-Algorithm presented by Marsland and Campbell [45] with a dynamic load distribution, which is the basis for many game playing programs using parallel hardware.

YBWC-1-2* vs. YBWC-1-2

The comparison of the standard version (YBWC-1-2*) with the YBWC-1-2 version does not show large differences between the two versions. The effects of using the YBWC-1-2 version instead of the YBWC-1-2* version are so small that in order to compare both versions a measurement of the deviations must have been used.

In Table 4.16 we present the results measured in five tests for the YBWC-1-2 version, compared to the average results of the standard version calculated in Section 4.2. The five tests show speedups between 142.6 and 135.77, the standard version achieved speedups between 146.12 and 138.12, such that a comparison on the basis of a single test is impossible. A comparison of the average results of the YBWC-1-2 version and the standard version uncovers only minor differences, which are caused by a somewhat (7.04%) smaller search overhead. The reduction of the search overhead is almost compensated by a decrease of the average processor work load by 3% roughly. In total a small advantage remains for the standard version. The differences are easily explained: the YBWC-1-2 version at type-2 nodes may use parallelism earlier than the

Version	SPE	LOAD	SOVHD	PERF
YBWC-1-2: V1	139.94	77.17 %	23.54 %	87.54 %
YBWC-1-2: V2	135.77	77.34 %	27.91 %	87.73 %
YBWC-1-2: V3	140.68	76.94 %	22.57 %	87.54 %
YBWC-1-2: V4	142.60	76.61 %	20.58 %	87.73 %
YBWC-1-2: V5	139.44	76.92 %	23.68 %	87.54 %
YBWC-1-2: Φ	139.67	77.00 %	23.66 %	87.62 %
Standard	142.62	74.07 %	16.62 %	87.73 %
Difference	2.95	-2.93 %	-7.04 %	0.11 %

Table 4.16: Results of the YBWC-1-2 version

Version	SPE	LOAD	SOVHD	PERF
YBWC: V1	139.94	77.17 %	23.54 %	87.41 %
YBWC: V2	135.77	77.34 %	27.91 %	87.41 %
YBWC: V3	136.02	75.88 %	24.50 %	87.22 %
YBWC: V4	136.61	76.12 %	24.54 %	87.41 %
YBWC: V5	138.94	75.74 %	21.78 %	87.41 %
YBWC: Φ	137.46	76.45 %	24.65 %	87.37 %
YBWC-1-2: Φ	139.67	77.00 %	23.66 %	87.62 %
Difference	2.21	0.55 %	-0.99 %	0.25 %
Standard	142.62	74.07 %	16.62 %	87.73 %
Difference	5.16	-2.38 %	-8.03 %	0.36 %

Table 4.17: Results of the YBWC version

YBWC-1-2* version. Thus, a better work load is achieved. On the other side a larger search overhead is caused by the rash use of parallelism.

Observation 4.6 *The YBWC-1-2* version is only slightly superior to the YBWC-1-2 version.*

YBWC-1-2 vs. YBWC

Table 4.17 contains the data measured for the YBWC version in five runs, compared to the results of the YBWC-1-2 version and the standard version.

By the comparison of the YBWC version and the YBWC-1-2 version it becomes apparent that using parallelism more rashly at type-3 nodes as in the YBWC-1-2 has only a small effect on the efficiency of the program. The YBWC-1-2 was developed to obtain a better processor work load even for a minor increase of the search overhead. A small improvement of the work load by 0.55% can be detected. One reason for the minor decrease of the search overhead might be that in the YBWC-1-2 version parallelism may be used more rashly at the type-3 nodes, such that it is used more rarely to evaluate type-2 nodes, where the first successor failed to cut off the search. However, since the effect is so small, we are not able to present measurements indicating this conjecture.

Version	SPE	LOAD	SOVHD	PERF
YBWC-1	39.22	90.57 %	416.85 %	84.04 %
Standard	142.62	74.07 %	16.62 %	87.73 %

Table 4.18: Results of the YBWC-1 version

Observation 4.7 *The YBWC-1-2 version is only slightly superior to the YBWC version.*

Only minor effects can be observed also from the comparison of the YBWC version and the standard version. Here the search overhead is reduced by (8.03%) in the standard version, however this positive effect is almost compensated by the better work load (2.38%) of the YBWC version. These results correspond to what we expected: the YBWC-1-2* version, which uses parallelism more carefully at type-2 nodes, evaluates the game trees with smaller search overhead, because the change from the YBWC to the YBWC-1-2 does not have any influence on the search overhead. However, by the more careful use of the parallelism the work load is somewhat decreased.

As a conclusion we may write:

Corollary 4.13 *The careful modifications of the YBWC resulting in the presented variations only have minor influences on the efficiency of the distributed algorithm.*

The short example of the next section, however, shows that minor changes to the YBWC may destroy the efficiency of the distributed algorithm.

YBWC-1-2* vs YBWC-1

If we change the YBWC-1-2 allowing direct use of parallelism even at the type-2 nodes, then we get an implementation of the PV-Split-algorithm, which was first presented by Marsland and Campbell [45], using a dynamic processor tree of depth > 1. We call this modification the YBWC-1 version. Because of the rash use of parallelism at the type-2 nodes, this algorithm will not search a best ordered game tree without search overhead. In practical applications we therefore expect that the search overhead of this version is larger than the search overhead of the YBWC-1-2 version. On the other hand, the load distribution is very easy, although by the iterative deepening process during a d-ply search $O(d^2)$ synchronization nodes will occur.

Table 4.18 shows the results obtained from an 8-ply search compared to the standard version.

The results show the advantages of the YBWC-1-2* compared to the YBWC-1 impressingly. The average running time per problem increases from 193 seconds to 701 seconds by more than a factor of 3.6. The search overhead increases by 400%, because in all right subtrees parallelism is used without care. For instance, it is used even for the evaluation of the right successors of a type-1 node, although a cutoff is expected to occur at these nodes and, in fact, happens at most of them. For this reason, even Marsland and Campbell [45] suggested not to use any parallelism at type-2 nodes. With this, however, the load distribution will become difficult, since, if a cutoff does not occur, then all of the successors of such a node must be evaluated one after the other.

Corollary 4.14 *The YBWC as used in our distributed algorithm, is a major improvement of the concept used to avoid search overhead in the basic PV-Split-algorithm.*

4.7 Use of the Load Distribution Strategies

In this section we will investigate, how the methods for load distribution described in Section 3.8 work, which take care of the structure of the underlying processor network during the game tree search. All results presented in this section are obtained from 8-ply searches of the 24 Bratko Kopec positions using 256 processors. For the comparison with the standard version we will use the average speedup, work load, search overhead and performance from Section 4.2, as long as the whole set of positions is considered. Results for single positions were obtained from the run wit speedup 142.32 in Section 4.2, which was closest to the average speedup of 142.62.

In Section 3.8 we described the following methods:

N1: Processors send their requests for work to a randomly chosen neighbor.

N2: If a processor P_1, which is not able to transmit a subproblem, gets a request from some processor P_2, which has been transmitted less than c times, then it transmits this request to a processor chosen randomly from the set of its ≤ 3 remaining neighbors. If it gets a request, which has been transmitted c times, then it sends a cancellation message to the origin of the request.

N3: A return is interpreted as a request for work by the receiver of the return. A processor, which finishes the evaluation of its subproblem, starts its search for work with the return message already. Processors, which are lying between the origin and the destination of the return message, route the message to its destination without reacting upon the request contained implicitly.

The standard version discussed in Section 4.3 uses N1, N2 and N3, as described in Section 3.8.4 on page 83.

Corollary 3.2 states, that the use of N1 without the use of N2 and N3 or N2 with a large radius constant c does not make any sense. We will see through results presented in this section that for the same reason even the use of N1 and N2 with a large radius constant c but without the use of N3 will cause a bad behaviour. This will become clear by some a comparison of the standard version with the version without N3.

In order to separate the effects caused by the use N1, N2 and N3 we will compare the following versions: the standard version, the version without N1, the version without N1 and N2 and the version without N1 and N3.,

Standard vs. Standard without N3

In the version with N1 and N2 but without N3 a return messages is not interpreted as a request for work. A processor is able to ask only those processors for work, which lie in its surrounding defined by the radius constant von of N2. This radius constant $c = 8$ is large enough to enable any processor of the $DB(8)$ to ask any other processor for work.

Version	SPE	LOAD	SOVHD	PERF
Std \ N3	106.59	55.09 %	21.02 %	91.55 %
Std	142.62	74.07 %	16.62 %	87.73 %

Table 4.19: Results of the version with N1 but without N3

Independently from the radius constant the following effect occurs: If a processor starts its search for work at a time, at which the overall processor work load is bad, then the request for work is transmitted several times, i.e. possibly processors are asked for work, which have a large distance to the origin of the request in the network. In this case, however, the probability is high that a search for work is not successful and thus is stopped by a cancellation. If a processor starts its search for work at a time, at which the processor work load is good, then the probability is high, that its request is transmitted to some few processors only, until it eventually is answered by sending a subproblem, i.e. an idle processor with high probability gets its new subproblem from a processor, which is not too far away. A global load distribution is done very rarely.

Table 4.19 shows the average results of the standard version obtained in Section 4.2 compared to the results of the version without N3. The most obvious effect is the decrease of the average processor work load by 20% nearly. The small increase of the search overhead lies within the possible deviations observed for the standard version in Section 4.2. A small increase, however, may result from the different way to evaluate the game trees in parallel: During the evaluation of the game trees the standard version transmits 10.7 million subproblems with an average depth of 7.74. The version without N3 transmits nearly 37.6 million subproblems with an average depth of 8.89. Obviously, in the latter version there are a lot of processors, which only get deep (and thus small) subproblems, therefore have to search for work very frequently, and thus get very many subproblems. However, this large number of small subproblems does not suffice, to guarantee a good processor work load. This smaller work load of the version without N3 is a reason for the increase of the performance by 3.82%. Processors are idle for a longer time and thus are not disturbed by the necessary routing tasks. We state:

Observation 4.8 *In the comparison of the standard version and the version without N3 the version without N3 shows a* 20% *worse processor work load.*

Standard vs. Standard without N1

In this section, we will compare the standard version and the version without N3. Since the differences will turn out to be very small, we measured the deviations of the version without N1. In Table 4.20 we present the results of this version compared to the average results of the standard version.

The results are as expected. They show that in conjunction with the use of N2 and N3 the use of N1 does not have any negative influences on the efficiency of the parallel algorithm. In contrast: although the average processor work load of the standard version is 0.84% smaller, the average speedup increases by 4.08. This increase of the speedup is the result of a small increase of the performance by 0.62% and a small decrease of the search overhead by 3.98%. On the other side, the minimal increase of the work load of the version without N1 may be due to the small increase of the search overhead, and thus of the work done. The increase of the search overhead

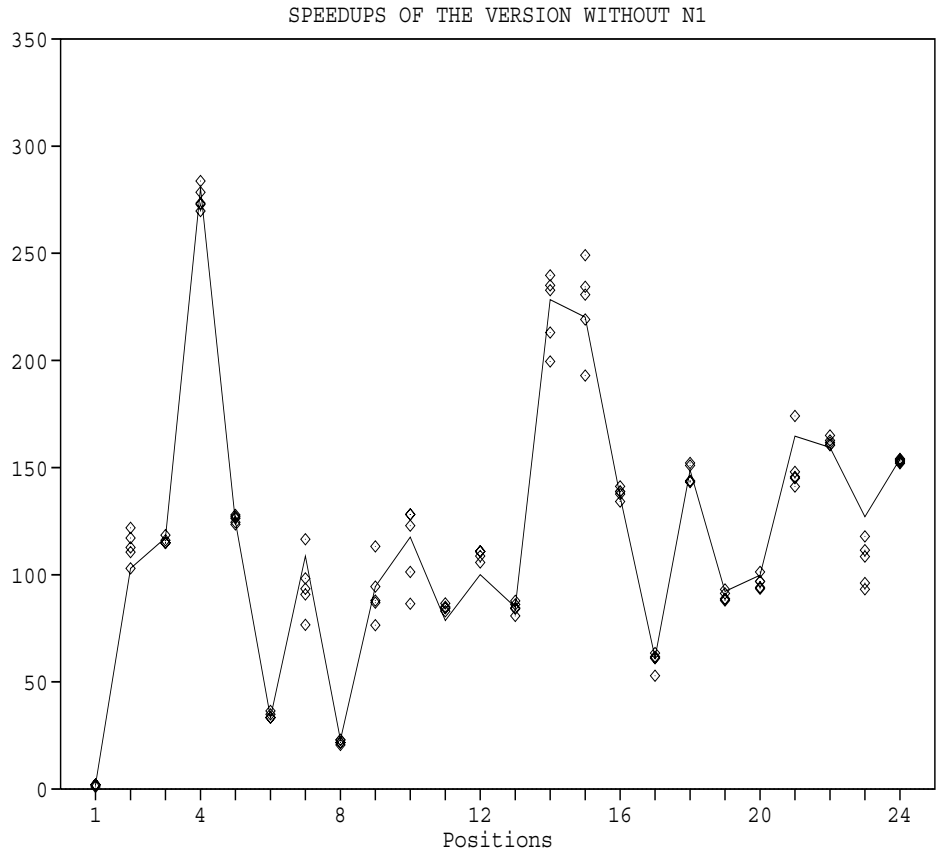


Figure 4.15: Results of the version without N1

of the version without N1 is very difficult to explain. First, the difference of the search overheads is very small. Second, if one compares the five tests of the version without N1 with the data obtained from the fifth run of the standard version of Section 4.2, then almost all problems are sometimes evaluated faster and sometimes evaluated slower than by the standard version.

Figure 4.15 shows the speedups measured for the single problems of the Bratko Kopec test (diamonds) compared to the fifth run of the standard version (polygon). It turns out that there are very few problems, which are always evaluated faster (11,12,22) or are always evaluated slower (8,23,24) by the version without N1. For instance, an effect which reduces the search overhead, as far as there is such an effect, can not be observed by a comparison of the average results for the 24 processors. The evaluations of single positions, however, are not well suited to find reasons for such an effect causing such small differences, because of the large deviations observed for the evaluation of single positions. Even the data measured for the number of messages transmitted does not give any hint for a completely different behaviour of the version without N1. Table 4.21 shows the number of different messages transmitted by the 256 processors during the 8-ply searches of the 24 test positions.

All together we are able to state the following:

Version	SPE	LOAD	SOVHD	PERF
Std \ N1: V1	135.49	75.11 %	23.18 %	86.77 %
Std \ N1: V2	140.44	74.64 %	18.77 %	87.34 %
Std \ N1: V3	137.58	75.24 %	22.45 %	87.54 %
Std \ N1: V4	139.00	74.74 %	19.76 %	86.97 &
Std \ N1: V5	140.18	74.80 %	18.86 %	86.97 %
Std \ N1: Φ	138.54	74.91 %	20.60 %	87.11 %
Std	142.62	74.07 %	16.62 %	87.73 %
Difference	4.08	-0.84 %	-3.98 %	0.62 %

Table 4.20: Results of the version without N1

		Subproblems	Transposition	Search control				
				Req	Can	Win	Cut	Ret
Std	$\times 10^6$	10.7	114.5	11.1	9.9	0.01	1.0	9.6
Std \ N1	$\times 10^6$	8.9	118.9	11.3	8.6	0.01	1.1	8.2

Table 4.21: Communication of the version with and without N1

Observation 4.9 *The use of N1 in conjunction with N2 and N3 only has minor influence on the behaviour of our distributed game tree search algorithm running on a* 256 *processor DeBruijn network.*

In Chapter 5, we will observe that the use of N1 makes sense, if processor networks with a large average distance between the processors are used.

Somewhat more significant effects can be seen, if one looks at the results of the version without N1 and N2 or the version without N1 and N3 respectively. The differences of the behaviour of these versions will be described in the next sections.

Standard Without N1 vs. Standard Without N1 and N2

In this section we will use the average results for the speedup, the work load, the search overhead and the performance of the previous section to compare the version without N1 with the version without N1 and N2.

Table 4.22 contains the average results of the version without N1 and the results obtained from one run of the version without N1 and N2.

These results show that the behaviour becomes a little bit worse by switching off the use of N2. The most obvious effect is the decrease of the average processor work load from 74.91% without N1 to 72.86% without N1 and N2.

This shows that the use of N2 enables the processors to faster establish a master-slave relationship. This is clearly to be expected, since a request for work is answered by a cancellation, only if $c + 1 = 9$ processors were not able to send a subproblem.

The number of requests and cancellations increases by switching off N2 by more than a factor

Version	SPE	LOAD	SOVHD	PERF
Std \ N1 and N2	130.71	72.86 %	23.52 %	86.58 %
Std \ N1: Φ	138.54	74.91 %	20.60 %	87.11 %
Difference	7.83	2.05 %	-2.92 %	0.53 %

Table 4.22: Results of the version with and without N2

		Subproblems	Transposition	Search control				
				Req	Can	Win	Cut	Ret
Std \ N1	$\times 10^6$	8.9	118.9	11.3	8.6	0.01	1.1	8.2
Std \ N1, N2	$\times 10^6$	8.4	122.8	93.4	77.7	0.01	1.0	7.7

Table 4.23: Communication of the versions with and without N2

of eight. Here one has to note that the requests are counted only at their origins but not at the intermediate processors, which transmit the request according to N2. In the version with N2 and a radius constant of $c = 8$ a request may travel over a maximum of $c+1 = 9$ processors. The data of Table 4.23 indicates that most of the requests indeed travel via eight or nine processors. The reason for this is that almost all searches for work are done in those phases of the computation, in which the work load everywhere in the network is bad. In these phases it is unlikely that a request comes to a processor, which is able to transmit a subproblem. In the version without N1 and N2 a processor has to send 9 requests, which then must be routed via the average processor distance of 4.97 in the $DB(8)$, while in the version with N2 only one request has to be routed via 4.97 processors on the average and then is transmitted to 8 other processors without routing. Furthermore, in the version without N1 and N2 in the worst case 9 cancellations must be sent for the 9 requests, while in the version with N2 only a single cancellation is necessary. The only small increase of the performance of the version with N2 indicates, that this reduction of the communication is achieved mainly in phases of the computation, in which the performance is not influenced very much by the communication. That is, in phases in which the processor work load is bad, because in large part of the processor network no subproblems are available for transmission.

The results for the version without N1 and N2 are from a single run. Measurings of deviations have not been made here, but in a second run the same effects could be observed besides the fact, that in the second run the average search overhead of 21.10% was slightly better than in the first run. Thus, the small increase of the search overhead may be caused by deviations.

Corollary 4.15 *The use of N2 speeds up the searches for work and reduces the routing necessary.*

This guarantees a smaller communication load even in those phases of the computation, in which the communication load is at its maximum. If one compares the version without N1 and N2 with the standard version, then we get

Version	SPE	LOAD	SOVHD	PERF
Std \ N1 and N3	130.25	78.77 %	33.85 %	86.39 %
Std \ N1: Φ	138.54	74.91 %	20.60 %	87.11 %
Difference	8.29	-3.86 %	-13.25 %	0.72 %

Table 4.24: Results of the version without N1 and N3

Observation 4.10 *The speedup of the standard version of* 142.62 *is better than the speedup of* 130.71 *achieved by the version without N1 and N2. This corresponds to a speedup of*

$$\frac{142.62 - 130.71}{130.71} \cdot 100\% = 9.1\%.$$

Standard Without N1 vs. Standard Without N1 and N3

This section describes a comparison of the version without N1 and the version without N1 and N3. It turns out that switching off N3 decreases the speedup by roughly the same amount as switching off N2. The reasons, however, are completely different.

Table 4.24 shows the results measured for the version without N1 and N3 compared to the version without N1. The speedup decreases by switching off N3 from 138.54 to 130.25 and even the performance decreases by 0.72%. Quite astonishing, however, is

Observation 4.11 *Without N3 the average search overhead increases from* 20.60% *to* 33.85, *i.e. by more than a half.*

The difference is much larger than the deviations observed for the search overhead of the version without N1. In the following sections we will present some reasons for this behaviour.

Better Use of the Killer Lists by N3:

The worse search overhead of the version without N1 and N3 has a simple reason: The local killer lists and history tables used to improve the move ordering in the game tree work more effectively in the version with N3. After a master-slave relationship has been ended, the receiver of the corresponding return message is interpreted as a request for work. If the former master has not yet finished the evaluation of its subproblem, then it is very likely, that it has another subproblem available for transmission. In this case it sends another subproblem to its former slave. This slave now evaluates a subproblem, which has in common with its former subproblem, that it is rooted in the same subtree as the subproblem the processor evaluated before. If the return is not interpreted as a request, then the former slave chooses a potential master randomly. The subproblem, which is sent to it then may be rooted in any part of the game tree actually under consideration. If a request is interpreted as a request, then it is likely, that two subproblems evaluated by a processor subsequently are very close to each other in the game tree.

Corollary 4.16 *The use of N3 increases the probability that the contents of the killer lists and the history table better fit for the subproblems evaluated.*

Version	Px	K1	K2	Nx	Rest
Std \ N1 and N3	10.84 %	7.95 %	25.72 %	18.18 %	37.32 %
Std \ N1	11.03 %	8.81 %	26.93 %	18.88 %	34.34 %
Difference	0.19 %	0.86 %	1.21 %	0.70 %	-2.98 %

Table 4.25: Cutoffs caused by right successors

Version	K1	K2	Delete
Std \ N1 and N3	85.30 %	14.70 %	13.72 %
Std \ N1	86.39 %	13.61 %	12.52 %

Table 4.26: Killer updates

Corollary 4.16 is indicated by the data of Table 4.25. It shows that the killer moves more frequently produce a cutoff in the version with N3 than in the version without N3. Table 4.25 contains the percentage of the cutoffs, which are caused by right successors for several classes of right successors. The percentages are calculated with respect to the number of all the cutoffs by right successors. Column "Px" contains the percentage for positive captures, the columns "K1" and "K2" the percentage for the first and the second killer move respectively. "Nx" represents negative captures, column "Rest" contains the percentage for all the other moves. It turns out that the killer heuristic works more effective in the version with N3, showing a rate of success of

$$8.81\% + 26.93\% = 35.74\%$$

compared to the version without N3 (33.67%).

The numbers of Table 4.26 also indicate an improvement of the use of the killer heuristic: In the version with N3 in 86.39% of the killer updates the bonus of the first killer move is increased, in 13.61% of the killer updates the bonus of the second killer move is increased or the second killer move is overwritten by a new one (12.52%). In the version without N3 in 14.70% of the killer updates the move to be updated is nt found at the first position of the killer list, the second killer move is overwritten in 13.72% of the killer updates.+ These numbers also indicate that in the version with N3 the killer moves have to be changed less frequently, because often subsequent subproblems to be evaluated by a processor require the same killer moves.

Better Move Ordering by N3:

Since the history heuristic and the killer heuristic are based on similar assumptions, we believe that the history heuristic too works more effective in the version with N3. The better killer moves and the better information from the history heuristic cause the search tree to be better ordered.

Table 4.27 shows the percentage of the nodes, for which the expected type turned out to be wrong during the game tree search. Looking at these results we find that in the version without N1 and N3 at 0.64% of all the type-2 nodes in the brute force tree (BF) no cutoff occurs and at 0.81% of all the type-3 nodes of the brute force tree a cutoff was computed. The results for the version with N3 are better: The percentages of 0.50% of the type-2 nodes, at which no cutoff occurred, as well as of 0.63% of the type-3 nodes, at which a cutoff is computed are both

Version	BF Typ 2	BF Typ 3	QS Typ 2	QS Typ 3
Std \ N1 and N3	0.64 %	0.81 %	1.25 %	1.25 %
Std \ N1	0.50 %	0.63 %	1.16 %	1.14 %
Std	0.49 %	0.63 %	1.16 %	1.14 %

Table 4.27: Percentage of the nodes with a wrong expected type

smaller than for the version without N3. Similar results have been obtained for the quiescence search (QS). Furthermore, Table 4.27 shows that switching off N1 does not have any influence on the fraction of the nodes having a wrong expected type. The results obtained for the standard version are almost equal to the results of the version without N1.

Better Use of Parallelism by N3:

The improvement of the killer moves by the use of N3 has an additional advantage. The *-modification of the YBWC described in Section 3.7 , which is used throughout this section as the YBWC-1-2*, uses parallelism depending on the expected type of the nodes. Thus, at a type-2 node a parallel evaluation of its successors is allowed, only if all the promising successors of this node have been completely evaluated. A part of the set of promising successors, however, are the successors reached via the killer moves. The improvement of the quality of the killer moves by the use of N3 increases the probability that a killer move produces a cutoff at a type-2 node, which otherwise would have been computed by evaluating a nonpromising move. In these situations by delaying the use of parallelism additional search overhead is avoided. A version with worse killer moves, thus, uses parallelism more often at nodes, at which a cutoff will be computed. The search overhead of such a version will be larger.

As a conclusion we may state:

Corollary 4.17 *The use of N3 improves the quality of the move ordering by improving the effects of the history heuristic and the killer heuristic. In addition, by the improved quality of the killer moves a better use of parallelism is possible. The improvement of the move ordering together with the better use of parallelism causes an overall better efficiency of the version with N3.*

Observation 4.12 *The speedup of the standard version of* 142.62 *is*

$$\frac{142.62 - 130.25}{130.25} \cdot 100\% = 9.5\%$$

better than the speedup of the version without N1 and N3 of 130.25.

4.8 Experiments with Varying Topologies

In this section we will investigate, which effects the processor network used has for the efficiency of our distributed $\alpha\beta$-algorithm. In order to do this we present results, which are obtained from tests with the three topologies DeBruijn Network, torus and grid, each with 256 processors.

Figure 4.16 shows the speedups achieved with the three networks for the 24 positions of the Bratko Kopec set. All the results are measured during an 8-ply search, the results of the DeBruijn network $DB(8)$ are the results obtained from the fifth run to measure the deviations (see Section 4.2).

The DeBruijn network shows a better behaviour than the torus or the grid on every single position. Only for position $B15$ the torus achieved the same efficiency as the DeBruijn network. The processor work load in Figure 4.17 looks similar: The DeBruijn network is for almost all test positions at least as good as the torus or the grid networks. The only exception here is the second position. The search overhead in Figure 4.18, however, looks somewhat nonuniform, which may be caused by the deviations observed especially for the search overhead.

Table 4.28 shows the results obtained on the average for the 24 positions evaluated using the above mentioned networks. From these results, it can be seen what can be expected from the results of the single positions presented in the Figures 4.16, 4.17 and 4.18. The DeBruijn network achieves a speedup of 142.32, which is much better than the speedup of the torus (121.79) and the speedup of the grid (107.93). The average work load in the DeBruijn network is slightly better than in the grid and the torus, the search overheads of the grid and the torus are slightly worse than the search overhead of the DeBruijn network and thus are in the interval of likely results observed in Section 4.2.

Table 4.28, however, shows the main reason for the bad efficiency of the torus and the grid networks: the average performance of of torus is 78.94% only, and the average performance of the grid of 72.44% is even less. In comparison to these the performance of the DeBruijn network is 87.91%. Table 4.29 shows that waiting times for transposition table messages are irrelevant compared to the total running times for each of the networks used. Thus, we obtain

Corollary 4.18 *The throughput of any of the three networks is sufficient.*

An insufficient throughput would have caused larger waiting times.

With this it is clear that even in the torus and the grid network the routing of messages is the only reason for the large performance losses. Corollary 4.5 stated that $3 \cdot \Delta_\Phi - 3.45$ is a very good approximation of the performance loss for all of the three networks.

Thus, we may state:

Observation 4.13 *The choice of the communication network heavily influences the efficiency of our distributed game tree search algorithm. The use of the DeBruijn network results in an increase of the speedup by*

$$\frac{142.32 - 107.93}{107.93} = 31.9\%$$

compared to the grid and by

$$\frac{142.32 - 121.79}{121.79} = 16.9\%.$$

compared to the torus.

The influence of the communication network, e.g. is much larger than the effect of multiplying the size of the transposition table of the sequential algorithm in the distributed system. On the other hand, we state that the throughput of the torus as well as of the grid are sufficient to

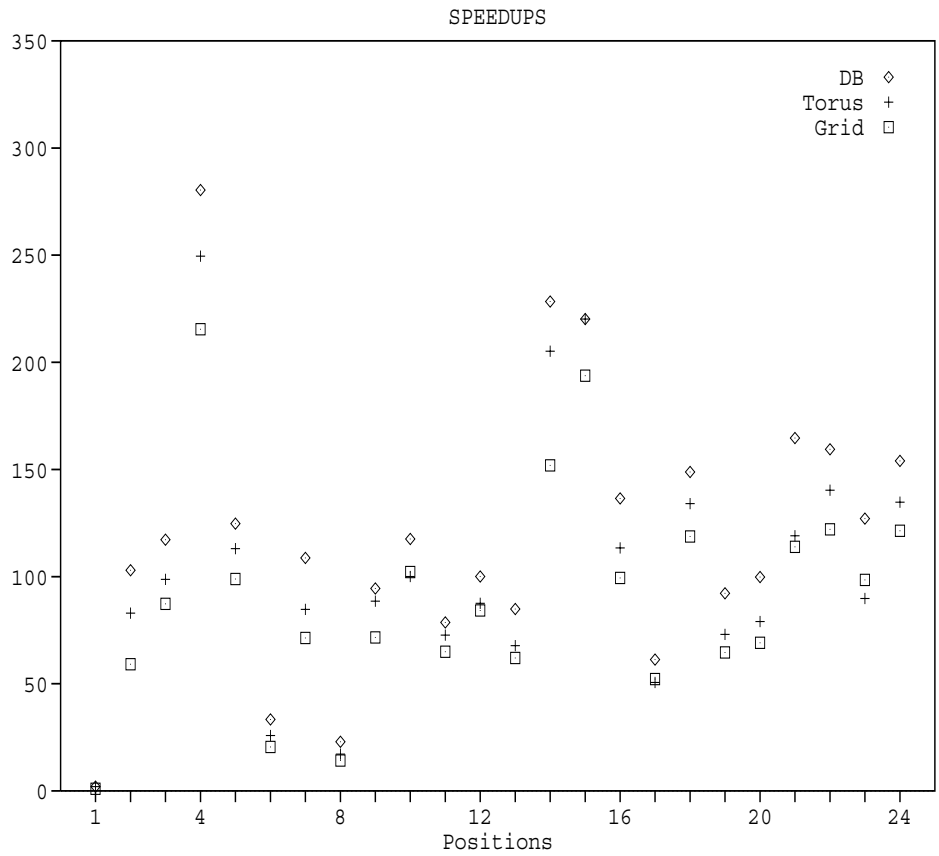


Figure 4.16: Speedups of the $DB(8)$, $T(16,16)$, and the $G(16,16)$

Network	SPE	LOAD	SOVHD	PERF	PERF loss
DeBruijn	142.32	74.13 %	17.30 %	87.91 %	12.09 %
Torus	121.79	72.09 %	19.57 %	78.94 %	21.06 %
Grid	107.93	70.59 %	21.39 %	72.44 %	27.54 %

Table 4.28: Average results of the DeBruijn, torus and grid

Network	$DB(8)$	$T(16,16)$	$G(16,16)$
Total time (sec)	4639	5421	6117
Waiting time (sec)	1.17	2.63	10.67

Table 4.29: Sum of the waiting times for transposition table messages

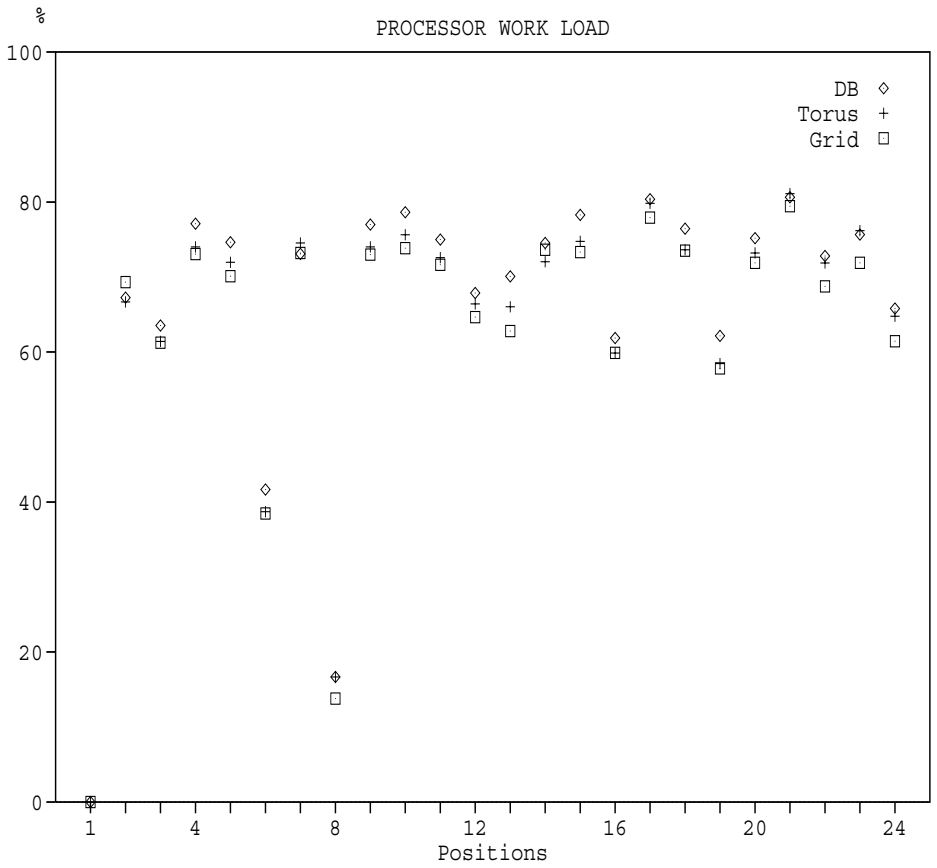


Figure 4.17: Work load of the $DB(8)$, $T(16,16)$, and $G(16,16)$

guarantee an efficient implementation of the distributed transposition table. The only serious disadvantage of the torus and the grid compared to the DeBruijn network is the larger amount CPU power, which has to be used for routing messages and thus is lost for the search process. In modern parallel computers more and more special routing hardware is used, which routes messages without the CPU. In parallel computers, which make use of these kind of hardware, networks like the torus or the grid were almost as useful for our purposes as the DeBruijn network is.

4.9 Influence of the Dynamic Load Distribution

The mechanisms to choose a potential master allow an establishment of a master-slave relationship between any two processors. This freedom result in a very good load distribution. Large waiting times for messages, which could be hidden by a time shared evaluation of several subproblems on a single processor, do not occur in our algorithm. Therefore, we omitted the implementation of methods to introduce a load balancing in the sense, that processors are allowed to get several subproblems in order not to become idle. Although such load balancing strategies

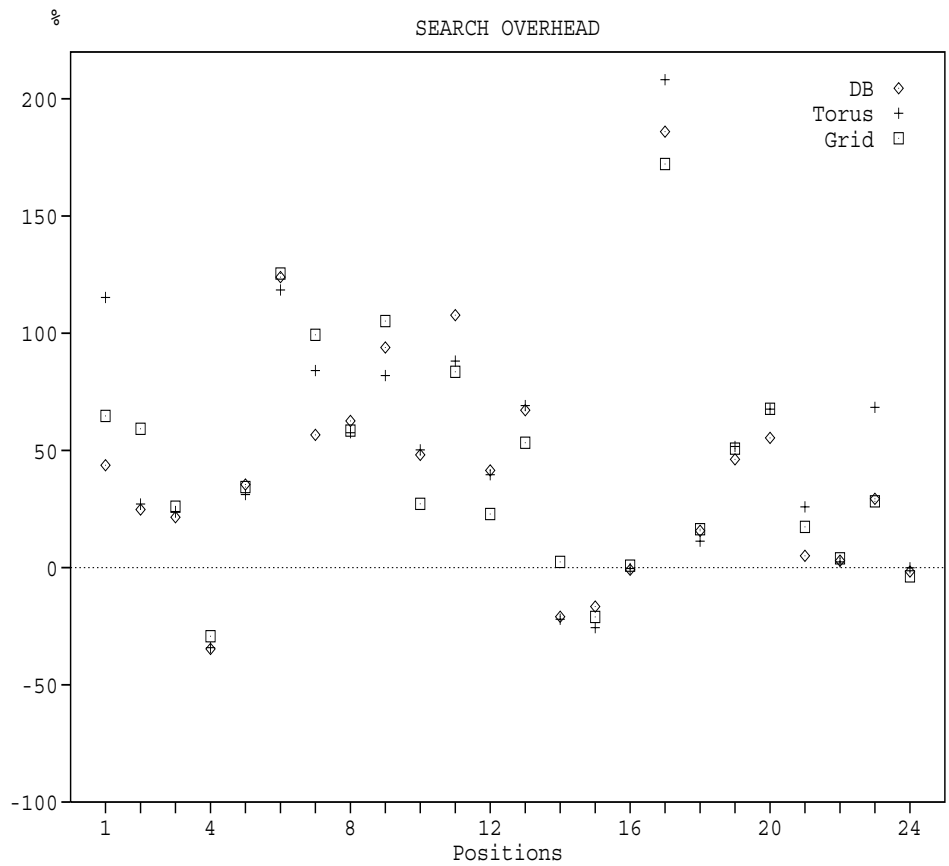


Figure 4.18: Search overhead of the $DB(8)$, $T(16,16)$, and $G(16,16)$

have been used successfully in other parallel tree searching algorithms like, for instance, parallel Branch & Bound, their use for a parallel game tree search seems questionable, because in general the evaluation of a node v may be finished only after all successors of v were completely evaluated. Subproblems, which are stored in queues for later evaluation may very easily delay the evaluation of other subproblems.

We now must ask, how static method for the load distribution will influence the efficiency of our distributed algorithm. Statical methods, in this context, are methods, in which processors no longer are allowed to choose arbitrary other processors as a potential master, but the choice is reduced to some few processors. Unfortunately, a good static load distribution strategy for the parallel game tree search is not obvious. Since the structure of the subtrees of the game tree to be searched becomes apparent only when the subtrees are searched, it is uncertain, whether a good load distribution strategy exists for every game tree.

Tree structured processor networks are taken into consideration, whenever trees of any kind must be searched in parallel: The root of the processor tree branches the root of the tree to be searched, the other processors get subproblems, which they for themselves branch into smaller subproblems, only from their father in the processor tree, etc. In order to show, how important a dynamic load distribution is in our parallel algorithm, we use the idea of the processor tree

	SPE	LOAD	SOVHD	PERF	PERF loss
TREE	13.84	6.40 %	7.36 %	90.59 %	9.41 %
Std	142.62	74.07 %	16.62 %	87.73 %	12.27 %

Table 4.30: Static and dynamic load distribution

again. Since we only want to use the tree structure of the processor network, to analyze the influence of the dynamic load distribution, it is not necessary to find an optimal processor tree for 256 processors. Thus, processor trees may be chosen more intelligently than in the following way: The network master is the root of the processor tree. 15 processor are connected to the network master as its successors. Each of these has 16 successors. The algorithm now is changed, such that a processor may ask for work only its father in the processor tree.

TREE:

We reduce the freedom of the processors searching for work by simulating the load distribution in a processor tree with 256 processors on the DeBruijn network $DB(8)$ with 256 processors.

Processors are not allowed to transmit requests to other processors, if they are not able to send a subproblem. Of course, the TREE-version uses the YBWC-1-2*.

The results of such a version are as bad as expected. E.g. all the right subtrees of the root of the game tree are searched with 17 processors, independently of their sizes.

Table 4.30 shows the results obtained for the static version (TREE) compared to the average results obtained for the standard version in Section 4.2. As expected, the work load of the static version is miserable.

Observation 4.14 *The dynamic load distribution is a very important presumption for the high efficiency of our distributed game tree search algorithm.*

Chapter 5

Behaviour on More Than 256 Processors

In this section we will describe the behaviour of our algorithm using more than 256 processors. For this we will use a machine different from the Parsytec SC 320 described in Section 3.1 and we will have to do some changes in the experiments, in order to get data of practical importance.

5.1 Changes of the Experiments

The experiments of the proceeding part of this book were done on a Parsytec SC 320 machine with 320 T800 processors. The experiments of this section, using 512 and 1024 processors are done on a Parsytec GCel with 1024 T805 processors. The following section describes the main differences between these two machines.

The GCel with 1024 Processors

The basis of the GCel is the T805 Transputer, which differs from the T800 by an increase of the clock rate from 25 MHz to 30 MHz. Tests with the sequential version of our program show that the running time of the T805 is exactly $\frac{5}{6}$ of the running time of the T800.

The above mentioned advantage, however, is more than compensated for by the fact, that a choice of the processor topology is not possible: The processors of the GCel are connected as a grid. Since the processors are connected directly, i.e. without routing hardware between them, for this machine the disadvantages of the grid topology observed in Section 4.8 occur. The transmission rate of the links is 20 Mbit/second just as for the T800 Transputer of the SC. However, the fixed interconnection network of the GCel has the advantage, that the messages no longer have to be routed via switches. Thus, a slightly larger communication speed is achieved between two neighbored processors of the GCel. The GCel may be partitioned into smaller subgrids.

The host processor of a GCel partition is part of the grid, such that e.g. a 2×2 - grid has three working processors and one host processor, which does not participate in the game tree search. In this part of the work we will especially investigate the behaviour of our algorithm on a 32×16 - grid with 512 processors as well as on a 32×32 - grid with 1024 processors. Using

		nodes	time	nodes/sec
SC: T800	dpth 8	345573567	660229	523.41
GCel: T805	dpth 8	345573567	550191	628.09
SC: T800	dpth 9*	1436971292	2621294	548.19
GCel: T805	dpth 9*	1436971292	2184412	657.83

Table 5.1: Sequential Version running on the SC and the GCel (computed)

such numbers of processor in practice it nearly makes no difference, whether one processor is used for input/output only, or not.

Because of these changes in the hardware, we changed our software in the following ways:

Implementation Details of the GCel Version

Using the GCel the host processor is part of the grid. In our case the host processor in any grid has the coordinates $(0,0)$. In order to keep the maximum distance of the processors from the network master as small as possible, this was placed onto the central processor $(\lfloor\frac{n}{2}\rfloor, \lfloor\frac{m}{2}\rfloor)$ of the $n \times m$ - grid. This results in a smaller maximal distance of the processors to the network master.

The effects of the following further changes will be investigated in this section:

1. Change from the Scout-algorithm to the $\alpha\beta$-algorithm in the parallel case.
2. Reduction of the number of transposition table accesses.

Experiments

In the following, we will study the effects of the last two changes described above. For this, we first will look at the behaviour of the standard version running on the grids $G(32,16)$ and $G(32,32)$ evaluating to search depths of 8 and 9*. Then, we will compare the standard version with the $\alpha\beta$-algorithm. Afterwards, we will describe a reduction of the use of the distributed transposition table, which significantly increases the efficiency of the parallel algorithm. To get speedup measurements we will multiply the running times of the sequential algorithm on the SC by the factor $\frac{5}{6}$, which results in the sequential running times on the GCel. In Table 5.1 the number of nodes visited, the sum of the running times for the 24 positions of the Bratko Kopec set and the number of nodes per second are summarized for the sequential version running on the SC and the GCel.

5.2 The Standard Version

Table 5.2 shows the results obtained for the standard version running on 512 and 1024 processors. The data was measured during an 8-ply search and 9$*$-ply search respectively of the 24 Bratko Kopec positions. The results of the $DB(8)$ are calculated with respect to the sequential version

	SPE	LOAD	SOVHD	PERF	t_Φ
8-ply search					
SC:$DB(8)$	142.62	74.07 %	16.62 %	87.73 %	193
GC:$G(32,16)$	160.13	59.37 %	28.87 %	68.01 %	143
GC:$G(32,32)$	201.17	43.73 %	42.90 %	64.18 %	114
9*-ply search					
SC:$DB(8)$	174.56	83.46 %	8.72 %	88.84 %	626
GC:$G(32,16)$	200.53	73.63 %	30.00 %	69.32 %	454
GC:$G(32,32)$	308.71	56.12 %	24.55 %	66.89 %	295

Table 5.2: Standard version running on 512 and 1024 processors

running on the SC, the results of the GCel versions are calculated with respect to the one running on the GCel.

During the 8-ply searches the search overhead of 16.62% for the $DB(8)$ increases to 28.87% for the $G(32,16)$ and 42.90% for the $G(32,32)$. Thus, the work done increases from 116.62% for the $DB(8)$ by $(\frac{142.90}{116.62} - 1) \cdot 100\% = 22.53\%$ only, although the number of processors has been increased by a factor of four. The processor work load, however, significantly decreases during the 8-ply searches from 74.07% in the $DB(8)$ to 43.73% only for the $G(32,32)$, i.e. by 30.34%. This is what must be expected, because the average running time of 114 seconds is very small. Even more important is the performance loss of 35.82% for the $GC(32,32)$. Waiting times for transposition table messages are negligible even in the 1024 processor system: the sum of all waiting times for transposition table messages is 2.31 seconds with a total running time of 2735 seconds. Therefore, the waiting times are no reason for the performance loss, i.e. in the 1024 processor system more than one third of the CPU time is used for routing messages.

The same effects can be observed for the 9*-ply searches. However, the average work load of 56.12% is significantly better than for the 8-ply searches. The average running time of 295 seconds per problem of the 1024 processor version is within tournament speed. 1024 processors achieve a speedup of 308.71 compared to a sequential GC-processor. An interesting effect is that the search overhead of the 1024 processor version decreases by 5.45% compared to the search overhead of the 512 processor version. There are three reasons for this:

1. Deviations of the search overhead.
2. The transposition table of the 1024 processor system, which is twice as large as the one of the 512 processor system.
3. The smaller processor work load of the 1024 processor system of 56.12%, which is significantly less than the 73.63% of the 512 processor system. For instance, although there are twice as many processors in the 1024 processor system, not twice as many processors participate in the game tree search on the average.

As a concluding observation we may state:

	SPE	LOAD	SOVHD	PERF	t_Φ
8-ply search					
GC:$G(32,16)$	138.90	59.76 %	34.41 %	61.16 %	165
GC:$G(32,32)$	169.55	46.75 %	48.52 %	52.71 %	135
9*-ply search					
GC:$G(32,16)$	196.87	74.09 %	22.49 %	63.70 %	462
GC:$G(32,32)$	275.12	64.79 %	33.57 %	55.48 %	331

Table 5.3: Standard version without N1 running on 512 and 1024 processors

Observation 5.1 *(Standard version running on the $G(32,16)$ and the $G(32,32)$)*

1. *The search overhead is reduced by the YBWC-1-2* in such a manner that even doubling of the number of processors from 512 to 1024 does not cause significant additional search overhead.*

2. *The average work load decreases greatly with an increase of the number of processors.*

3. *The average performance is only two thirds of the performance of the sequential version.*

Standard vs. Standard without N1

Observation 4.9 states that the use of the method N1 in the standard version running on the $DB(8)$ has no influence on the overall efficiency of the program. In the large grids $G(32,16)$ and $G(32,32)$ the use of N1 influences the efficiency more significantly. Table 5.3 contains the results obtained for the version without N1. If one, for example, looks at the 9*-ply searches with the $G(32,32)$, then by the use of N1 the performance is increased by 11.41%. On the other hand, the average processor work load decreases by 8.67%. The version without N1, as already observed in Section 4.7, produces a somewhat larger search overhead. As a total, the efficiency is much better using N1. The same effects, although slightly weaker, can be observed for the $G(32,16)$. The fact that the search overhead of the version with 512 processors using N1 is slightly smaller than the search overhead of the standard version indicates, that the 30.00% search overhead of the standard version is increased by deviations.

Observation 5.2 *The standard version achieves an improvement of the speedup of from 169.55 to 201.17 (275.12 to 308.71) compared to the version without N1 using 1024 processors during an 8- (9*-) ply search. This corresponds to an increase of the speed of*

$$\frac{201.17 - 169.55}{169.55} \cdot 100\% = 15.3\% \text{ or } \frac{308.71 - 275.12}{275.12} \cdot 100\% = 12.2.\% \text{ resp.}$$

In the next two sections we will describe two modifications to the algorithm, which will improve the efficiency drastically.

	SPE	LOAD	SOVHD	PERF	t_Φ
8-ply search					
GC:$G(32,16)$	164.28	61.57 %	32.25 %	69.12 %	140
GC:$G(32,32)$	206.99	46.73 %	50.42 %	65.14 %	111
9*-ply search					
GC:$G(32,16)$	210.95	74.75 %	27.12 %	70.23 %	431
GC:$G(32,32)$	322.37	62.41 %	27.02 %	64.16 %	282

Table 5.4: Standard version without zero window search using 512 and 1024 processors

5.3 Scout vs $\alpha\beta$: Zero Window Search ?

Observation 5.1 states, that for the standard version the average work load heavily decreases when doubling the processors from 512 tp 1024.

Zero Window Search ?

The synchronization nodes, which result from the use of the YBWC-1-2* and the iterative deepening, require a complete redistribution of the work load, which last longer the larger the processor network is. On the other hand, by iterative deepening the algorithm accumulates the information necessary to obtain a good move ordering during the search of the deeper game trees, such that the algorithm has to use both methods. The number of synchronization nodes as well as the number of moments, in which only very few nodes are alive, may be decreased, if the pure $\alpha\beta$-algorithm is used for the massively parallel versions instead of the Scout-algorithm. The Scout-algorithm has to research e.g. a successors of the root after a fail high search of this node. In many cases we observed that these researches are begun at a time, at which only very few subproblems are alive. Thus, we get additional moments, at which no load distribution is possible, because the number of processors is much larger than the number of living subproblems. If one does not use the zero window search, then there are no researches. By doing so, however, one gets a search algorithm, which even in the sequential version is slightly worse than the Scout-algorithm. In addition, the parallel algorithm may use the tight bounds of the Scout-algorithm to avoid search overhead. By not using the zero window search the artificial tight bounds may be replaced by possibly untight ones. Thus, when switching to the pure $\alpha\beta$-algorithm, the search overhead should increase.

Results Without Zero Window Search

Table 5.4 shows the results obtained for the version without zero window search running on 512 and 1024 processor. The processor work load of the pure $\alpha\beta$-algorithm is better than the one of the Scout-algorithm. For example, the average work load for a 9*-ply search with 1024 processors increases from 56.12% to 62.41%. The average search overhead, however, only increases from 24.55% to 27.02%, which may even be caused by deviations. The somewhat better performance of the Scout version may be due to the fact that the evaluation of the game tree nodes need not to be equally fast: The nodes, which are generated within a research can be generated very fast, because important information may be found in the transposition table. Thus the performance of the pure $\alpha\beta$-algorithm is slightly smaller. Summing up we may state:

	SPE	LOAD	SOVHD	PERF	t_Φ
8-ply search					
GC:$G(32,16)$	176.80	60.99 %	53.52 %	87.12 %	130
GC:$G(32,32)$	237.15	46.12 %	68.40 %	84.57 %	97
9*-ply search					
GC:$G(32,16)$	237.03	74.56 %	39.82 %	86.96 %	384
GC:$G(32,32)$	344.49	62.67 %	55.55 %	83.61 %	264

Table 5.5: Reduced TR-Accesses with 512 and 1024 processors

Observation 5.3 *The improvements of the speedup caused by switching from the Scout-algorithm to the $\alpha\beta$-algorithm are between* 2.6% *for an* 8*-ply search with the* $G(32,16)$ *and* 5.2% *for a 9*-ply search with the* $G(32,16)$.

Therefore the reduction of the number of transposition table accesses described in the next section will be done in the $\alpha\beta$-version instead of the Scout version.

5.4 Reduced Transposition Table Accesses

In addition to the poor average work load of the standard version in large grids, the efficiency is influenced by the large performance loss. Corollary 4.10 states that the performance loss is caused mainly by the transposition table messages. Therefore, we reduce the number of accesses to the transposition table to get a better performance. Any reduction of the use of the transposition table, however, must cause additional search overhead.

The deepest nodes of the game tree, for which a transposition table access is done in the standard version, are the fathers of leaves of the brute force search tree. For these nodes a transposition table access is less important, because the subtrees below them are very small. In addition, most of the nodes, for which the transposition table is accessed lie in the level of the fathers of the leaves, the communication necessary for the transposition table can be reduced greatly by simply not allowing transposition table accesses for the fathers of the leaves any more. This, however, does no make sense as long as, correspond to the iterative deepening method, only small search trees are evaluated. Table 5.2 shows that the standar version is able to run an 8-ply search faster than tournament speed on the average, such that during a game under tournament conditions almost every move is evaluated at least to a search depth of 8. Thus, we decided to reduce the transposition table accesses in such a way, that for search depths ≥ 8 the transposition table is no more accessed for the fathers of the leaves of the brute force tree.

Table 5.5 shows the results obtained for the $\alpha\beta$-version with reduced transposition table accesses. Compared to the standard version without zero window search the speedup is improved from 322.37 to 344.49 with 1024 processors doing a 9*-ply search. This improvement mainly is due to an increase of the performance by $83.61 - 64.16\% = 19.45\%$, i.e. more than half of the performance loss of the original version are avoided. On the other hand, the search overhead worsens: it increases from 27.02% to 55.55%. The average processor work load remains constant.

For the increase of the search overhead there are mainly two reasons:

	BF		QS	
depth	type 2	type 3	type 2	type 3
9*/TR+	0.83 %	0.95 %	1.20 %	1.13 %
9*/TR−	0.90 %	1.03 %	1.16 %	1.12 %

Table 5.6: Percentage of the nodes having a wrong expected type

1. In the case of transpositions to the same position at the level of the fathers of the leaves equal subtrees are searched twice now.
2. The reduced transposition table accesses results in a worse move ordering in the deeper part of the brute force search tree, which has negative influences on the search of the parallel algorithm too.

The second effect can be observed from the data of Table 5.6. By the reduction of the transposition table accesses the quality of the expected type of nodes function gets worse for the nodes of the brute force tree, it remains unchanged for the nodes of the quiescence search.

The increase of the performance is caused by the largely reduced number of transposition table messages. For instance, in the standard $\alpha\beta$-version roughly 156.4 million TR-requests are sent during a 9*-ply search, in the version with the reduced accesses 43.8 million only. This corresponds to a reduction by 72%.

The average processor work load is not changed by the reduction of the transposition table accesses, although the average running time per problem is slightly smaller. The better performance allows a faster decomposition of the game trees into subtrees resulting in a better load distribution.

Observation 5.4 *The version with the reduced transposition table accesses shows improved speedups compared to the standard version. The largest improvement is by* 14.6% *for an* 8-*ply search with* 1024 *processors. The smallest improvement observed is by* 6.9% *for a 9*-ply search with* 1024 *processors.*

5.5 DB(8) vs. GC(32,32) under tournament conditions

The parallelization presented in this work is used in the distributed chess program *ZUGZWANG*. The successor version ZZ 5.62x of the version ZZ 5.42x discussed in this work, which, other than a few minor details, uses the parallelization described above became Vice World Champion at the Computer Chess World Championships in Madrid, Spain, in November 1992. It ran on a $G(32,32)$ and finished the tournament with three wins and two draws, i.e. without a loss. The standard version for the DeBruijn network became chess champion of the city of Paderborn, Germany, in December 1991, running on a $DB(8)$. Almost the same program became second best computer at the 7th AEGON Man vs. Machine Tournament in The Hague, Netherlands, in May 1992. During the latter two tournaments, *ZUGZWANG* played at a rating of approximately 2300 ELO points (German ELO rating). This corresponds to an ELO rating of approximately 2400 (USCF). Therefore, it may be of interest to compare the version described last for the 32×32 - grid and the standard version running on the $DB(8)$. This comparison is done below.

	SPE	LOAD	SOVHD	PERF	t_Φ
Speedup					
SC:$DB(8)$ wrt T800	174.56	83.46 %	8.72 %	88.84 %	626
GC:$G(32,32)$ wrt T805	344.49	62.67 %	55.55 %	83.61 %	264
GC:$G(32,32)$ wrt T800	413.39	62.67 %	55.55 %	96.32 %	264

Table 5.7: Standard version on the $DB(8)$ vs. $\alpha\beta$on the $G(32,32)$

We will compare the absolute speed of the $G(32,32)$ and the $DB(8)$ to get an approximation for the playing strength of *ZUGZWANG* running on the $G(32,32)$. We may presume that *ZUGZWANG* running on the $DB(8)$ has a playing strength of 2300 ELO approximately.

There are several different conjectures for the question, how playing strength and computing power are related. Nau [51] analyzes game trees, in which the deeper searches return worse values than the shallow ones. Schrüfer [67] analyzed the s-tree model and showed for these class of game trees that the results of deeper searches are worse than the results of shallower ones, if the probability of an erroneous static evaluation of the leaves becomes too large. In practice, i.e. in game playing programs, this so called pathology has never been observed. Newborn [52] conjectured an increase of the playing strength by more than 100 ELO points for each doubling of the computational power. Levy and Newborn [44] analyzed the increase of the playing strength as a function of the search depth, which are reached by the programs under tournament conditions. They come to the conclusion that in the aera from 1972 to 1989 each additional ply, which corresponds to the five- to sixfold speed of the computation, corresponded to an increase of the playing strength of 200 ELO points. The last approximation, however, does not consider all the development of the software during this time, such that one may conjecture that an increase of the computational power does not necessarily correspond to an increase of the playing strength of 200 ELO points. Szabo and Szabo [71] extend the experiments of Condon and Thompson [17, 72] and come to the result that the playing strength grows as

$$R(P) = \frac{1}{-0.0756} \cdot e^{-0.0756 \cdot P + 5.33} + 2963.$$

Here P is the playing strength scaled logarithmically in the computational power, i.e. doubling the speed increases P by one. $R(P)$ is the ELO number reached. This function grows slower the larger the playing strength is. The fact remains that the faster computer is the better one.

Table 5.7 shows the data measured during a 9*-ply search for the $DB(8)$ on the SC, as well as for the $G(32,32)$ on the GCel. The first line contains the results of the $DB(8)$ with respect to a sequential SC-processor, the second line the results of the $G(32,32)$ measured with respect to a sequential GCel-processor. The lowest line of the table shows the results for the $G(32,32)$ with respect to the sequential SC-processor version. With this the speedup of the $G(32,32)$ on the GCel compared to the $DB(8)$ on the SC can be shown.

Observation 5.5 *The $G(32,32)$ achieves a speedup of 413.39 compared to th e sequential SC-processor and thus, consisting of the fourfold number of processors, is*

$$\frac{413.39}{174.56} = 2.37$$

times as fast as the $DB(8)$ on the SC.

This speedup mainly is due to the larger average performance, which is achieved by the reduction of the use of the transposition table as well as by the higher clock rate. One has to pay for the reduction of the use of the transposition table by a significant increase of the average search overhead. The processor work load is much worse in the $G(32,32)$ caused by the two dimensional grid structure of the network and the smaller average running times.

ZUGZWANG on the $DB(8)$, having 2300 ELO points, has a playing strength according to Szabo and Szabo [71] of $P_{2300} \approx 18.73$, therefore the playing strength of the $GC(32,32)$ is $18.73 + log_2\ 2.37 = 19.97$. With this we get the following conjectures about the paying strength of *ZUGZWANG* on the $GC(32,32)$:

Corollary 5.1 *ZUGZWANG on the $GC(32,32)$ has an approximated ELO number of*

- *2424 following Newborn [52]*
- *2396 following Levy and Newborn [44]*
- *2360 according to the Szabo curve [71]*

With this, *ZUGZWANG* is amongst to the strongest chess programs in the world.

Chapter 6

Conclusions and Open Problems

In this thesis, we described a parallelization of the game tree search, which enables us to use massively parallel systems efficiently. In addition, with the parallelization presented, the use of all state of the art techniques for speeding up the sequential search is possible. The parallelization presented here is more efficient than any other parallel game tree search algorithm known before.

We showed that the success of the parallelization is due to the following features:

1. Dynamic load distribution in all parts of the game tree to be searched.
2. Local criteria, whether or not to use parallelism.
3. The choice of a suitable processor network.
4. Efficient implementation of a distributed hash table.
5. Development of a local and global load distribution strategy, which does not have the disadvantages of a purely local or purely global load distribution, but instead improves the move ordering of the game trees to be searched.

The parallelization is of practical importance, since it is successfully used in the chess program *ZUGZWANG*.

Some problems, however, remain open:

Figure 3.20 in Section 3.7 describes, how synchronization nodes occur during the parallel search.

Open problem 1 *(Iterative deepening)*

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

Open problem 2 *(Tradeoff SOVD-LSTG)*

In massively parallel systems without routing hardware the use of a distributed transposition table causes a large loss of the performance, reducing the number of accesses to the table, however, results in a large search overhead. Is there a combination of

Hash function & Parallelization strategy & Processor topology,

which makes the accesses to a large transposition table cheaper without increasing the search overhead or decreasing the work load ?

There are some theorems claiming a linear speedup of some parallel game tree search algorithms [6, 39, 33]. These theorems, however, are of no practical importance, because

1. the theoretical models of the game trees investigated are far from the game trees occurring in practical applications,
2. the game trees to be searched have to be unrealistically large to achieve a linear speedup and
3. the constants of the linear function describing the speedup are very small.

In practice several difficulties arise, which result in a speedup clearly smaller than the number of processors. In the past several alternatives to the brute force search with the help of the $\alpha\beta$- or Scout-algorithm respectively were developed, e.g. the B*-algorithm presented by Berliner [10], which becomes more and more important very recently by the successful use in the chess program *B*-HITECH* or the *Conspiracy Number Search* algorithm of McAllester [48]. The game trees searched by these algorithms in general are of a much more nonuniform structure than the game trees searched by a brute force search. The question of how well these algorithms parallelize is still an open one:

Open problem 3 *($\alpha\beta$-alternatives)*

Is there a better parallelization for the B-algorithm or the Conspiracy Number Search algorithm than for the $\alpha\beta$-algorithm ?*

Human chess players seem to search a game tree extremely selectively. The best human players are still superior to the best machines, which imposes the question, if the computation of a move based on the brute force search with the help of the $\alpha\beta$- or Scout-algorithm will lead to a machine playing complex game like chess stronger than the best humans in the near future. Therefore alternatives to the brute force search should be studied carefully.

Acknowledgement

I would like to thank my supervisor Prof. Dr. Burkhard Monien, who gave me the opportunity to do this research. Additionally he supported the development of the chess program *ZUGZWANG*, as an application for the parallelization described in this book, and with many discussions and many hints he contributed to its success. With many helpful comments he positively influenced the final form of this book.

In the same way I would like to thank my colleague Peter Mysliwietz, the second "father" of *ZUGZWANG*, for the good cooperation over many years.

Furthermore, I would like to thank Dr. Ingo Althöfer and Dr. Walter Unger for carefully reading the German version of this work and for their helpful comments.

Appendix A

A BCH Hash Function

In order to generate a BCH Code, we will proceed as described by Berlekamp [9]. We compute the cyclotomic polynomial

$$Q^{(1023)}(x) := \prod_{\alpha}(x - \alpha),$$

where α becomes any element from the set of the primitive 1023rd roots of unity. This polynomial can be determined without explicitly knowing all the roots of unity. It is described in a short notation below:

$Q^{(1023)}(x) = 1$	1100000000	1110000000	0000000000	1101100000	0110110000
	0000000000	0001101100	0000110110	0000000000	0011101101
	1001110110	1100000000	0001101101	1011110110	1101100000
	0000001101	1011011110	1101101100	0000011101	1011010101
	1101101101	1000001101	1011011101	1011101101	1011000001
	1011011011	1011011101	1011011011	1011011010	1011011011
	1011011010	1011011011	1011011011	0111011011	1011011011
	0000011011	0110111011	0111011011	0110000011	0110110111
	0101011011	0111000000	0110110110	1111011011	0110000000
	0000110110	1101111011	0110110000	0000000110	1101110011
	0110111000	0000000000	1101100000	0110110000	0000000000
	0001101100	0000110110	0000000000	0000001110	0000000111

Here only the coefficients of $Q^{(1023)}(x)$ are given, the polynomial itself therefore is

$$Q^{(1023)}(x) = x^{600} + x^{599} + x^{598} + x^{589} + \ldots + x^2 + x + 1.$$

The last numbers are the coefficients of the small powers of x.

A factor of $Q^{(1023)}$, which is irreducible over $GF(2)$ is

$$m^{(1)}(x) = x^{10} + x^3 + 1$$

or

$$m^{(1)}(x) = 1\ 00000\ 01001$$

for short.

We represent each element of the finite field $GF(2^{10})$ as a residue class modulo $m^{(1)}(x)$, which enables us to add, substract, multiply and divide the elements of $GF(2^{10})$. Each residue class is represented by the uniquely determined polynomial of degree < 10 of the class. Then in our short notation

$$\alpha = [x]_{m^{(1)}} = [10]_{m^{(1)}}$$

is a root of $m^{(1)}(x)$, and thus $m^{(1)}(x)$ is the minimal polynomial of α. Table A.1 contains the powers $\alpha^1, \ldots, \alpha^{19}$ of α and the corresponding minimal polynomials $m^{(i)}(x)$ of α^i.

i	α^i	$m^{(i)}(x)$
1	10	1 00000 01001
2	100	1 00000 01001
3	1000	1 00000 01111
4	10000	1 00000 01001
5	1 00000	1 01000 01101
6	10 00000	1 00000 01111
7	100 00000	1 11111 11001
8	1000 00000	1 00000 01001
9	10000 00000	1 00101 01111
10	1001	1 01000 01101
11	10010	1 00001 10101
12	1 00100	1 00000 01111
13	10 01000	1 00011 01111
14	100 00000	1 11111 11001
15	1001 00000	1 01101 01011
16	10010 00000	1 00000 01001
17	100 01001	1 11010 01101
18	1000 10010	1 00101 01111
19	10001 00100	1 01111 11011

Table A.1: Powers of α and the corresponding minimal polynomials

Note that α^i and α^{2i} are binary conjugates and thus have the same minimal polynomial.

A BCH-code with a Hamming distance of $2 \cdot t + 1$, $t \geq 1$, is now obtained by using the product of the <u>distinct</u> minimal polynomials $m^{(1)}(x), \ldots, m^{(2 \cdot t - 1)}(x)$ as a generator polynomial $g(x)$, i.e.

$$g(x) := lcm\{m^{(1)}(x), m^{(3)}(x), \ldots, m^{(2 \cdot t - 1)}(x)\}.$$

Especially for our chess program, let $s \in \{0,1\}^{749}$ be the representation of a position. For $t \leq 27$ the BCH-code with a Hamming distance of $2 \cdot t + 1$ has a parity check matrix

$$H := \begin{pmatrix} 1 & \alpha^1 & \alpha^2 & \alpha^3 & \ldots & \alpha^{748+10 \cdot t} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \ldots & \alpha^{(748+10 \cdot t)3} \\ & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^{2t-1} & \alpha^{2(2t-1)} & \alpha^{3(2t-1)} & \ldots & \alpha^{(748+10 \cdot t)(2t-1)} \end{pmatrix}.$$

Here the α^i are the powers of the primitive 1023rd root of unity α, written as polynomials of degree < 10 in the short notation including the leading zeroes, e.g. $\alpha^1 = [10]_{m^{(1)}} = (00000\ 00010)^T$, $\alpha^2 = [100]_{m^{(1)}} = (00000\ 00100)^T$, etc.

By adding a parity check line to the parity check matrix we another parity check matrix

$$\hat{H} := \begin{pmatrix} 1 & \alpha^1 & \alpha^2 & \alpha^3 & \dots & \alpha^{748+10\cdot t} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \dots & \alpha^{(748+10\cdot t)3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^{2t-1} & \alpha^{2(2t-1)} & \alpha^{3(2t-1)} & \dots & \alpha^{(748+10\cdot t)(2t-1)} \\ 1 & 1 & 1 & 1 & \dots & 1 \end{pmatrix}.$$

Each codeword $x \in \{0,1\}^{749+10\cdot t+1}$ now satisfies

$$\hat{H} \cdot x = 0.$$

By diagonalization we obatin a new parity check matrix

$$H' = [I|A]$$

with

$$H' \cdot x = 0 \Leftrightarrow \hat{H} \cdot x = 0.$$

Here I is the $(10 \cdot t + 1) \times (10 \cdot t + 1)$ unit matrix. Now a codeword $x \in \{0,1\}^{749+10\cdot t+1}$ is of the form

$$x = \underbrace{x_1 \cdots x_k}_{Checkbits} \underbrace{x_{k+1} \cdots x_n}_{Positionbits},$$

with $k = 10 \cdot t + 1$ and $n = 749 + 10 \cdot t + 1$.

Then we have

$$x_1 = -A_{1,\cdot} \cdot s = A_{1,\cdot} \cdot s, x_2 = A_{2,\cdot} \cdot s, \dots$$

We get a BCH hash function

$$h : \{0,1\}^{749} \to \{0,1\}^{10\cdot t+1}$$

by simply defining

$$h(s) := A \cdot s.$$

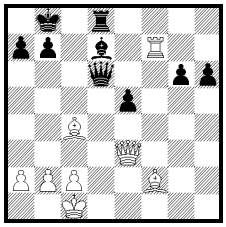
The representation of a chess position is used only implicitly. If e.g. v is a position having a representation s and a hash value $h(s)$ and v' is a successor of v in the search tree, then the representations of v and v' only differ in a few (very often only two) bits. The indices, at which the representations differ, are determined uniquely by the move from v to v'. If $i_1, \dots i_k$ are the indices of the bits, at which the representations of v and V' differ, then the hash value $h(s')$ of v' is obtained by computing

$$h(s') := h(s) \bigoplus_{j=1}^{k} A_{\cdot,i_j},$$

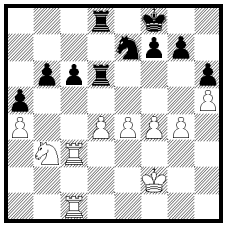
i.e. by some few XOR-operations.

Appendix B

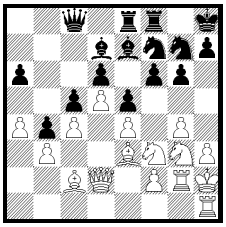
The Bratko Kopec Positions



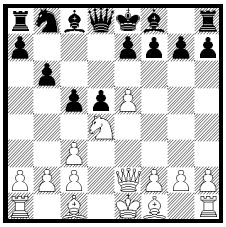
B01
(Black to move)



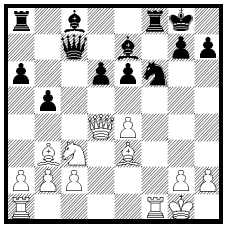
B02
(White to move)



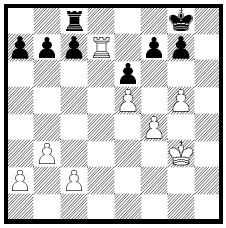
B03
(Black to move)



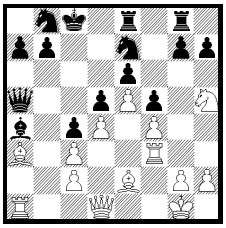
B04
(White to move)



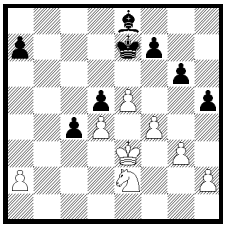
B05
(White to move)



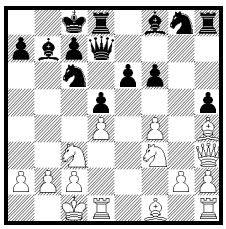
B06
(White to move)



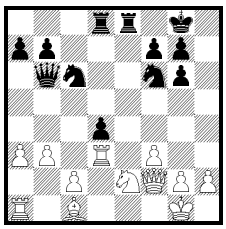
B07
(White to move)



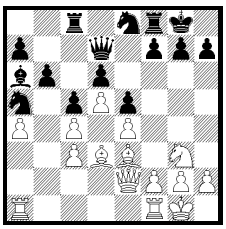
B08
(White to move)



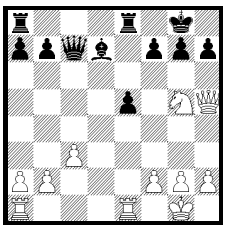
B09
(White to move)



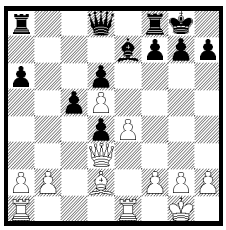
B10
(Black to move)



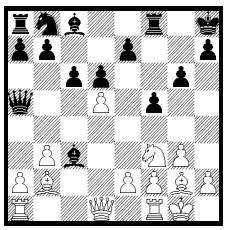
B11
(White to move)



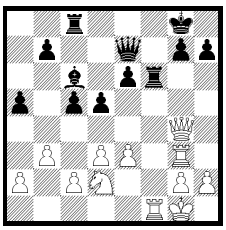
B12
(Black to move)



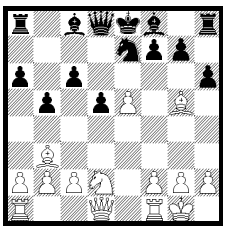
B13
(White to move)



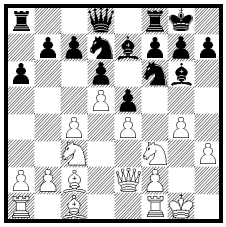
B14
(White to move)



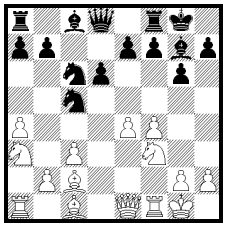
B15
(White to move)



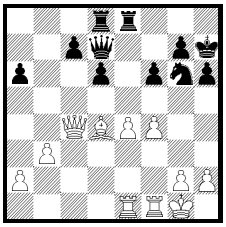
B16
(White to move)



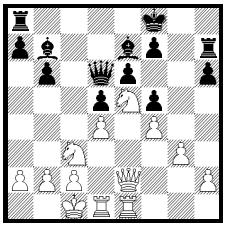
B17
(Black to move)



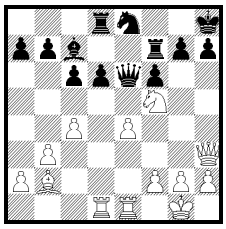
B18
(Black to move)



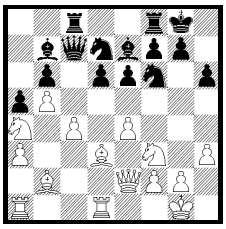
B19
(Black to move)



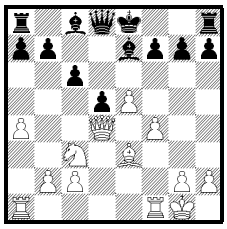
B20
(White to move)



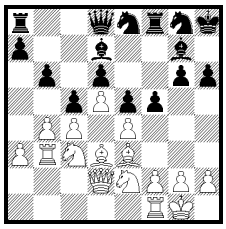
B21
(White to move)



B22
(Black to move)



B23
(Black to move)



B24
(White to move)

Bibliography

[1] *S.G. Akl, M. Newborn* **The priciple continuation and the killer heuristic** ACM Annual Conference, pp 466-473, (1977)

[2] *S.G. Akl, D.T. Barnard, R.J. Doran* **Simulation and Analysis in Deriving Time and Storage Requirements for a Parallel Alpha-Beta Pruning Algorithm** IEEE International Conference on Parallel Processing, pp 231-234, (1980)

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

[4] *K. Almquist, N. McKenzie, K. Sloan* **An Inquiry into Parallel Algorithms for Searching Game Trees** Technical Report, University of Washington, Seattle, USA, (1988)

[5] *I. Althöfer* **The Complexity of a Simple Stochastic OR-Tree Model in Which "Directional Search" Is Bad** Journal of Algorithms, Vol 6, pp 264-277, (1990)

[6] *I. Althöfer* **A Parallel Game Tree Search Algorithm with a Linear Speedup** erscheint in Journal of Algorithms, (1993)

[7] *G.M. Baudet* **On the Branching Factor of the Alpha-Beta Pruning Algorithm** Artificial Intelligence, Vol. 10, pp 173-199, (1978)

[8] *G. Baudet* **The Design and Analysis of Algorithms for Asynchronous Multiprocessors** Doktorarbeit, Carnegie-Mellon University, Pittsburgh, USA, (1978)

[9] *E.R. Berlekamp* **Algebraic Coding Theory** Revised Edition, Aegean Park Press, (1984)

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

[11] *H.J. Berliner* **Computer Chess at Carnegie-Mellon University** Advances in Computer Chess IV, D.F. Beal (Editor), Pergamon Press, pp 166-180, (1984)

[12] *M. Böhm, E. Speckenmeyer* **A Dynamic Processor Tree for Solving Game Trees in Parallel** Technical Report, University of Dortmund, Germany, (1989)

[13] *I. Bratko, D. Kopec* **A Test for Comparison of Human and Computer Performance in Chess** Advances in Computer Chess III, M.R.B. Clarke (Editor), Pergamon Press, pp 31-56, (1982)

[14] *A. Burns* **Programming in Occam 2** Addison-Wesley Publishing Company, (1988)

[15] *M.S. Campbell, T.A. Marsland* **A Comparison of Minmax Tree Search Algorithms** Artificial Intelligence, Vol. 20, pp 347-367, (1983)

[16] *J.H. Condon, K. Thompson* **Belle Chess Hardware** Advances in Computer Chess III, M.R.B. Clarke (Editor), Pergamon Press, pp 44-54,(1982)

[17] *J.H. Condon, K. Thompson* **Belle** Chess Skill in Man and Machine, P.W. Frey (Editor), Springer Verlag, pp 201-210,(1983)

[18] *N.G. DeBruijn* **A Combinatorial Problem** Indagationes Math., Vol. 8, pp 461-467, (1946)

[19] *A.E. Elo* **The Rating of Chessplayers, Past and Present** Arco Publishing, New York, 1978

[20] *R. Feldmann, P. Mysliwietz, O. Vornberger* **A Local Area Network Used as a Parallel Architecture** Technical Report, University of Paderborn, West Germany, (1986)

[21] *R. Feldmann, P. Mysliwietz* **Parallele Spielbaumsuche** Diplomarbeit, University of Paderborn, Germany, (1988)

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

[23] *R. Feldmann, B. Monien, P. Mysliwietz, O. Vornberger* **Distributed Game Tree Seach** Parallel Algorithms for Machine Intelligence and Vision, V. Kumar, L.N. Kanal, P.S. Gopalakrishnan (Editors), Springer Verlag, pp 66-101, (1990)

[24] *R. Feldmann, B. Monien, P. Mysliwietz* **A Fully Distributed Chess Program** Advances in Computer Chess VI, D.F. Beal (Editor), pp 1-27, (1991)

[25] *R. Feldmann, P. Mysliwietz, B. Monien* **Spielbaumsuche auf einem Transputernetzwerk** Parallel - Algorithmen und -Rechnerstrukturen (PARS), Workshop Sprachen und Systeme zur Parallelverarbeitung, Gesellschaft für Informatik e.V. (1990)

[26] *R. Feldmann, P. Mysliwietz, B. Monien* **Experiments with a Fully Distributed Chess Program** Heuristic Programming in Artificial Intelligence 3, J. van den Herik, V. Allis (Editors), pp 72-87, (1992)

[27] *R. Feldmann, P. Mysliwietz, B. Monien* **Distributed Game Tree Search on a Massively Parallel System** in: Data structures and efficient algorithms: Final report on the DFG special joint initiative, Springer, Lecture Notes on Computer Science 594, B. Monien, T. Ottmann (Editors), pp 270-288, (1991)

[28] *Ch. Ferguson, R.E. Korf* **Distributed Tree Search and its Application to Alpha-Beta Pruning** Proceedings AAAI-88, Seventh National Conference on Artificial Intelligence, Vol. 2, pp 128-132, (1988)

[29] *R.A. Finkel, J.P. Fishburn* **Parallel Alpha-Beta Search on Arachne** IEEE International Conference on Parallel Processing, pp 235-243, (1980)

[30] *R.A. Finkel, J.P. Fishburn* **Parallelism in Alpha-Beta Search** Artificial Intelligence Vol. 19, pp 89-106, (1982)

[31] *R. Funke, R. Lüling, B. Monien, F. Lücking, H. Blanke-Bohne* **An Optimized Reconfigurable Architecture for Transputers** Proceedings of the 25th Hawaii Int. Conference on System Sciences (HICSS), Vol. 1, pp 237-245, (1992)

[32] *R.D. Greenblatt, D.E. Eastlake III, S.D. Crocker* **The Greenblatt Chess Program** Proceeding of the Fall Joint Computing Conference, San Francisco, pp 801-810, (1967)

[33] *J. Harting* **Analyse eines parallelen Spielbaumsuchverfahrens mit linearem Speedup** Diplomarbeit, Universität Bielefeld, Fakutät für Mathematik, Bielefeld, Deutschland, (1991)

[34] *F.H. Hsu* **Large Scale Parallelization of Alpha-Beta Search: An Algorithmic Architectural Study with Computer Chess** Phd. thesis, Carnegie Mellon University, Pittsburgh, USA, (1990)

[35] *M. M. Huntbach, F. W. Burton* **Alpha - Beta Search on Virtual Tree Machines** Information Sciences, Vol. 44, pp 3-17, (1988)

[36] *R.M. Hyatt* **Parallel Chess on the Cray X-MP/48** ICCA Journal, Vol. 8 No. 2, pp 90-99, (1985)

[37] *R.M. Hyatt, B.E. Gower, H.L. Nelson* **CRAY BLITZ** Advances in Computer Chess IV, D.F. Beal (Editor), Pergamon Press, pp 8-18, (1985)

[38] *R.M. Hyatt, B.W. Suter, H.L. Nelson* **A parallel alpha/beta tree searching algorithm** Parallel Computing, No. 10, pp 299-308, (1989)

[39] *R. M. Karp, Y. Zhang* **On Parallel Evaluation of Game Trees** Proceedings of SPAA'89, pp 409-420, (1989)

[40] *D.E. Knuth, R.W. Moore* **An Analysis of Alpha - Beta Pruning** Artificial Intelligence, Vol. 6, pp 293-326, (1975)

[41] *H.-J. Kraas* **Zur Parallelisierung des SSS*-Algorithmus** Doktorarbeit, Universität Braunschweig, Deutschland, (1990)

[42] *T. Leighton, B. Maggs, S. Rao* **Universal Packet Routing Algorithms** IEEE FOCS, pp 256-271, (1988)

[43] *C. Leiserson* **Persönliches Gespräch** (1992)

[44] *D. Levy, M. Newborn* **How Computers Play Chess** Computer Science Press, (1991)

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

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

[47] *T.A. Marsland, M. Olafsson, J. Schaeffer* **Multiprocessor Tree-Search Experiments** Advances in Computer Chess IV, D.F. Beal (Editor), Pergamon Press, pp 37-51, (1986)

[48] *D.A. McAllester* **A New Procedure for Growing Min-Max Trees** Artificial Intelligence, Vol. 35, No. 3, pp 287-310, (1988)

[49] *B. Monien, O. Vornberger* **Parallel Processing of Combinatorial Search Trees** Proceedings International Workshop on Parallel Algorithms and Architectures, Math. Research Nr. 38, Akademie - Verlag Berlin, pp 60-69, (1987)

[50] *P. Mysliwietz* **Konstruktion und Optimierung von Bewertungsfunktionen beim Schach (vorläufiger Titel)** Doktorarbeit, Universität GH Paderborn, Deutschland, (1993)

[51] *D.S. Nau* **Pathology on Game Trees Revisited and an Alternative to Minimaxing** Artificial Intelligence, Vol. 21, pp 221-244, (1983)

[52] *M. Newborn* **Computer Chess: Recent Progress and Future Expectations** Information Technology, J. Moneta (Editor), pp 189-192, (1978)

[53] *M. Newborn* **A Parallel Search Chess Program** ACM Annual Conference 1985, pp 272-277, (1985)

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

[55] *A. Newell, J.C. Shaw, H.A. Simon* **Chess-Playing Programs and the Problem of Complexity** IBM-Journal of Research and Developement, 2, pp 320-355, (1958) Reprint in: Computers and Thought, E.A. Feigenbaum, J. Feldman (Editors), McGraw-Hill, pp 39-70, (1963)

[56] *S.W. Otto, E.W. Felten* **Chess on a Hypercube** The Third Conference on Hypercube Concurrent Computers and Applications, Vol. 2, pp 1329-1341, (1988)

[57] *J. Pearl* **Asymptotic Properties of Minmax Trees and Game Searching Procedures** Artificial Intelligence, Vol. 14, pp 113-139, (1980)

[58] *J. Pearl* **Heuristics: Intelligent Search Strategies for Computer Problem Solving** Addison-Wesley Publishing Company, (1984)

[59] *A. Reinefeld* **Spielbaum-Suchverfahren** Springer-Verlag, Informatik-Fachberichte, Band 200, (1989)

[60] *J. Rezaie, R. Finkel* **A Comparison of some Parallel Game-Tree Search Algorithms** Technical Report, University of Kentucky, Lexington, USA, (1992)

[61] *I. Roizen, J. Pearl* **A Minmax Algorithm Better than Alpha-Beta ? Yes and No** Artificial Intelligence, Vol. 21, pp 199-219, (1983)

[62] *A.L. Samuel* **Some Studies in Machine Learning Using the game of Checkers. II - Recent Progress** IBM-Journal of Research and Developement, 11, pp 601-617 (1967)

[63] *J. Schaeffer* **The History Heuristic and Alpha-Beta Search Enhancements in Practice** IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 11 No. 11, pp 1203-1212, (1989)

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

[65] *J. Schaeffer* **Comment on 'Distributed Game Tree Search'** ICCA Journal, Vol. 12 No. 4, pp 216-217, (1989)

[66] *C.E. Shannon* **Programming a Computer for Playing Chess** Philosophical Magazine 41, pp 256-275, (1950)

[67] *G. Schrüfer* **Minimax-Suchen Kosten, Qualität und Algorithmen** Doktorarbeit, Universität Braunschweig, Deutschland, (1988)

[68] *J.R. Slagle, J.K. Dixon* **Experiments with some Programs that Search Game Trees** Journal of the ACM, 16, pp 189-207, (1969)

[69] *D.J. Slate, L.R. Atkin* **CHESS 4.5 - The Northwestern University Chess Program** in Chess Skill in Man and Machine, P.W. Frey (Editor), Springer Verlag, pp 82-118, (1977)

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

[71] *A. Szabo, B. Szabo* **The Technology Curve Revisited** ICCA Journal, Vol. 11 No. 1, pp 14-20, (1988)

[72] *K. Thompson* **Computer Chess Strength** Advances in Computer Chess III, M.R.B. Clarke (Editor) Pergamon Press, pp 55-56, (1982)

[73] *A.M. Turing* **Digital Computers Applied to Games** in B.V. Bowden: Faster than Thought: A Symposium on Digital Computing Machines, Pitman, pp 286-310 (1953)

[74] *J. v. Neumann, O. Morgenstern* **Theory of Games and Economic Behavior** Princeton University Press, Princeton, USA, (1944)

[75] *O. Vornberger, B. Monien* **Parallel Alpha-Beta versus Parallel SSS*** Proceedings IFIP Conference on Distributed Processing, North Holland, pp 613-625, (1987)

[76] *T. Warnock, B. Wendroff* **Search Tables in Computer Chess** ICCA Journal, Vol. 11 No. 1, pp 10-13, (1988)

[77] *E. Zermelo* **Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels** 5. Internationaler Mathematikerkongreß, Cambridge, Band 2, pp 510-504, (1912)

[78] *K. Zuse* **Der Computer - Mein Lebenswerk** Springer Verlag, (1984)

Index

$\alpha\beta$-algorithm, 7, 13, 20, 21, 37, 38, 42, 134, 137, 138
9*-ply search, 37, 103, 104, 134

actual variation, 16, 51–53, 59, 61, 63, 67
answer, 71, 73, 109, 113
Artificial Intelligence, 5
aspiration search, 7, 21, 29, 46

B*-algorithm, 144
b/d-uniform, 7, 11, 40, 96, 98
BCH, 22–25, 31, 42, 145
bound
  lower, 22, 28
  upper, 22, 28
Bound and Branch algorithm, 49
bound variable, 17, 63
Bratko Kopec test, 36, 88, 91, 115, 149
breadth first search, 98
brother, 11
brute force search, 34, 40, 41, 144
brute force tree, 34, 35, 37, 40, 68, 95, 96, 98, 106, 138
buffer, 59, 61

cancellation, 55, 63, 81, 123, 124
channel, 88
chess position, 23
clock rate, 35, 36
collision, 22, 24, 25, 31, 41, 42, 71, 112
  expected, 42
communication, 109, 124
communication load, 61, 63, 81
communication process, 62
Conspiracy number search, 144
cutoff, 7, 13, 14, 16, 19, 30, 31, 34, 35, 40, 41, 51, 52, 57, 67, 68, 74, 76, 77, 126
  deep, 7, 19
  direct, 7, 19, 48
cutoff message, 57, 59, 60, 107
cutoff node, 19
cycle, 10
cyclotomic polynomial, 145

DeBruijn network, 43–45, 47, 92, 100, 127–129
degree, 10, 44
demultiplexing, 59, 63
depth, 11, 19, 37, 67
depth first search, 98
deviation, 87–91, 135
Dewey decimal system, 11
diameter, 10, 45
distance, 10
  average, 10, 45, 81, 100, 124

efficiency, 85, 103, 104
ELO, 5, 6, 139–141
exact value, 28
exchange function, 43

fail node, 19, 32
fail search, 19
fail soft, 14, 19
father, 11, 132
finite field, 146
flag, 22
free, 53, 74, 77, 78

game tree, 6, 12
game tree decomposition, 47
game tree search, 12, 46
game tree search problem, 12
generator polynomial, 146
graph, 10
grid, 45–47, 100, 127–129, 133

Hamming distance, 146
hardware scheduler, 35
hash function, 22, 23, 25, 31, 42, 71, 145
heuristics, 21
history
  heuristic, 8, 21, 31, 68, 69, 103, 127

table, 31–33, 69, 83, 89, 94, 125
horizon effect, 34
host processor, 44, 45, 52, 133, 134

indegree, 10
iteration, 29
iterative deepening, 7, 21, 25, 28, 29, 31, 75, 102, 103, 137, 143

key, 22, 24
killer
bonus, 29
heuristic, 7, 21, 29, 41, 68, 103, 127
list, 30, 68, 83, 89, 94, 125, 126
move, 29, 30, 32, 41, 68, 78, 126
update, 30, 126

leaf, 11, 14, 33, 40, 46, 75, 97, 99
length, 10
level, 11
link, 35, 43, 133
load balancing, 75, 130
load distribution, 48, 67, 75, 80, 88, 120, 121, 130–132, 137, 143
global, 79, 83
local, 80, 83

main memory, 35
master, 51, 57, 67, 79, 82, 83, 88
master-slave relationship, 51–53, 55, 57, 79, 123, 125, 130
MAX-node, 12, 14
MIN-node, 12, 16
minimal polynomial, 146
minmax
algorithm, 6, 7
value, 12, 14, 34, 37, 57, 67, 88, 102
move
best, 21, 28, 29, 31, 37, 88, 101
fail, 25, 28, 31, 32
noncapturing, 30
positive capturing, 32, 33, 41, 78
move ordering, 7, 21, 25, 28, 29, 31, 52, 67, 68, 74, 78, 83, 89, 109, 110, 126, 127, 137, 139
multiplexing, 59, 63

neighbor, 81, 120
network, 43
master, 79
network master, 44, 52, 67, 75, 77, 132, 134
newentry, 71
node
actual, 16, 34
free, 53, 63
inner, 11, 51, 99
nondeterminism, 88, 89, 91

Occam, 36
outdegree, 10

parallelism, 34, 48, 51, 52, 68, 74–78, 85, 94, 95, 117–119, 127, 143
parity check, 146, 147
path, 10
performance, 86, 90, 108, 136
loss, 63, 98–100, 107, 108, 115, 116, 128, 135, 138, 143
permanent brain, 101, 102
playing strength, 28, 36, 140
principal variation, 12, 31, 55, 102
heuristic, 21
processor, 35
tree, 131, 132
PV-Split-algorithm, 48, 76, 119, 120
dynamic, 48
E-, 49

quiescence search, 34, 35, 40, 41, 95–98, 106, 139
node, 35

radius constant, 82, 83, 120, 121
refutation variation, 21, 31
representation, 23, 24, 146, 147
request for work, 53, 54, 63, 79, 81–83, 107, 120, 123, 125, 132
research, 14, 19, 21, 40, 42, 55, 66, 67, 86, 107, 137
return, 55, 56, 59, 82, 83, 107, 120, 125
root, 11, 29, 74, 75, 79, 88
root of unity, 145
routing, 59, 63, 73, 81, 86, 107, 109, 113, 114, 116, 124, 128, 130
routing hardware, 105, 130, 133, 143

scalability, 105
Scout-algorithm, 14, 15, 17–21, 37, 38, 42, 134, 137, 138

iterative, 16
search
selective, 34, 35
search depth, 7, 22, 25, 28, 29, 33, 34, 37, 75, 93, 95, 96, 98, 101–103, 107, 134, 138
search depth reduction, 34
search extension, 33, 67
search for work, 53, 63, 67, 75, 80–83, 121, 124
global, 82
local, 79, 81
with N1, 81, 120–123, 125, 127, 136
with N2, 81, 120, 123–125
with N3, 82, 120, 121, 125, 127
search overhead, 52, 74, 76–78, 86, 89, 90, 93–96, 98, 112, 131, 136
negative, 86, 97, 98, 106
search phase, 34
search window, 16, 19, 28, 29, 33, 34, 46, 59, 63, 67, 74
shuffle function, 43
slave, 51, 55, 57, 59, 67
son, 11
speedup, 85, 87, 90–93, 100, 101, 111, 113, 129
calculated, 87, 92, 105
superlinear, 97
SSS*-algorithm, 49
standard version, 92, 110, 117, 120–122, 125, 127, 134–140
static evaluation function, 12, 14, 16, 19, 35, 46, 97
subproblem, 16, 52, 53, 55, 63, 67, 75, 79, 81, 83, 107, 108, 125, 131
subtree, 11, 30, 34, 51, 63, 83
successor, 6, 11, 74
best, 7, 12, 28, 31
left, 11, 41, 74, 75
promising, 78, 127
right, 11, 14, 77, 126
synchronization node, 75, 119, 137, 143

T800, 35, 43, 104, 133
T805, 36, 43, 133
T9000, 104, 105
test position, 28
throughput, 128
time control, 28
time stamp, 22, 25
torus, 45, 47, 100, 127–129
tournament
condition, 100, 101, 105, 112, 138
conditions, 139
game, 28, 101, 106
speed, 104, 110
tournament speed, 135, 138
TR-request, 71, 73, 113
transposition, 70, 109, 139
transposition table, 7, 21, 22, 26–28, 31, 41, 42, 49, 69, 93, 94, 98, 103, 107–109, 116, 134, 135, 138, 139, 143
distributed, 71, 72, 89
global, 69, 70
local, 70
size of the, 110–113, 116
Transputer, 35, 36, 43, 105, 109
TREE, 132
tree, 11
minimal, 28, 39, 40, 49, 74, 76, 96, 98, 107
uniform, 11
two person zero sum game, 6
type of nodes, 39, 94, 139
expected, 76, 77, 96, 117, 126, 127, 139

waiting time, 73, 109, 113, 114, 128, 129, 135
window message, 57, 59, 60, 65, 67, 107
work load, 51, 52, 63, 77, 78, 80, 86, 88, 90, 93, 94, 109, 130, 136

YBWC*, 74–76, 78, 116, 118, 119, 127
YBWC-1, 119
YBWC-1-2*, 76–78, 117–119, 136, 137
Young Brothers Wait Concept, 74

zero window search, 14, 16, 65, 67, 107, 137

List of Figures

2.1 Basic form of the $\alpha\beta$-algorithm . . . 13
2.2 The Scout-algorithm . . . 15
2.3 Iteration for the iterative Scout-algorithm . . . 17
2.4 Iterative Scout-algorithm . . . 18
2.5 Searching a game tree with the Scout-and the $\alpha\beta$-algorithm . . . 20
2.6 Storing entries of the transposition table . . . 26
2.7 Reading entries from the transposition table . . . 27
2.8 Update of the killer lists . . . 30
2.9 Update of the history table . . . 32
2.10 Search phases . . . 34
2.11 Schematic description of a Transputer . . . 36
2.12 Distance from the minimal game tree: Scout, 8-ply search . . . 39

3.1 Debruijn network of dimension four: $DB(4)$ . . . 44
3.2 A 4×4 - torus . . . 45
3.3 A 4×4 - grid . . . 46
3.4 Results of parallel game tree search algorithms . . . 50
3.5 Starting a search for work . . . 53
3.6 Algorithm reacting upon a request . . . 54
3.7 Initializing a subproblem . . . 55
3.8 Algorithm reacting upon a cancellation . . . 55
3.9 Reacting upon a return . . . 56
3.10 Sending of cutoffs and window messages . . . 57
3.11 Update of the search windows . . . 58
3.12 Reacting upon a window resp. cutoff message . . . 60
3.13 Structure of the processes running on a processor . . . 61
3.14 Communication process of the parallel Scout-algorithm . . . 62
3.15 Parallel search of a game tree without any new windows . . . 64
3.16 Window message for a zero window search . . . 65
3.17 Window message for a research . . . 66
3.18 Accesses to the distributed transposition table . . . 72
3.19 Using the waiting time . . . 73
3.20 Synchronization nodes caused by the YBWC . . . 75
3.21 Using parallelism more rashly with the YBWC-1-2 . . . 77
3.22 Layers for local search for work . . . 79
3.23 Local load distribution with $c = 1$ in the $DB(4)$ . . . 80
3.24 Local and global search for work . . . 82

4.1 Deviation of the speedup . . . 87

4.2 Deviation of the work load 88
4.3 Deviation of the search overhead 89
4.4 Speedups for several search depths 92
4.5 Processor work load for several search depths 93
4.6 Search overhead for several search depths 94
4.7 Search overhead in the BFS and QS for several search depths 95
4.8 Negative search overhead in the quiescence search 97
4.9 Loss of performance for several search depths 99
4.10 Saving the smaller search depths 102
4.11 Efficiency as a function of the running time 104
4.12 Speedups for varying sizes of the transposition table 111
4.13 Search overheads for varying sizes of the transposition table 112
4.14 Performance loss as a function of the numbr of processors 115
4.15 Results of the version without N1 122
4.16 Speedups of the $DB(8)$, $T(16,16)$, and the $G(16,16)$ 129
4.17 Work load of the $DB(8)$, $T(16,16)$, and $G(16,16)$ 130
4.18 Search overhead of the $DB(8)$, $T(16,16)$, and $G(16,16)$ 131

List of Tables

2.1 Number of bits protected by a BCH code 24
2.2 Sequential search with the Scout-algorithm 38
2.3 Sequential search with the $\alpha\beta$-algorithm 38
2.4 Cutoffs in the BFT and QST 41
2.5 Accesses to the transposition table 42

3.1 Distances in some DeBruijn, torus and grid networks 47
3.2 Cutoffs by killer moves in % of the number of cutoffs by right sons 69
3.3 Results for local and global transposition tables 70

4.1 Percentage of nodes having the wrong expected type 96
4.2 Number of transposition table messages 98
4.3 Speedups under tournament conditions 101
4.4 Efficiency under tournament conditions 105
4.5 8-ply search with 256 processors 106
4.6 Performance after doubling messages 108
4.7 Amount of communication 109
4.8 Successful accesses for varying transposition table sizes 113
4.9 Search behaviour for varying transposition table sizes 113
4.10 Improvements of the speedup for varying transposition table sizes 113
4.11 Percentage of the number of collisions 114
4.12 Successful transposition table accesses 114
4.13 Sum of the waiting times for TR-requests 114
4.14 Performance loss by routing tasks 116
4.15 Percentage of the nodes with a wrong expected type 117
4.16 Results of the YBWC-1-2 version 118
4.17 Results of the YBWC version 118
4.18 Results of the YBWC-1 version 119
4.19 Results of the version with N1 but without N3 121
4.20 Results of the version without N1 123
4.21 Communication of the version with and without N1 123
4.22 Results of the version with and without N2 124
4.23 Communication of the versions with and without N2 124
4.24 Results of the version without N1 and N3 125
4.25 Cutoffs caused by right successors 126
4.26 Killer updates 126
4.27 Percentage of the nodes with a wrong expected type 127
4.28 Average results of the DeBruijn, torus and grid 129
4.29 Sum of the waiting times for transposition table messages 129

4.30 Static and dynamic load distribution 132

5.1 Sequential Version running on the SC and the GCel (computed) 134
5.2 Standard version running on 512 and 1024 processors 135
5.3 Standard version without N1 running on 512 and 1024 processors 136
5.4 Standard version without zero window search using 512 and 1024 processors . . . 137
5.5 Reduced TR-Accesses with 512 and 1024 processors 138
5.6 Percentage of the nodes having a wrong expected type 139
5.7 Standard version on the $DB(8)$ vs. $\alpha\beta$on the $G(32,32)$ 140

A.1 Powers of α and the corresponding minimal polynomials 146