Computer Chess: Algorithms and Heuristics for a Deep Look into the Future *

Rainer Feldmann

University of Paderborn, Germany

Abstract. In this paper we will describe some of the basic techniques that allow computers to play chess like human grandmasters. In the first part we will give an overview about the sequential algorithms used. In the second part we will describe the parallelization that has been developed by us. The resulting parallel search algorithm has been used successfully in the chess program ZUGZWANG even on massively parallel hardware. In 1992 ZUGZWANG became Vize World Champion at the Computer Chess Championships in Madrid, Spain, running on 1024 processors. Moreover, the parallelization proves to be flexible enough to be applied successfully to the new ZUGZWANG program, although the new program uses a different sequential search algorithm and runs on a completely different hardware.

1 Introduction

The game of chess is one of the most fascinating two-person zero-sum games with complete information. Besides of being one of the oldest games of this kind it is still played by millions of people all over the world. Moreover, during the last years, chess had been put to a worldwide attention due to the interest of the people in superstars like International Grandmaster Kasparov and due to the fact that computers now start to challenge the worlds best chess players [41].

The playing strength of the top human chess players is due to their strategic planning ability, lookahead, intuition and creativity. The playing strength of the computers, however, is mainly based on their speed. It will be interesting to watch the contest between humans and computers in the future.

On the other side, strategic games like chess are considered to be excellent test environments for planning devices [39]. It is great evidence that algorithms, that enable a computer to play chess successfully, can be of use in other domains where strategic planning is required, e.g. in expert systems, motion planning in dynamic environments, or business planning [61].

The modern history of computer chess started with the work of Shannon [56] and Turing [58]. In 1974 a first Computer Chess World Championship was held in Stockholm seeing a win of the soviet program KAISSA. Software as well as hardware improvements significantly increased the playing strength of the

* This work was partly supported by the Leibniz award fund from the DFG under grant Mo 476/99 and by the DFG research project "Selektive Suchverfahren" under grant Mo 285/12-2

computers. Most of the references given at the end of the paper are offsprings of this research in computer chess. Levy and Newborn [40, p 5] mention that the playing strength increased from 1640 ELO points [17] in 1967 to more than 2500 points nowadays. For a detailed description of the history of computer chess we refer to [40].

In this paper we will give an overview about the most important lookahead techniques used in modern chess programs. In the first part we will describe some sequential search algorithms, that enable computers to choose their move based upon a deep look into the future. We will describe the standard enhancements used to speed up the search. Then we will present two selective search algorithms allowing even further lookahead along promising lines of play at the risk of overlooking chances or threats on other lines. The first one is the Null Move Search Algorithm [8, 29, 16] that is widely used in state-of-the-art chess programs today. The second one, the so called Fail High Reduction Algorithm, is an alternative for programs using a sophisticated static evaluation function [25]. We developed it quite recently and showed its superiority to the standard algorithm in a sequence of tests. It is now used in ZUGZWANG and at least one commercially available chess program.

In the second part we will describe a parallel approach to the search algorithm allowing an even deeper lookahead. The resulting distributed algorithm has been used successfully in the chess program ZUGZWANG even on massively parallel hardware [20, 21, 22, 23, 24]. In 1992 ZUGZWANG became Vize World Champion at the Computer Chess Championships in Madrid, Spain, running on 1024 T805 processors. Moreover, the parallelization proves to be flexible enough to be applied successfully to the new ZUGZWANG program, although the new program uses a different sequential search algorithm and runs on a completely different hardware. This new program recently played successfully at the 12th AEGON Man vs. Machine tournament in Den Haag, The Netherland, running on 40 M604 processors. Our parallelization is the first one that has been applied efficiently to massively parallel hardware.

1.1 The Problem

Informally spoken, given a chess position S we are interested in the best move for the side to move in S. Zermelo [66] formalized the notion of a theoretical win, loss or draw in two-person zero-sum games. From this the following recursive minmax function $\mathcal{F}$ can be deviated: For a terminal position t

$$\mathcal{F}(t) := \begin{cases} 1, \text{ if } t \text{ is won for the side to move in } t \\ 0, \text{ if } t \text{ is a draw} \\ -1, \text{ if } t \text{ is lost for the side to move in } t \end{cases}$$

The value of a nonterminal position v with children $v_1, \ldots v_w$ is then given as

$$\mathcal{F}(v) := \max\{-\mathcal{F}(v_1), \ldots, -\mathcal{F}(v_w)\}.$$

If the game does not contain infinitely long sequences of moves, $\mathcal{F}(v)$ can be computed for any position v. This is done by building a search tree with root

v and the leaves being the terminal positions reachable from v. $\mathcal{F}$ is applied to the leaves and then backed up to the root. A best move in v is then obtained by choosing an edge to one of the children v_i with $\mathcal{F}(v_i) = \mathcal{F}(v)$.

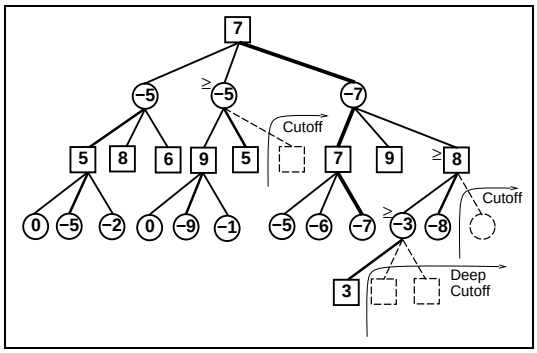
Unfortunately, due to time constraints, in strategic games it is not possible to generate the complete search tree, i.e. the tree cannot be extended to the terminal positions. Therefore, instead of $\mathcal{F}$, a static evaluation function f is computed mapping positions to numbers. $f(v)$ is a heuristic estimation of the chances to win for the side to move in v. f is then used as an approximation of $\mathcal{F}$. A simple recursive algorithm, cutting the search tree at depth d and applying f at the leaves is used to compute the minmax value $\mathcal{F}_{f,d}$ of the root. Then a move is chosen as if $\mathcal{F}$ was computed. We can now define the problem somewhat more precisely as

> Given a position v, a search depth d and a static evaluation function f.
> Compute $\mathcal{F}_{f,d}(v)$.

Although it can be shown in theoretical game tree models that searching deeper may result in an even worse decision at the root of the tree [3, 52, 57], this has barely been observed in practice. For the game of chess an increase of 100-250 ELO points has been observed by searching one ply deeper [59, 60, 48]. Therefore it is essential to search as deep as possible while paying attention to the time constraints posed to the program by the chess clocks.

2 Sequential Game Tree Search

The recursive algorithm to compute the minmax value of the root visits every leaf of the search tree. But often one can do better as is shown in the figure below. Suppose, some game tree search algorithm already computed the value -5 for the leftmost child of the root. Then the algorithm starts its search below the second child. After it sees the value 5 a cutoff can be performed pruning the rest of the subtree, since the second child can never improve on the value of the first child.



The situation is similar below the node marked -3 : If the third child of the root will improve on the result found below the first one, this cannot happen by choosing the move to -3 at the node marked 8, since the result at the root would be ≤ 3. A deep cutoff occurs. The rightmost child of the node marked 8 can be pruned by a cutoff because of the values 7 and -8 already computed. Note that instead of an exact minmax value only lower bounds are computed for the nodes where a cutoff occurs. An upper bound is computed for a node if for all its children a lower bound has been computed.

2.1 $\alpha\beta$-Search and $\alpha\beta$-Enhancements

The $\alpha\beta$-algorithm shown to the right uses the cut-off technique to speed up the search. Knuth and Moore [36] analyzed the $\alpha\beta$-algorithm started with alphaBeta($root, -\infty, \infty$) and showed that in a uniform tree of width w and depth d it visits $\lceil w^{d/2} \rceil + \lfloor w^{d/2} \rfloor$-1 leaves in the best case but all w^d leaves in the worst case.

```
int alphaBeta(node v, int α, β) {
    int val;
    generate all successors v_1, ..., v_w of v;
    if v is a leaf return(f(v));
    val := −∞;
    for i := 1 to w do {
        val := max(val,−alphaBeta(v_i, −β, −α);
        if val ≥ β return(val); /* cutoff */
        α := max(α, val);
    }
    return(val);
}
```

Moreover, they showed that the best case is achieved, if the best move is considered first at any inner node of the tree. From then on, research in the field of game tree search algorithms concentrated on the question on how to improve on the move ordering to get full use of the efficiency of the $\alpha\beta$-algorithm. In the following we will describe some of the enhancements.

The Transposition Table The $\alpha\beta$-Algorithm is a tree searching algorithm. However, due to transpositions of moves, the same position may be reached at different nodes in the tree. In this case the same subtree would have to be searched twice. Since in general it is not possible to store the whole search tree in the main memory, a hash table is used to store the results already known. An entry of the hash table consists of the components: $(lock, d, x, flag, move)$. The lock typically consists of 64 bits. It can be computed efficiently in the same way as the hash function used to access the hash table [67], and in practice suffices to identify the chess position uniquely [64]. The component d gives the search depth, x is the value computed. $flag$ indicates whether x is an upper bound, a lower bound or an exact value, and $move$ is the best move obtained.

When a subtree below a chess position v has been evaluated with search window $[\alpha, \beta]$ and a result x has been computed, an entry is written to the hash table. x is an upper bound, iff $x \leq \alpha$, a lower bound iff $x \geq \beta$ and an exact value iff $\alpha < x < \beta$.

Before the search starts at a chess position v, a hash table lookup is done for v. If there is an entry $(lock, d, x, flag, move)$ for v and d is large enough, the search may be speeded up by immediately returning x, if x is an exact value, increasing α if x is a lower bound, or decreasing β if x is an upper bound. If d is not large enough the move from the hash table is tried first in the search, since there is good hope that the move found as a best one in a shallow search remains the best one in the deeper search.

For a detailed description of the use of transposition tables see [44]. Collision resolution strategies have been investigated in [13, 10].

Other Enhancements Several other $\alpha\beta$-enhancements are widely used:

The *iterative deepening* technique is common in any chess program. Instead of searching the root of the tree directly to a search depth d, it is searched successively to search depths $1, 2, \ldots, d$. Many of the results of the shallow searches are stored in the transposition table and help to improve the move ordering for the deeper searches. Moreover, in real time applications like chess, the iterative search guarantees, that there is a best move available even after completion of the 1-ply search. Thus, the process may be stopped at any time and the best move from the last iteration may be delivered as the move to be played.

The *aspiration search* starts the $\alpha\beta$-search at the root by the function call alphaBeta$(root, \alpha_0, \beta_0)$ for some $-\infty < \alpha_0 < \beta_0 < \infty$. If the search terminates with a result $\alpha_0 < x < \beta_0$, x is correctly determined and some effort may have been saved by the use of the narrow aspiration window $[\alpha_0, \beta_0]$. Otherwise, if $x \leq \alpha_0$ or $x \geq \beta_0$, a re-search with the full window $[-\infty, x]$, or $[x, \infty]$ resp., has to be performed. α_0 and β_0 are usually estimated using the result of the preceding iteration.

Many heuristics to improve the move ordering have been developed. The *killer heuristic* stores a number of moves that led to cutoffs in subtrees already searched [1]. Moves that produce a cutoff in level d are stored in a list for level d. These lists usually have only two entries. A benefit is stored for each move and new moves overwrite the one with the smaller benefit. Moves from these killer lists are tried early in the search, if they are found to be legal moves.

The *history heuristic* [54] increments a bonus in a table for moves, that become best or lead to cutoffs in subtrees previously searched. These bonuses are then used to sort the moves at a chess position, if no other information is available.

Bad moves are often refuted by the same good move, no matter where they are applied in the tree. The *countermove heuristic* stores such refuting moves and provides them for early use in the search [62].

The total move ordering then is as follows: First the move from the transposition table is searched. Afterwards positive captures, i.e. captures that look at least equal at first glance, are tried in decreasing order of their potential benefit. If a countermove is available it is tried after the positive captures. Then, the moves from the killer list are tried in decreasing order of their benefit. Afterwards all the remaining moves are searched in decreasing order of their history values.

2.2 The Negascout Algorithm

Using the above described heuristics the move ordering usually is very good. Therefore, the first move considered in the search is very often found to be the best move, or at least suffices to produce a cutoff. This fact is used by the Negascout algorithm of Reinefeld [53] shown in figure 1.

The main idea of the algorithm is that it computes the first successor of a position just as the $\alpha\beta$-algorithm would do, but then tries to show that the other successors are inferior to the first one. This is done by searching them with an

```
int Negascout(node v, int α, β, d) {
    int x, low, val, high;
    if d > 0 generate all successors v_1, ..., v_w of v;
    if v is a leaf return(f(v));                          /* static evaluation */
    low := α; high := β; val := −∞;
    for i := 1 to w do {
        x := −Negascout(v_i, −high, −low, d − 1);         /* i > 1 : null window */
        if x > low and x < β and i > 1
            x := −Negascout(v_i, −β, −x, d − 1);          /* re-search */
        val := max(val, x);
        if val ≥ β return(val);                           /* cutoff */
        low := max(low, x); high := low + 1;
    }
    return(val);
}
```

Fig. 1. The Negascout algorithm

artificial search window $[-high, -low]$ of width zero, i.e. $high - low = 1$. Only if this search shows that the corresponding node is superior to the best one found so far, a re-search is done with the full window $[-\beta, -x]$. The Negascout algorithm has the same best case behaviour as the $\alpha\beta$-algorithm, i.e. in a uniform tree of width w and depth d it searches exactly $\lceil w^{d/2} \rceil + \lfloor w^{d/2} \rfloor - 1$ leaves. In the worst case all w^d leaves have to be visited, many of them even twice. In practice the Negascout algorithm appears to be faster than the pure $\alpha\beta$-algorithm [53], and the average running time is very close to the best case running time due to the use of the $\alpha\beta$-enhancements . But the depth of the search is still limited by the exponential running time of the algorithm.

Therefore, many heuristics have been developed to guide the lookahead of the game tree search algorithms into those branches of the tree, which are relevant for the decision at the root, in order to search them deeper than the irrelevant lines. Many of these heuristics are domain dependent, like e.g. check evasion extensions. An overview and an empirical comparison is given in [6]. There has also been some research on developing domain independent heuristics to guide the search in game trees [11, 47, 5, 8, 16, 15, 42, 25]. The effects of some combinations are studied in [9]. The selective search heuristic most widely used today is described in the next section.

2.3 Null Move Search

In the late eighties Beal [8] again looked at his idea of the Null Move Heuristic. It is based upon the observation that in many games and game like applications for the side to move there is almost always a better alternative than doing nothing

```
const maxGain = 1.5;     /* pawn units */
const nmReduce = 2;      /* depth reduction for null move searches */

int NmNegascout(node v, int α, β, d, bool nullmove) {
    int x, y, low, val, high;
    if d > 0 generate all successors v_1, ..., v_w of v;
    if v is a leaf return(f(v));                          /* static evaluation */

    /* null move search */
    if (nullmove and d > 1 and x > α−maxGain and not in check) {
        v' := switchSide(v);
        y := −NmNegascout(v',−β,−α,d − nmReduce − 1,false);
        if y ≥ β return(y);                               /* null move cutoff */
        α := max(α, y);
    }

    /* regular Negascout algorithm */
    low := α; high := β; val := −∞;
    for i := 1 to w do {
        x := −NmNegascout(v_i, −high, −low, d − 1, true);
        if x > low and x < β and i > 1
            x := −NmNegascout(v_i, −β, −x, d − 1, true);  /* re-search at v_i */
        val := max(val, x);
        if val ≥ β return(val);                           /* cutoff */
        low := max(low, x); high := low + 1;
    };
    return(val);
}
```

Fig. 2. The Negascout Algorithm with Null Move Searches

(the null move). We call this the Null Move Observation (NMO). Beal proposed to use cutoffs found by shallow searches after a null move and thus avoid to compute the cutoffs in deep searches after real moves. A first implementation is presented in [29]. Donninger [16] proposed a null move search algorithm similar to the one shown in figure 2.

The main drawback of the Null Move Search is that it fails in zugzwang positions. In these positions the NMO is not valid, causing chess programs to fail heavily. Therefore, in endgames, chess programs usually switch off the Null Move Search or try to verify the NMO by some extra searches. Another drawback of the Null Move Search is that extra searches are carried out in order to establish the cutoffs by the null move. If the search after a null move fails to produce a cutoff, extra effort has been spent for nothing. It has been observed by many programmers that the Null Move Search improves the computers play in tacti-

```
int FhrNegascout(node v, int α, β, d) {
    int x, low, val, high, eval, δ, t;
    eval := f(v);                                   /* static evaluation */
    δ := d;                                         /* save search depth */
    if (eval ≥ β and α = β − 1) δ := δ − 1;
    if δ > 0 generate all successors v_1, ..., v_w of v;
    if v is a leaf return(eval);
    low := α; high := β; val := −∞;
    for i := 1 to w do {
        x := −FhrNegascout(v_i, −high, −low, δ − 1);   /* i > 1 : null window */
        if x > low and x < β and i > 1
            x := −FhrNegascout(v_i, −β, −x, d − 1);    /* re-search: no Fhr */
        val := max(val, x);
        if val ≥ β return(val);                     /* cutoff */
        low := max(low, x); high := low + 1;
    };
    return(val);
};
```

Fig. 3. The FhrNegascout Algorithm

cal positions but sometimes performs poorly in positional ones. These effects, however, are hard to quantify and therefore no publications investigating the behavior of the algorithm are available. Nevertheless, the Null Move Search is used widely by professional as well as amateur chess programmers.

2.4 Fail High Reductions

The algorithm presented in this section tries to overcome the drawbacks of the Null Move Search algorithm. Again, it is based upon the Nmo. It uses the static evaluation function f to compute a value even for inner nodes of the tree. If this static value indicates, that the side to move is already better than β, a cutoff is expected and the remaining search depth is reduced by one. These depth reductions, however, are done only in the searches with an artificial zero width window.

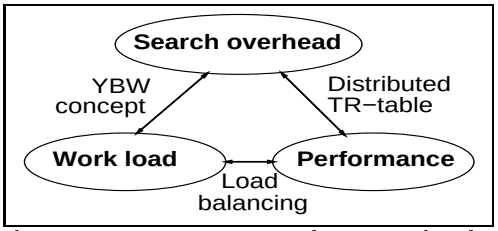
A slightly simplified form of the algorithm is given in figure 3. For details we refer to [25].

The Fhr-algorithm is used in Zugzwang since 1995, and at least one commercially available chess program [34]. In [25] we showed in three different tests that the Fhr-algorithm is superior to the regular $\alpha\beta$-algorithm with statistical significance. All three tests indicate that the improvement in playing strength is about 120 ELO points. Recently, Ulf Lorenz implemented both the Fhr-algorithm and the Null Move Search in his chess program Conny. The tests

performed with these implementations indicate that the playing strengths of the FHR-algorithm and the Null Move Search are equal [43].

3 Parallel Game Tree Search

In this part of the paper we will sketch the concept of our parallelization of the game tree search. A detailed description of an implementation on a network of 1024 T805 processors can be found in [23].



The only hope to get scalable parallelism in a game tree search is to split the game tree and search the resulting subtrees in parallel. For many years it has been an open problem how to parallelize the game tree search and a lot of research has been done in this field [7, 27, 2, 28, 44, 32, 49, 63, 26, 31, 46, 50, 51, 18, 33, 35, 55, 30, 37, 19, 21, 22, 23, 4, 38, 65].

Three difficulties have to be dealt with when searching subtrees in parallel. They are shown in the figure above as a triangle.

- **Processor work load:** Since the game tree search routines use cutoffs during the search, the size of a subtree is not known in advance. Thus, a dynamic load balancing mechanism is required.
- **Search overhead:** Many subtrees are not visited by the sequential search algorithm. The information required to prune a subtree has been found by searching the subtrees to the left. In a parallel system, the information about the left subtrees may not be available in time or may be distributed over the processors. In both cases the parallel algorithm will miss cutoffs and will search more nodes than the sequential would have done. A method to reduce this search overhead is required.
- **Performance:** For the dynamic load balancing and the information sharing, the processors have to communicate. In a message passing system like the ones used, this is done by exchanging messages. The sending and receiving of messages costs CPU cycles, and therefore processors work at a decreased performance in a parallel environment.

The edges between the nodes in the triangle above indicate the tradeoffs between the three losses:

- Increasing the number of messages for the dynamic load balancing may improve the average work load but at the costs of a decrease of the performance. Reducing the numbers of messages for the load balancing will result in a decrease of the average work load.
- Increasing the number of messages for information sharing may improve the efficiency of the parallel search but again at the costs of a decrease of the performance. On the other hand, the search overhead will increase if important information is not available.

– Applying parallelism rashly will result in a perfectly balanced work load, but most of the processors will search subtrees that are not visited by the sequential algorithm. A careful use of parallelism is necessary reducing the average work load of the processors.

3.1 Parallel Search in ZUGZWANG

In this section we will describe the main concept of the parallel algorithm used in ZUGZWANG since 1990. The concept is designed for the use on massively parallel distributed systems, i.e. systems without shared memory. The communication is realized by message passing. In the years from 1990 - 1994 we ran ZUGZWANG on a system based upon T800/805 Transputers, with up to 1024 processors. Today ZUGZWANG is running on an M604 based system with 40 processors. The adaptation of the parallelization to the new hardware or software required only minor changes in the implementation, showing the flexibility of our approach.

The Algorithm Our parallelization of the game tree search is based upon a decomposition of the tree to be searched, and on a parallel evaluation of the resulting subproblems. The sequential algorithm is a depth first search algorithm. It visits the nodes of the tree from left to right. To do so, a current variation is stored in a stack. 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 current variation for parallel evaluation as possible. By this, several processors may start a tree search on disjoint subtrees of the whole game tree.

For a formal definition of the algorithm we refer to [23]. Informally the algorithm can be described as follows:

– Initially all processors but the master are idle. The master gets the root of the search tree, and starts a sequential search generating a current variation.
– Idle processors send a request for work to any other processor. When a processor gets a request for work it looks for free subproblems to the right of its current variation. If one is found it is sent to the requesting processor. Otherwise the request is forwarded to another processor.
– After an idle processor has sent its request for work it waits for a new subproblem.
– After a processor has solved a subproblem it sends a return message to its master, containing the result computed for the subproblem. The master, upon receiving the return message, updates the bounds on its current variation.
– Whenever the bounds of the current variation at a processor are changed, either by computation or by a message, the processor immediately informs all slaves using the corresponding bounds about the update by sending a new search window. The new window may be empty causing the slaves to cancel their subproblems.

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.
2. *Distribution of information in the distributed algorithm:* The sequential algorithm allows many cutoffs. 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 nodes and by the knowledge about the current bounds. The sequential search algorithm uses the information computed during the search of the left parts of the tree to improve the ordering of the successors of inner 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 is obtained by communication between the processors.
3. *Restricting the use of parallelism to avoid search overhead:* The 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 will visit only nodes, which the sequential one does not search. In situations like this, one will prefer an idle processor to remain idle for a short time and then to start working on a subproblem which is useful to evaluate, rather than to start working on a subproblem immediately, which is irrelevant with high probability.

The Load Balancing Algorithm For the load balancing we have chosen an adaptive algorithm:

- Local balancing: A processor chooses a neighbor in the processor network as a target for its first request for work.
- Medium range balancing: A processor that gets a request for work from its neighbor, but is not able to send a subproblem, forwards the request to one of its remaining neighbors, if the request has not been forwarded too far. In this case the request is sent back to the origin.
- Global balancing: Any return message is interpreted as a request and is handled in the same way.

The local and medium range load balancing is cheap in terms of communication costs, since messages are sent only to neighbors in the network, but not sufficient to balance large networks. The global balancing has three effects: As the search proceeds, the request are spread all over the network. Second, slaves tend to stay at the same master and therefore local move ordering heuristics tend to work better. Last but not least it saves a request, since the return message has to be sent anyway.

The Distributed Transposition Table Marsland and Popowich [45] showed that local transposition tables do not work well in a parallel system, even if the number of processors is small, since processors can access only the information computed by themselves. Therefore we have implemented the transposition table as a large distributed table, and organize the access to the table by routing messages. Although this is quite communication intensive, it has the advantage that the parallel system has access to a much larger hash table than the sequential one, improving its search behaviour.

In an N-processor system, the value $h(v)$ of the hash function for a chess position v is interpreted as the concatenation of the lock of v, a processor number $p(v) := h(v) \bmod N$ and a local index $i(v) := \lfloor \frac{h(v)}{N} \rfloor$. Therefore, a processor that wants to access the table for a position v sends a read request to processor $p(v)$ containing the local index $i(v)$ and the lock. A processor, upon getting a read request for a local index i, compares the lock of the corresponding entry with the lock of the message. If both are equal the entry from the table is sent back.

A critical delay may occur when a processor wants to read a remote table entry for a position v. Sometimes a valid entry is found, and the move contained in this entry has to be searched first below v. But this means, that the processor has to wait until the answer arrives. In large networks these waiting times may last longer than the search below v. Therefore, after sending a read request the processor continues the search below v immediately. If an answer from the remote table arrives, the entry is checked and the search is restarted if the first move is not the one proposed by the transposition table entry.

In the current version a processor requests the global transposition table for a node v only if v is the root or if an entry has been found for the father of v. This keeps the number of messages necessary to access the table reasonably small. The result for a node v is written into the global table only if the remaining search depth below v is larger than 1.

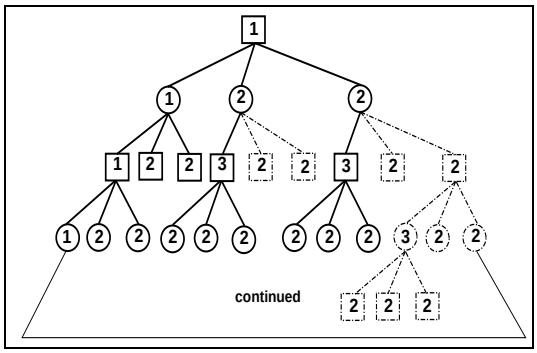
The Young Brothers Wait Concept In this section we describe how to restrict parallelism to reduce the amount of work done in irrelevant subtrees, but get enough parallelism out of the application to achieve a good processor work load. The problem is that the sequential version determines the shape of the tree on the basis of the search windows. These are narrowed as the search proceeds. Therefore the shape of the tree is not known in advance. We know on the other side, that the tree, that is searched by the sequential version, usually is very close to the minimal game tree and the shape of the minimal game tree can be determined exactly. This has been done by Knuth and Moore [36] by recursively assigning types to the nodes of the tree as follows:

- The root gets type 1.
- The first child of a type-1 node gets type 1. The other children get type 2.
- The first child of a type-2 node gets type 3.
- All children of a type-3 node get type 2.

By this a type is assigned to exactly the nodes of the minimal tree. If only the minimal game tree is searched a cutoff happens exactly at the nodes of type 2. Hsu [30] implicitly extended the assignment of types to the whole tree by adding the rule

– The right children of a type-2 node get type 2.

The new assignment is shown in the figure below. We use these extended node types to restrict the use of parallelism in the distributed search by assuming that a cutoff will happen at type-2 nodes. As a result the use of parallelism is delayed at type-1 nodes, even further delayed at type-2 nodes but enabled at type-3 nodes immediately after they have been generated. This results in the so called *Young Brothers Wait Concept*:



– The parallel search below right children of a type-1 node is allowed only if the search below the leftmost child is complete.
– The parallel search below right children of a type-2 node is allowed only if the searches below all promising children are complete. Here, promising children are the leftmost one, those reached by a move proposed by the transposition table, the killer heuristic, or any positive capturing move.
– The parallel search below a type-3 node may be started immediately after its generation.

Parallelism is used carefully at type-2 nodes. Since we expect a cutoff to happen at these nodes, we evaluate the promising moves sequentially, one by one. Only if these promising moves fails to produce a cutoff the remaining children may be searched in parallel. By this the parallel search algorithm finds a lot of cutoffs found by the sequential one without searching irrelevant subtrees. At type-1 nodes the use of parallelism is delayed until the first child is completely evaluated, since the result from this search is used as a bound for the search below the right children. At type-3 nodes we do not expect a cutoff and therefore allow the parallel search of the children as soon as possible. The Young Brothers Wait Concept is a heuristic to guide the use of parallelism. In case of a mistake at a type-2 node, the search is sequentialized below this node and thus the processor work load may be decreased. In case of a failure at a type-3 node, processors search irrelevant subtrees in parallel increasing the search overhead. However, in [23] we present statistical data showing that this heuristic works very well.

Results In [23] we describe experiments showing the efficiency of the proposed approach for a parallel search of the Negascout algorithm using up to 1024 T805

Transputers. The Transputers were connected as a DeBruijn network (≤ 256 processors) or as a 2-dimensional grid (> 256 processors). Here, we give only a small summary:

Proc.	1	8	16	32	64	128	256	512	1024
depth	5	6	7	7	7	8	8	9	9
time	163	125	342	193	113	320	193	384	264
SPE	**1.00**	**6.13**	**12.33**	**21.83**	**37.34**	**86.03**	**142.32**	**237.03**	**344.49**
SO %	0.00	15.81	14.04	19.79	25.10	10.05	17.30	39.82	55.55
LOAD %	100.00	90.10	89.36	85.48	77.86	81.98	74.13	74.56	62.67
PERF %	100.00	98.56	98.68	95.61	93.78	90.22	87.91	86.96	83.61

Fig. 4. Results of ZUGZWANG on the Transputer networks

Figure 4 contains the speedup measurements on the basis of a set of 24 chess positions [12]. The speedup (SPE) is defined as the running time of the sequential algorithm divided by the running time of the parallel algorithm, both with the same depth parameter. The search overhead (SO), processor work load (LOAD) and performance (PERF) are given in percent compared to the sequential version with the same depth parameter.

For each number of processors we have chosen the search depth such that the average running time is close to the running time under tournament conditions ($150-300$ seconds). The single processor version needs 163 seconds on the average for a 5-ply search while the 1024 processor is able to complete a 9-ply search at 264 seconds on the average. For the tests on the two grid networks we had to reduce the number of accesses to the distributed transposition table since otherwise the performance would have been too bad. As a consequence the search overhead increased.

The distributed search algorithm proposed in this paper was the heart of the chess program ZUGZWANG at the Computer Chess Championships in Madrid, Spain, 1992. ZUGZWANG ran on 1024 T805 processors and finished second.

In the meantime ZUGZWANG uses the FHR-algorithm and runs on a 40 processor machine based upon M604 processors. Since the FHR-algorithm uses the search windows to determine the shape of the tree, the sequential and the parallel search algorithm may not compute the same result, even if the search parameters are the same. Therefore it is very hard to measure the speedup. Different metrics have to be developed to determine the efficiency of the parallel algorithm. This work is still in progress. Here we present only preliminary data, showing that the approach can be used to parallelize the search in very irregular trees, as generated by the Fail High Reductions algorithm. In the data of figure 5 we substituted the term "speedup" with "acceleration".

The loss of performance of about 25% is mainly due to the large number of messages for the distributed transposition table and to the worse ratio of com-

Proc.	1	4	8	16	32	40	40
depth	9	10	10	11	11	11	12
time	255	360	215	408	205	161	475
ACC	**1.00**	**2.49**	**4.16**	**7.02**	**13.96**	**17.71**	**22.58**
SO %	0.00	8.14	19.90	43.94	22.28	16.46	0.95
LOAD %	100.00	82.23	79.19	79.38	68.99	67.94	73.48
PERF %	100.00	81.70	78.73	79.52	77.35	75.88	77.57

Fig. 5. Results of ZUGZWANG on the M604 based systems

munication speed to computational speed of the M604 based machine compared to the T805 based machines.

At the 12th Aegon Man vs. Machine Tournament in Den Haag, The Netherlands 1997, ZUGZWANG ran on a 44 M604 processor machine and achieved a tournament performance rating of more than 2600 ELO points showing that ZUGZWANG belongs to the worlds top chess programs.

4 Conclusions

First, we presented some sequential algorithms used in modern chess programs. We showed how heuristics like the transposition table are used to speed up the sequential search. An even deeper lookahead is obtained by using either the Null Move Search or the Fail High Reductions algorithm. In the second part of the paper we presented our approach to a distributed game tree search algorithm. This approach has been used successfully on massively parallel systems like a 1024-processor Transputer network. Basically the same approach is used for the parallelization of the Fail High Reductions algorithm. Although the trees searched by this algorithm are different in structure, the distributed search algorithm again shows promising results.

Nevertheless, more locality has to be exploited in the distributed algorithm, especially in using the transposition table in a parallel environment. New methods to save a large portion of the store and read messages without increasing the search overhead have to be developed. Another open problem is a problem of testing: How can we compare the sequential and the parallel program if for fixed search parameters they do not deliver the same results ?

Acknowledgements I would like to thank my supervisor Burkhard Monien, who enthusiastically supported the computer chess projects at the University of Paderborn since the early beginning. Many of the results presented here have been joined work with Peter Mysliwietz, who left the University of Paderborn in autumn 1996.

References

1. *S.G. Akl, M. Newborn* **The principal continuation and the killer heuristic** ACM Annual Conference, pp 466-473, 1977
2. *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, 14(2), pp 192-203, 1982
3. *I. Althöfer* **On Pathology in Game Tree and Other Recursion Tree Models** Habilitation thesis, University of Bielefeld, Germany, 1991
4. *I. Althöfer* **A Parallel Game Tree Search Algorithm with a Linear Speedup** Journal of Algorithms, 15(2), pp 175-198, 1993
5. *T.S. Anantharaman, M. Campbell, F.H. Hsu* **Singular Extensions: Adding Selectivity to Brute-Force Searching** ICCA Journal, 11(4), pp 135–143, 1988.
6. *T.S. Anantharaman* **Extension Heuristics** ICCA Journal, 14(2), pp 47–63, 1991.
7. *G. Baudet* **The Design and Analysis of Algorithms for Asynchronous Multiprocessors** Phd thesis, Carnegie-Mellon University, Pittsburgh, USA, 1978
8. *D.F. Beal* **Experiments with the Null Move** Advances in Computer Chess V, D.F. Beal (ed.), pp 65–79, 1989.
9. *D.F. Beal, M.C. Smith* **Quantification of Search-Extension Benefits** ICCA Journal, 18(4), pp 205–218, 1995.
10. *D.F. Beal, M.C. Smith* **Multiple Probes of Transposition Tables** ICCA Journal, 19(4), pp 227–233, 1995.
11. *H.J. Berliner* **The B* Tree Search Algorithm: A Best-First Proof Procedure** Artificial Intelligence, 12, pp 23-40, 1979
12. *I. Bratko, D. Kopec* **A Test for Comparison of Human and Computer Performance in Chess** Advances in Computer Chess III, M.R.B. Clarke (ed.), Pergamon Press, pp 31-56, 1982
13. *D.M. Breuker, J.W.H.M. Uiterwijk, H.J. van den Herik* **Replacement Schemes for Transposition Tables** ICCA Journal, 17(4), pp 183–193, 1994.
14. *D.M. Breuker, J.W.H.M. Uiterwijk, H.J. van den Herik* **Replacement Schemes and Two-Level Tables** ICCA Journal, 19(3), pp 175–180, 1996
15. *M. Buro* **ProbCut: An Effective Selective Extension of the $\alpha\beta$ - Algorithm** ICCA Journal, 18(2), pp 71–76, 1995
16. *C. Donninger* **Null Move and Deep Search** ICCA Journal, 16(3), pp 137–143, 1993.
17. *A.E. Elo* **The Rating of Chessplayers, Past and Present** Arco Publishing, New York, 1978
18. *R. Feldmann, B. Monien, P. Mysliwietz, O. Vornberger* **Distributed Game-Tree Search** ICCA Journal, 12(2), pp 65-73, 1989
19. *R. Feldmann, B. Monien, P. Mysliwietz, O. Vornberger* **Distributed Game Tree Search** Parallel Algorithms for Machine Intelligence and Vision, V. Kumar, L.N. Kanal, P.S. Gopalakrishnan (eds.), Springer, pp 66-101, 1990
20. *R. Feldmann, B. Monien, P. Mysliwietz* **A Fully Distributed Chess Program** Advances in Computer Chess VI, D.F. Beal (ed.), pp 1-27, 1991
21. *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 (eds.), pp 72-87, 1992
22. *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 (eds.), pp 270-288, 1991
23. *R. Feldmann* **Game Tree Search on Massively Parallel Systems** Doctoral thesis, University of Paderborn, Germany, 1993
24. *R. Feldmann, P. Mysliwietz, B. Monien* **Studying Overheads in Massively Parallel MIN/MAX-Tree Evaluation** Proceedings of SPAA'94, pp. 94-103, 1994
25. *R. Feldmann* **Fail High Reductions** Advances in Computer Chess VIII, H.J. van den Herik, J.W.H.M. Uiterwijk (eds.), University of Maastrich, The Netherlands, pp 111-127, 1997
26. *C. Ferguson, R.E. Korf* **Distributed Tree Search and its Application to Alpha-Beta Pruning** Proceedings AAAI-88, 7th National Conference on Artificial Intelligence, 2, pp 128-132, 1988
27. *R.A. Finkel, J.P. Fishburn* **Parallel Alpha-Beta Search on Arachne** IEEE International Conference on Parallel Processing, pp 235-243, 1980
28. *R.A. Finkel, J.P. Fishburn* **Parallelism in Alpha-Beta Search** Artificial Intelligence, 19, pp 89-106, 1982
29. *G. Goetsch, M.S. Campbell* **Experiments with the Null-Move Heuristic** Computers, Chess, and Cognition, T.A. Marsland and J. Schaeffer (eds.), Springer, pp 159-168, 1990
30. *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
31. *M. M. Huntbach, F. W. Burton* **Alpha - Beta Search on Virtual Tree Machines** Information Sciences, 44, pp 3-17, 1988
32. *R.M. Hyatt* **Parallel Chess on the Cray X-MP/48** ICCA Journal, 8(2), pp 90-99, 1985
33. *R.M. Hyatt, B.W. Suter, H.L. Nelson* **A parallel alpha/beta tree searching algorithm** Parallel Computing, No. 10, pp 299-308, 1989
34. *G. Isenberg* (author of ISICHESS), personal communication, February 1997
35. *R. M. Karp, Y. Zhang* **On Parallel Evaluation of Game Trees** Proceedings of SPAA'89, pp 409-420, 1989
36. *D.E. Knuth, R.W. Moore* **An Analysis of Alpha - Beta Pruning** Artificial Intelligence, 6, pp 293-326, 1975
37. *H.-J. Kraas* **Zur Parallelisierung des SSS*-Algorithmus** Doctoral thesis, University of Braunschweig, Germany, 1990
38. *B.C. Kuszmaul* **The Startech Massively-Parallel Chess Program** ICCA Journal, 18(1), pp 3-19, 1995
39. *R. Levinson, F-h. Hsu, J Schaeffer, T.A. Marsland, D.E. Wilkins* **The Role of Chess in Artificial Intelligence Research** Proc. of the 12th IJCAI, Morgan Kaufman Publishers, pp 557-562, 1991
40. *D. Levy, M. Newborn* **How Computers Play Chess** Computer Science Press, 1991
41. *D. Levy* **Crystal Balls: The Meta-Science of Prediction in Computer Chess** ICCA Journal, 20(2), pp 71-78, 1997
42. *U. Lorenz, V. Rottmann, R. Feldmann, P. Mysliwietz* **Controlled Conspiracy-Number Search** ICCA Journal, 18(3), pp 135-147, 1997
43. *U. Lorenz* (author of ULYSSES, CHEIRON, CONNY), personal communication, 1997
44. *T.A. Marsland, M.S. Campbell* **Parallel Search of Strongly Ordered Game Trees** Computing Surveys, 14(4), pp 533-551, 1982

45. *T.A. Marsland, F. Popowich* **Parallel Game Tree Search** IEEE Transactions on Pattern Analysis and Machine Intelligence, 7(4), pp 442-452, 1985
46. *T.A. Marsland, M. Olafsson, J. Schaeffer* **Multiprocessor Tree-Search Experiments** Advances in Computer Chess IV, D.F. Beal (ed.), Pergamon Press, pp 37-51, 1986
47. *D.A. McAllester* **A New Procedure for Growing Min-Max Trees** Artificial Intelligence, 35(3), pp 287-310, 1988
48. *P. Mysliwietz* **Konstruktion und Optimierung von Bewertungsfunktionen beim Schach** Doctoral thesis, University of Paderborn, Germany, 1994
49. *M. Newborn* **A Parallel Search Chess Program** ACM Annual Conference 1985, pp 272-277, 1985
50. *M. Newborn* **Unsynchronized Iterative Deepening Parallel Alpha-Beta Search** IEEE Transactions on Pattern Analysis and Machine Intelligence, 10(5), pp 687-694, 1988
51. *S.W. Otto, E.W. Felten* **Chess on a Hypercube** The Third Conference on Hypercube Concurrent Computers and Applications, 2, pp 1329-1341, 1988
52. *J. Pearl* **Heuristics: Intelligent Search Strategies for Computer Problem Solving** Addison-Wesley Publishing Company, 1984
53. *A. Reinefeld* **Spielbaum - Suchverfahren** Springer, 1989
54. *J. Schaeffer* **The History Heuristic and Alpha-Beta Search Enhancements in Practice** IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(11), pp 1203-1212, 1989
55. *J. Schaeffer* **Distributed Game-Tree Searching** Journal of Parallel and Distributed Computing, 6(2), pp 90-114, 1989
56. *C.E. Shannon* **Programming a Computer for Playing Chess** Philosophical Magazine 41, pp 256-275, 1950
57. *G. Schrüfer* **Minimax-Suchen Kosten, Qualität und Algorithmen** Doctoral thesis, University of Braunschweig, Germany, 1988
58. *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
59. *K. Thompson* **Computer Chess Strength** Advances in Computer Chess III, M.R.B. Clarke (ed.), Pergamon Press, pp 55-56, 1982
60. *A. Szabo, B. Szabo* **The technology curve revised** ICCA Journal 11(1), 1988
61. *J. v. Neumann, O. Morgenstern* **Theory of Games and Economic Behavior** Princeton University Press, Princeton, USA, 1944
62. *J.W.H.M. Uiterwijk* **The Countermove Heuristic** ICCA Journal 15(1), pp 8-15, 1992
63. *O. Vornberger, B. Monien* **Parallel Alpha-Beta versus Parallel SSS*** Proceedings IFIP Conference on Distributed Processing, North Holland, pp 613-625, 1987
64. *T. Warnock, B. Wendroff* **Search Tables in Computer Chess** ICCA Journal, 11(1), pp 10-13, 1988
65. *J.-C. Weill* **The ABDADA Distributed Minimax-Search Algorithm** ICCA Journal, 19(1), pp 3-16, 1996
66. *E. Zermelo* **Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels** 5. Int. Mathematikerkongreß, Cambridge, 2, pp 510-504, 1912
67. *A.L. Zobrist* **A New Hashing Method with Applications for Game Playing** TR-88, University of Wisconsin, Computer Science Department
Reprint in ICCA Journal 13(2), pp 69-73, 1990

This article was processed using the LaTeX macro package with LLNCS style