A Fully Distributed Chess Program [*][†]

R.Feldmann P.Mysliwietz B.Monien

Department of Mathematics and Computer Science
University of Paderborn
West Germany

ABSTRACT

We show how to implement the $\alpha\beta$-enhancements like iterative deepening, transposition tables, history tables etc. used in sequential chess programs in a distributed system such that the distributed algorithm profits by these heuristics as well as the sequential does. Moreover the methods we describe are suitable for very large distributed systems. We implemented these $\alpha\beta$-enhancements in the distributed chess program *ZUGZWANG*. For a distributed system of 64 processors we obtain a speedup between 28 and 34 running at tournament speed. The basis for this chess program is a distributed $\alpha\beta$-algorithm with very good load balancing properties combined with the use of a distributed transposition table that grows with the size of the distributed system.

1. INTRODUCTION

In this paper we describe a fully distributed chess program *ZUGZWANG* running on a network of Transputers. We present experimental results that show the efficiency of our implementation. The good behavior of the sequential $\alpha\beta$-algorithm shown by [KM75] mainly poses three difficulties to any distributed algorithm. First, the cutoffs of the sequential algorithm may be overlooked by the distributed version resulting in search overhead. Second, the problem of load balancing is very difficult for tree decomposition algorithms. This is caused by the unpredictable size of subproblems as well as by the idle times resulting from iterative deepening. Third, the

[*]This work was partly supported by grant no. Mo 285/3-2 from the DFG
[†]The Parsytec GmbH, Aachen provided us with a 32 Transputer system for tournament play

sequential $\alpha\beta$-algorithm strongly profits by several $\alpha\beta$-enhancements as iterative deepening, transposition tables, history tables, killer lists etc.. These $\alpha\beta$-enhancements often delay the distributed algorithm. For example iterative deepening is inherent sequential and the transposition table should be accessible to all processors in the system.

In [VM87, FMMV89, FMMV90] we present a distributed $\alpha\beta$-algorithm that achieves a speedup of 11.5 running on a network of 16 processors. This algorithm shows a very good performance when searching well ordered game trees. This is due to the Young Brothers Wait Concept that helps to reduce search overhead. Furthermore the algorithm shows very good load balancing properties. However the algorithm from [VM87, FMMV89, FMMV90] does not use $\alpha\beta$-enhancements. In this paper we present a distributed chess program using the distributed $\alpha\beta$-algorithm from [VM87, FMMV89, FMMV90]. The distributed version of this chess program profits by the state-of-the-art $\alpha\beta$-enhancements as well as the sequential one does. It uses iterative deepening, a distributed transposition table, shared history tables and shared killer lists, zero-width search and a distributed quiescence search. The transposition table for example is implemented as a distributed transposition table. The loss of work load caused by the iterative deepening algorithm is kept very small by the good load balancing capabilities of our distributed algorithm.

The reduction of search overhead was one of the main topics in the field of parallel $\alpha\beta$-algorithms. Akl et.al. proposed the mandatory work first approach in [ABD80]. The PVS algorithm is used in [MP85, MOS86, Sch89b, New88, HSN89]. A description of this algorithm can be found in [MC82]. It evaluates right sons of game tree nodes with a minimal $\alpha\beta$-window in parallel and then re-evaluates them if necessary. Processors are assigned to subtrees along the principal variation. Alternatively game tree nodes are evaluated in parallel only if they had acquired an $\alpha\beta$-bound before ([FK88]). Another approach applies in the distributed chess program Waycool running on a hypercube ([OF88]): if the transposition table proposes some move for a game position then this move is tried first. Parallel evaluation of the other moves is started only if the evaluation of the transposition move yields no cutoff. The PVS algorithm and the approach of Ferguson and Korf guarantee that best ordered game trees are searched without any search overhead. This leads to very efficient implementations, if the game trees to be searched are close to the minimal game tree as it is usually the case for sequential chess programs. However, to keep the game trees close to the minimal game tree, the sequential chess programs use several $\alpha\beta$-enhancements. The use of these $\alpha\beta$-enhancements in a distributed system together with a distributed $\alpha\beta$-algorithm with very good load balancing properties is the main topic of this paper.

In [MP85] Marsland and Popowich compared the use of local and global transposition tables. Schaeffer uses a hybrid version of these methods for his chess program Sun Phoenix ([MOS86, Sch89b]). The same program compares knowledge accumulated in the history tables after every iteration of its iterative deepening algorithm. Newborn ([New88]) tried to overcome the idle times caused by iterative deepening with the UIDPABS algorithm. Zero-width search accelerates the sequential $\alpha\beta$-algorithm. Otto and Felten in [OF88] claimed that this method does not parallelize as well as the $\alpha\beta$-algorithm without zero-width search. Therefore they parallelized the $\alpha\beta$-algorithm without zero-width search. In [Sch89a] Schaeffer stated that speedups are strongly tied to the (in)efficiency of the $\alpha\beta$-search and that the use of $\alpha\beta$-enhancememts in a parallel implementation of the $\alpha\beta$-algorithm dramatically affects the performance of a parallel implementation. This results in speedups of 5.93 using 16 processors in [HSN89], 5.67 using 9 processors in [MOS86, Sch89b] and 5.03 using 8 processors in [New88]. Moreover increasing the number of processors either decreases the speedup ([HSN89]) or at least does not increase it ([MOS86, Sch89b]). The speedup of 101 using 256 processors presented in [OF88] has been achieved by parallelizing a suboptimal version of the sequential $\alpha\beta$-algorithm. Another interesting result is the speedup of 12 achieved by Ferguson and Korf in [FK88] for an Othello playing program using iterative deepening.

Our distributed system achieves speedups of 15.77 running on 16 processors and 25.08 running on 32 processors searching a 7-ply search on the positions of the Bratko-Kopec experiment ([BK82]). Very recent experiments show that 64 processors can achieve a speedup of 34 for a 7-ply search running on tournament speed. This is obtained by the use of a distributed $\alpha\beta$-algorithm from [VM87, FMMV89, FMMV90] with very good load balancing properties. The main reason for the good behavior of our distributed chess program however is the fact that we combined a parallelization of the $\alpha\beta$-enhancements together with the good load balancing properties of our distributed $\alpha\beta$-algorithm. We implemented a distributed transposition table, iterative deepening and a distributed quiescence search. Not yet finished is the implementation of a shared history table and shared killer lists that are updated dynamically during the tree search in the distributed system. It turns out that our chess program is a good chess player too. It played successfully during the 2. Computer Games Olympiad in London 1990.

A short description of some features of our sequential chess program is given in section 2. In sections 4. and 5. we describe our distributed chess program. In section 6. we present results gained from experiments with up to 64 processors.

2. ENHANCEMENTS OF THE SEQUENTIAL CHESS PROGRAM

In this section we describe the basic features of our sequential chess program *ZUGZWANG*. Like most of todays programs *ZUGZWANG* performs a full width search of the tree of possible moves and countermoves. The program starts with a search depth 1 and increments the depth by one until the allotted time has run out, or a given maximum depth is reached. This technique known as iterative deepening has proven to be quite effective, especially in combination with several other heuristics, that are described in the following.

During a depth d search non-quiescent nodes in level d are extended through promising moves which include some capturings, pawn promotions, and checking moves. This quiescence search is essential to reduce evaluation errors, which occur if positions, that are not tactically quiet are evaluated with a static evaluation function.

The search algorithm used is based on Reinefeld's Negascout algorithm with fail soft improvement ([Rei89]). This procedure was slightly changed to incorporate aspiration windows which are provided by the preceding search iteration. We used this kind of search algorithm despite the fact that the zero-width searching was found to parallelize not as well as alpha-beta ([OF88]), because it appears to be significantly better on sequential computers.

Several well known heuristic methods are used to achieve well ordered game trees :

- The principal variation and the refutation lines for the other moves at root level computed in a d-ply search are used to guide the search during the d+1-ply search.
- History and killer tables are an inexpensive but efficient tool to rearrange successors at a non-leaf position. The basic idea behind these two techniques is, that moves found to be best in one position are likely to be good in lots of other positions (where they are legal) throughout the whole game tree. A history table remembers for a move m the count of positions visited in which m was found to be the best move or to provide a cutoff. The arrangement of moves is done due to this count. Killer tables store a fixed number of killer moves (i.e moves that yield a cutoff) for each level of the game tree. Whenever a cutoff occurs in level i the corresponding move is stored in that table with one bonus point, if the move is not already included in the killer list of level i. If the move is already included, its bonus is incremented by one. If a move has to be stored in an already full list, the move with smallest bonus is deleted first. At non-leafs these killer moves are tried prior to other moves, because they have a good chance to yield an immediate cutoff. Our implementation uses two killers per level.
- Transposition tables store informations about positions that have already been evaluated. These informations may be used, if positions are visited more than once. This may occur for two reasons : a position can be reached by several move sequences, and many positions visited during a prior iteration are also visited in the succeeding iteration.

  Stored information includes the minmax value, the best move, and the corresponding search depth. Before a subtree is searched, a check is done whether the transposition table can provide any useful information. This may lead to the elimination of the whole subtree, to improved windows, or only to an improved tree ordering, if the best move stored is considered first. Our current implementation uses a hash table with 20000 entries, in which only non-leaves within the regular search depth are stored (i.e. during a d-ply search only position in level less or equal d-1 are stored).

History, killer, transposition tables are rather domain independent mechanisms that gain their power through their special kind of experience, which is accumulated during the search. These techniques together with several additional features of minor importance lead to game trees that are close to the size of the minimal game tree. On our hardware *ZUGZWANG* running on a single processor visits about 350 nodes per second.

3. HARDWARE USED

In this section we describe the hardware used to run our sequential and distributed algorithm. We used the Inmos Transputer T800 as a sequential processor as well as to build up processor networks. This processor has nice features we used: hardware supported process scheduling and the ability to simultaneously compute and communicate. It is the basic component of our Transputer systems produced by Parsytec/Paracom GmbH, Aachen.

A Transputer is a microprocessor with 4 communication links to connect it to other Transputers. It runs at a speed of 10 MIPS. This results in a performance of $1.58 * 10^{-6}$ seconds for the 4 Byte integer operation $a := b + c$.

Transputer are designed to run parallel processes on one processor very efficiently by hardware supported time sharing. In particular it is able to communicate on all links and compute on its own local memory in parallel. It is this feature that proved to be very helpful when implementing shared heuristics. As long as the processors run computing processes they are not significantly delayed by communication tasks running at the same time.

Our Transputers have a local memory of 1 MByte each and run at a clock speed of 20 MHz.

Since each Transputer has 4 links for connecting it with other Transputers, the user is allowed to build any Transputer networks with degree less or equal 4. In such networks Transputers are connected by point-to-point connections to their neighbors. Therefore two neighbored Transputers that communicate do not disturb the work of the other Transputers in the network. Especially communication tasks of other Transputers are not delayed by this communication as for example in a bus oriented system. On the other hand messages from one Transputer to a non-neighbor have to be routed along possibly several intermediate processors. However, this intermediate Transputers are able to route the message without a significant loss of efficiency.

Our distributed algorithm works for any underlying Transputer network. For our experiments, however, we use the DeBruijn network family $DB(n)$, $n \in \mathbb{N}$, first defined in [deB46]:

$DB(n) = (V, E)$, where V, the set of nodes is defined as

- $V = \{0,1\}^n$ the set of binary strings of length n and
- $E = \{\{x, s(x)\} \mid x \in V\} \cup \{\{x, e(x)\} \mid x \in V\}$.

Here $s : V \to V$ is the shuffle edge function

$$s(x_{n-1} \cdots x_0) = x_{n-2} \cdots x_0 x_{n-1}, \;\; x_i \in \{0,1\}$$

and $e : V \to V$ is the shuffle exchange edge function

$$e(x_{n-1} \cdots x_0) = x_{n-2} \cdots x_0 \bar{x}_{n-1}, \;\; x_i \in \{0,1\}$$

Since the graphs of the DeBruijn network family have maximum degree 4 and logarithmic diameter, this scheme is well suited to serve as a Transputer interconnection scheme.

Each $DB(n)$ for $n \geq 3$ has three edges that make little sense in a Transputer application: the self loops of node $0 \cdots 0$ resp. $1 \cdots 1$ and one of the double edges between nodes $1010 \cdots$ and $0101 \cdots$. Therefore we run our algorithm on a slightly modified version:

- the self loop of node $0 \cdots 0$ is connected to a host Transputer. This host Transputer is the only one that has access to keyboard, screen, disks, etc. The host does not take part in the game tree search but serves as I/O administrator.

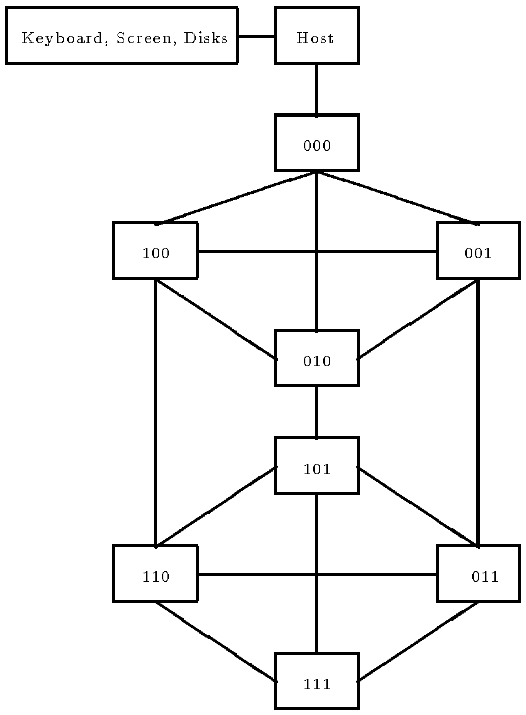

Figure 1: DeBruijn-like connected Transputer network of dimension three

- one of the edges between $1010\cdots$ and $0101\cdots$ is deleted.
- Transputer $0\cdots0$ is connected to $0101\cdots$ and Transputer $1\cdots1$ to $1010\cdots$.

Figure 1 shows a DeBruijn-like connected Transputer network of dimension 3.

4. OUR DISTRIBUTED $\alpha\beta$-ALGORITHM

In this section we recall our strategy for solving tree search problems on a distributed system. Our strategy is fully distributed and allows the use of an arbitrary number of processors. It has proven to behave well not only if applied to game tree search, but also to branch and bound applications ([MV87]). A first version of a distributed $\alpha\beta$-algorithm using this strategy is described in [VM87]. A more detailed description of the distributed algorithm can be found in [FMMV90].

In our strategy every processor has the same program and all processors perform similar tasks. We do not use any kind of centralized mechanisms, because this would lead to a bottleneck if the number of processors increases.

We talk of search problems in terms of nodes of the corresponding search tree. The root ϵ of a

game tree represents the whole search problem to compute $F(\epsilon)$. For any non-leaf v the problem to evaluate v has the subproblems to evaluate some sons $v.i$ $(1 \leq i \leq b(v))$. The subsolutions are combined (by maximization or minimization) in order to yield *F(v)*.
The computation starts with the root node assigned to one of the processors, the other processors are idle. A processor that works on a problem splits it into subproblems. Since only one subproblem is attacked at a time the other subproblems are stored in the local memory. Therefore it is possible to transfer those subproblems to other processors. If there is more than one subproblem ready for transmission, the algorithm transmits one at highest level in the tree.

In our scheme idle processors must take care to get work for themselves (with the exception of the initial search problem which is assigned to one of the processors). This means that an idle processor has to ask one of the other processors for work.
A processor is responsible for a problem it got until its solution is computed. It can send subproblems to other processors which then become its slaves. After its subproblem is finished it has to respond the solution to the sender of the problem. The processor the root node is assigned to, will eventually finish with the solution of the whole problem.

Figure 2 shows a few snapshots of a distributed computation. Processor P_1 is working on a

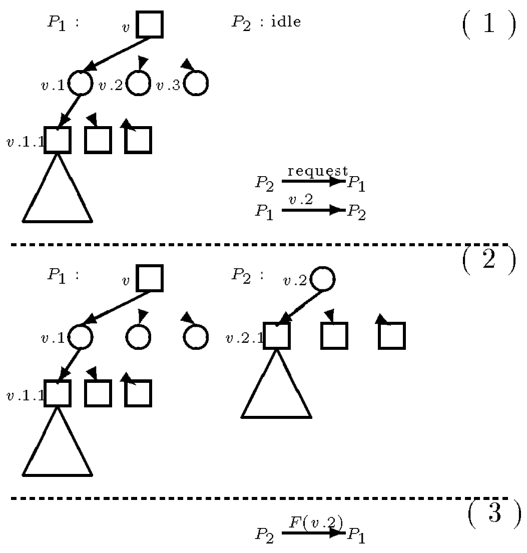

Figure 2: Example computation of the distributed algorithm

problem rooted at v. Currently it works on subproblems located in the labeled subtree. An idle processor P_2 sends a request for work to P_1 and therefore P_1 transmits node $v.2$ to P_2 which is already generated but not evaluated. P_1 labels this node as transmitted and P_2 starts to work on that problem.
After P_2 finishes work on $v.2$ it returns the value *F(v.2)* to P_1.

This example does not take into account $\alpha\beta$-windows and cutoffs. Since a search window has been assigned to each node by the $\alpha\beta$- algorithm, this window clearly has to be transmitted too, whenever a node is sent to a requesting processor. Each processor that observes an improvement of a search window assigned to a node v has to inform all of its slaves with subproblems rooted below v. Three different actions may force the update of a search window:

(i) finishing the evaluation of a local subproblem

(ii) receiving the solution of a subproblem solved by a slave

(iii) receiving an UPDATE-message from the master

The three cases can be handled the same way:
Decreasing a search window for a node v forces the update of all current search windows below v. If some subproblems rooted below v have been transmitted, these processors are informed by a NEWBOUND-message. Note that this rule is applied recursively. Actually a search window for a node v may not only be decreased, but can also collapse. In this case all problems located below v are immediately finished. To reduce the number of communications we introduce a special message CUTOFF which is sent to all processors that got a subproblem rooted below v to prevent them from responding their results that are of no use now. A processor receiving a CUTOFF-message finishes the problem it is working on and sends a CUTOFF-message to all of its slaves.

Using a zero-width approach the above remains essentially true, although now in almost every case a NEWBOUND-message leads to an immediate cutoff. Only processors working on moves that had to be re-searched may possibly continue their work with an improved window. Since processors are completely responsible for problems they got, a processor, that determines a fail high or fail low has to re-search this problem with an open search window and to respond to its master after this re-search.

In our implementation a random chosen processor is asked for work, regardless of its distance in the processor network. This leads to a very efficient load balancing, especially if the game trees are very irregular like those found in chess programs.

In [HSN89] Hyatt concludes that future algorithms will be more complex in order to allow for an efficient use of a large number of processors. We believe that this very simple load balancing scheme is the solution to the problem of unequal work assignment. Note that we make no restriction to the size of problems that may be distributed. Every node, even nodes in the quiescence search are distributed, although the receiving processor will often find this node to be a leaf node. This prevents processors from performing huge quiescence searches while lots of other processors may be idle.

Since the wide variation of tree sizes appears to be one of the main reasons for the often used PVS approach to fall short on achieving good speedups ([New88]), several recent approaches to distributed $\alpha\beta$-search made the way towards a more dynamic correspondence between processors and nodes of the game tree. Hyatt improved the PVS algorithm to EPVS ([HSN89]), Schaeffer improved PVS to DPVS ([MOS86] and more recently to a more dynamic version of DPVS ([Sch90]). Our solution makes the consequent step to a really dynamic processor allocation.

Search overhead arises if the parallel algorithm visits nodes the sequential algorithm would not visit. To prevent the algorithm from searching many superfluous nodes we have to choose subproblems for transmission carefully. We use a concept we call Young Brothers Wait Concept which is defined in [FMMV90]. This concept avoids any search overhead if working on best ordered game trees and has comparable small search overhead if working on well ordered game trees:

Assume a processor P is working on a subproblem rooted at a node v and is currently visiting a node $w = v.v_1.v_2.\cdots.v_n$ when a request for work arrives and it has to decide which subproblem (if any) it is going to transmit. The set of all subproblems ready for transmission at this moment is the set of all right brothers of nodes $v.v_1.\cdots.v_i$ $(i \in \{1, \cdots, n\})$.
The Young Brothers Wait Concept now postulates:

> The eldest son of any node must be completely evaluated before younger brothers of that node may be transmitted.

Especially, if all subproblems ready for transmission do not yet have an evaluated left brother, then these subproblems have to wait. They can not be transmitted unless the evaluation of the first son has finished. Note that the node transmitted in the example computation may not be transmitted using the Young Brothers Wait Concept.

This restriction reduces search overhead. Assume a node $v.i$ without evaluated elder brother is transmitted from a processor P_1 to a processor P_2. The search overhead will be very large, if the evaluation of $v.1$ on P_1 causes a cutoff. In this case all the work P_2 does on its subproblem is completely superfluous. Note that the probability for a cutoff after evaluation of the first son is very high, especially in game trees which are not far away from being best ordered. Even if the evaluation of $v.1$ does not cause a cutoff, an improvement of the search window for v and thus for $v.i$ is very likely. In this case P_2 has evaluated $v.i$ with a perhaps worse search window and therefore it may visit many nodes, it would not have visited in the presence of the improved window.

Although the Young Brothers Wait Concept does not prevent us from doing superfluous work in general, it leads to good behavior when searching well ordered game trees and to perfect behavior in the case of best ordered game trees.

5. $\alpha\beta$-ENHANCEMENTS IN THE DISTRIBUTED SYSTEM

In this section we describe how some of the most important $\alpha\beta$-enhancements as transposition tables, killer lists and history tables can be implemented in a distributed system.

Iterative deepening is a very powerful tool to improve the performance of the sequential $\alpha\beta$-algorithm. This is done by searching the game tree sequentially for depth $1, 2, 3 \cdots$. The depth

i search speeds up the depth $i+1$ search by providing it with a principal variation for the best depth i move and refutations for the other moves. Iterative deepening in the distributed algorithm is done the same way as in the sequential algorithm.
During the depth i search of the distributed algorithm the principal variation can be computed in the same way as in the sequential algorithm. With the exception that an update of the principal variation of some game tree node v may be necessary, if the master for v gets a RETURN - message from its slave that improves the minmax-value of v. Refutations are computed analogously.
At the beginning of the depth $i+1$ search the depth i principal variation is used in the same way as in the sequential algorithm. This is possible without any change, because it is searched sequentially using the Young Brothers Wait Concept. Refutation are sent together with the nodes to the slaves, if necessary.
In order to run iterative deepening in our distributed algorithm we must accept a loss of work load. However this loss of work load is not significant for larger search depths. This is due to the excellent load balancing capabilities of our algorithm and to the fact that there are only $O(d^2)$ short moments during the computation of iterative deepening up to depth d that all processors but one are idle. In figure 4 we present some experiments that indicate these excellent load balancing capabilities.

Another $\alpha\beta$-enhancmenent that poses difficulties to the distributed algorithm is the *transposition table.* Three approaches have been tried in distributed algorithms to give the transposition table the same power as in the sequential algorithm:

- global transposition table
  One special processor holds the whole hash table. Requests and stores to the transposition table must be sent to this processor. Answers are returned.
- local transposition table
  Every processor holds its own local hash table
- distributed transposition table
  The hash table is distributed among all processors. Thus every processor holds a part of the hash table

The effects of local and global transposition tables are compared by Marsland and Popowich in [MP85]. Their experiments show that the global transposition approach suffers from an increase in communication overhead while in the local transposition approach every processor is provided only with a part of the accumulated knowledge. Hybrid versions of these two concepts are used in chess programs like Sun Phoenix ([Sch89b]).
The third method to hold transposition tables is to distribute the whole table among all processors. Every processor holds a part of the data structure. Requests and stores to as well as answers from the transposition table must be implemented by exchanging messages between the requesting processor and the one that holds the requested transposition entry. This approach is used by Otto and Felten in [OF88].

The first advantage of this method is that the whole knowledge accumulated in the transposition table is available to all processors. On the other side the communication delays that "destroy the programs performance" ([MP85]) can be kept very small, because the communication bottleneck of a central transposition manager is avoided. Our experiments show that a distributed transposition table can be implemented very efficiently on the hardware we use for our program. The main reason for us, however, to choose the distributed transposition table approach was that the transposition table grows with the underlying distributed system. The more processors we use the more transposition table entries are accessible to all processors.

We implemented the distributed transposition table as described below. Each of the processors holds an equal amount of the transposition table entries in its local memory. A hash function $h : \{chesspositions\} \rightarrow \{0, \cdots, p \cdot k\}$ is used to determine the processor number $h(v)\ mod\ p$ and the local entry address $h(v)\ div\ p$. Here p is the number of processors used and k the number of transposition entries available at a single processor. A processor that wants to access the transposition entry for node v sends a REQUEST-message for $h(v)\ div\ p$ to processor $h(v)\ mod\ p$. A processor that gets a REQUEST-message for the local address x sends back the transposition entry $T(x)$ in an ANSWER-message. The transposition entry is checked at the receiver of the ANSWER-message. Whenever a processor wants to store a transposition entry for node v it sends a PUT-message for $h(v)\ div\ p$ to processor $h(v)\ mod\ p$. The routing of messages as well as the quick response to REQUEST-messages is perfectly supported by the Transputer. The ability to run parallel processes on a single Transputer guarantees fast routing and response without delaying the local computation significantly.

However, there is a problem one has to overcome. A processor that sends a request to another one is idle until it receives the answer to its request. This amount of time significantly delays the requesting processor. Again the ability of Transputers to run several processes in parallel is used to reduce this idle time. The algorithm we implemented is the following:
Assume a processor in the distributed system starts to evaluate a node v.

```
1) calculate $h(v)$;
2) PAR
      send a transposition request for $h(v)\ div\ p$ to $h(v)\ mod\ p$,
      wait for the answer;
      update the chess position $v$;
   END PAR
```

Processors use the time from sending the transposition request to the receipt of an answer by updating the chess position associated with v. This time is wasted only, if the exact value or bounds sufficient for an immediate cutoff are stored for the position v.

A delay of only about 5 % is introduced by this method:
We computed a 5-ply full width search on position 22 of the Bratko-Kopec experiment under different circumstances. During the first run one sequentially working processor had access to the transposition table of a 32 processor system. The only communication that took place in this system is that for the transposition requests, answers and stores. For the second run we additionally simulated the communication of the system at that moments, when all processors but one are idle. These idle processors sent request for work to randomly chosen processors in the network but never got any work. Therefore at this moments during a computation phase the communication was very high. For the third run the use of any transposition table is forbidden.

Pos. 22, depth 1-5	use of transp. table	nodes/sec	delay in %
1 processor	yes	318.30	0.00
1st run, 32 processors	yes	303.00	4.81
2nd run, 32 processors	yes	290.96	8.59
1 processor	no	319.33	0.00
3rd run, 32 processors	no	308.09	3.52

Figure 3: Delay by transposition communication

Communication on the links is simulated as for the second run. We measured the number of nodes per second, the working processor visits during its game tree search. The results presented in figure 3 show the effectiveness of our method.

The sequential algorithm without any transposition use is delayed by the link activities by 3.5 %. Using additionally the distributed transposition table, the sequential algorithm is delayed by 8.5 %. Thus the very low price of only a five percent delay must be paid for the advantage of a very large transposition table. The time necessary to update a chess position v is a constant, but the time to route the transposition messages depends at least on the diameter of the Transputer network. However, the diameter of the network we have chosen grows only logarithmically in the number of processors. Therefore we are sure that, when doubling the number of processors, the effect of a doubled transposition table will compensate for the growing of the network diameter by one.

Killer lists as well as *history tables* are held locally by each processor in the distributed system. To implement history tables and killer lists similar to the distributed transposition table is not appropriate since these tables are used to sort the moves at all interior nodes. However to include more than local knowledge in this tables we use a strategy we call "Shared History" resp. "Shared Killer": Whenever local computation has changed the value of an entry in the history table by a special amount, an UPDATE-HISTORY-message for this entry is sent to all neighbored processors. This message contains a tuple (m, x) where m is a move and x the history value of this move in the history table of the sender. The receiver of an UPDATE-HISTORY message containing (m, x) sets its own local history value for move m to the maximum of this value and x. An UPDATE-KILLER-message is sent to all neighbors, if local computation has changed a move in the some killer list. This method is inspired by load balancing methods we developed for branch & bound algorithms ([MV87]). We expect that the additional communication going on with this strategy will be very small. This is because UPDATE-HISTORY resp. UPDATE-KILLER messages will be much less frequent than messages necessary for the transposition table. Remember that for any inner tree node there are three messages necessary to use the distributed transposition table. This messages must be routed across the whole Transputer system. For Shared History and Shared Killer strategies at most one message per node must be sent to a constant number of neighbors. Routing is not necessary. Thus we believe that Shared History and Shared Killer can be done without a significant drop of performance. Unfortunately the implementation of this strategies has not yet finished. Therefore in this paper we present experiments with local history and killer tables.

6. EXPERIMENTAL RESULTS

In this section we describe some of our experiments. The diagrams for speedup, load and search overhead show the average speedup, work load and search overhead we obtained searching the 24 Bratko-Kopec positions (see [BK82]) with a various number of processors and various search depths. Note that the maximum average run time of the 64 processor system is 277 seconds. This is roughly tournament speed.

To test our distributed algorithm we run several tests on the 24 positions from [BK82]. All these positions were searched with a 5-ply, 6-ply and 7-ply depth. This performance is compared to the sequential algorithm using all the $\alpha\beta$-enhancements we described in 2..

This sequential algorithm is an efficient searcher. The figure below shows, that *ZUGZWANG* searches trees with roughly the size of the minimal game tree.

5-ply	6-ply	7-ply	
1.002	1.098	0.994	$\times$ the min. tree

For fixed search depth d we define the following measures for the performance of our distributed algorithm running a d-ply search: Let B_i be the i-th position from the Bratko-Kopec set of test positions, Let $T_i(p)$ be the total time, $I_i(p)$ the average idle time and $N_i(p)$ the number of nodes visited by the p processor version of our distributed algorithm for a d-ply search on B_i.

$$SPE(p) := (\sum_{i=1}^{24} T_i(p))/(\sum_{i=1}^{24} T_i(1))$$

$$LD(p) := 100 \cdot [1 - (\sum_{i=1}^{24} I_i(p))/(\sum_{i=1}^{24} T_i(p))]$$

$$SO(p) := 100 \cdot [1 - (\sum_{i=1}^{24} N_i(p))/(\sum_{i=1}^{24} N_i(1))]$$

The curves for $SPE(p)$, $LD(p)$ and $SO(p)$ are given in the three diagrams for speedup, load and search overhead for 5-ply, 6-ply and 7-ply search and processor numbers 4,8,16,32 and 64.

Speedups grow with an increasing of search depth. This effect indeed is not new but also described in [Sch89b].

64 processors yield a speedup of 34.77 compared to the sequential version. However, the table in figure 5 shows that this number is influenced strongly by the speedup of 63.57 for the position 5. The same is true for the speedups of smaller systems. For this reason we present the speedups, loads and search overheads for a 7-ply search on the single positions in the table of figure 5. At the end of this table we present the measurements for the 6-ply search of position 5 and give a Φ'- row that results if the 7-ply search of position 5 is replaced by the 6-ply search. The speedup decreases to 27.99.

	Speedup	Load
7-ply search	40.55	90 %
1-7-ply search	34.77	83 %

Figure 4: Speedup and load for the last iteration (64 processors)

Another interesting fact is the superlinear speedup for small processor numbers and large problems. We conjecture that this superlinear speedup is due to the larger transposition table the distributed algorithm has access to. The search overhead resulting from this experiments is negative. An interesting feature of a distributed algorithm that influences the performance of the distributed algorithm is its nondeterminism. This nondeterminism is due to the facts that our algorithm runs on a completely distributed system and to that fact that our algorithm acts nondeterministically if a processor is looking for work. For single positions running times of the distributed algorithm can be observed that differ widely. However these differences seem to disappear in the average data of the 24 positions.

The load we present in the load diagram is measured as the ratio of idle time and total time. For the Transputer hardware we use one has to mention that some amount of CPU work that is necessary to buffer messages is done in parallel with local $\alpha\beta$-search. For this reason this amount of time is not included in the idle times and therefore not considered in the load. The load diagram shows, that the work load is strictly decreasing for increasing processor numbers. For the 7-ply searches however the work load in the 64 processor system is still good (83 %). To indicate the excellent load balancing capabilities of our algorithm we calculated the load and speedup for a 7-ply search: We subtracted the results for the 6-ply iterative deepening from the results for the 7-ply iterative deepening of the 64 processor system. Figure 4 compares this results with the results obtained by a 7-ply iterative deepening.

The search overhead we present in the search overhead diagram generally increases with the number of processors, but is still moderate for the 7-ply search with 64 processors (36,6 %). For small distributed systems and 7-ply search it is surprisingly negative in the average. This indicates that the larger transposition table of this systems compared to the sequential version accelerates the $\alpha\beta$-search. However this effect is not useful in practice because the average total time of 610 seconds for a 7-ply search on 16 processors is far more than can be used in playing tournaments.

7. CONCLUSIONS

We presented the implementation of the distributed chess program *ZUGZWANG*. This chess program profits strongly by the use of a distributed transposition table and other $\alpha\beta$-enhancements we implemented. The basis for its very good performance is a distributed $\alpha\beta$-algorithm that achieves an excellent load balancing. This results in an average speedup of at least 28 at tournament speed using 64 processors. Our experiments indicate that the speedup of our algorithm will even increase again significantly if we double the number of processors to 128.

		1 processor		64 processors				
No.	move	N	T	N	T	LD	SO	SPE
B01	d6d1!	279	1	302	1	3	8.24	1.00
B02	e4e5	960353	2794	1450340	110	75	51.02	25.40
B03	f6f5!	1234177	3934	1248054	110	67	1.12	35.76
B04	e5e6!	2694688	8341	2642330	182	77	-1.94	45.83
B05	c3d5!	27045744	83853	23755133	1319	93	-12.17	63.57
B06	g5g6!	135159	341	196952	35	32	45.72	9.74
B07	h5f6!	2151962	6761	2928631	222	74	36.09	30.45
B08	f4f5!	40244	98	58387	18	18	45.08	5.44
B09	d1e1	4880187	15146	10613436	629	87	117.48	24.08
B10	c6e5!	3145315	9514	7878164	493	86	150.47	19.30
B11	f2f4!	3144883	9990	4268380	294	84	35.72	33.98
B12	d7f5!	1174965	3832	1720659	150	70	46.44	25.55
B13	a1c1	4294276	13595	8979006	590	88	109.09	23.04
B14	d1d2!	1183076	3719	1875714	154	68	58.55	24.15
B15	g4g7!	591698	1830	1038041	83	72	75.43	22.05
B16	d2e4!	1321136	4245	1394034	120	68	5.52	35.38
B17	h7h5!	863240	2742	1428535	124	71	65.49	22.11
B18	c5b3!	3518916	11209	5322578	346	87	51.26	32.40
B19	e8e4!	1999356	6233	3113029	216	78	55.70	28.86
B20	e1g1	2897562	9594	5668009	387	87	95.61	24.79
B21	f5h6!	4676965	14724	5393288	344	89	15.32	42.80
B22	b7e4!	1580285	4925	4420070	306	84	179.70	16.09
B23	c8f5	2234088	7336	3600473	267	82	61.16	27.48
B24	e4f5	2119696	6344	1905989	147	72	-10.08	43.16
Σ	18	73888253	231112	100899579	6647			
Φ		3078677	9629	4204149	277	83	36.56	34.77
B05	a2a4!	1807502	5715	1842254	138	75	1.92	41.41
Φ'		2027083	6373	3291111	227	81	62.36	27.99

Figure 5: Performance of the 64 processor system searching 7 ply

Acknowledgements

We acknowledge gratefully the help of the chess club *SK 1926 Paderborn.* In particular we thank Wolfgang Labahn, who participated in the design of *ZUGZWANG's* opening book.

References

[ABD80] S.G. Akl, D.T. Barnard, and 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*, pages 231–234, 1980.

[BK82] I. Bratko and D. Kopec. *A Test for Comparison of Human and Computer Performance in Chess, in Advances in Computer Chess 3, M.R.B. Clarke (editor)*, pages 31–56. Pergamon Press, 1982.

[deB46] N.G. deBruijn. A combinatorial problem. *Indagationes Math.*, 8:pp 461–467, 1946.

[FK88] Ch. Ferguson and R.E. Korf. Distributed tree search and its application to alpha-beta pruning. *Proceedings AAAI-88, Seventh National Conference on Artificial Intelligence*, 2:pp 128–132, 1988.

[FMMV89] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed game-tree search. *ICCA Journal*, 12(2):pp 65–73, 1989.

[FMMV90] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. *Distributed Game Tree Seach, in Parallel Algorithms for Machine Intelligence and Pattern Recognition, V. Kumar, L.N. Kanal and P.S. Gopalakrishnan (editors).* Springer Verlag, 1990.

[HSN89] M. Hyatt, B.W. Suter, and H.L. Nelson. A parallel alpha/beta searching algorithm. *Parallel Computing*, (10):pp 299–308, 1989.

[KM75] D.E. Knuth and R.W. Moore. An analysis of alpha - beta pruning. *Artificial Intelligence*, (6):pp 293–326, 1975.

[MC82] T.A. Marsland and M.S. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):pp 533–551, 1982.

[MOS86] T.A. Marsland, M. Olafsson, and J. Schaeffer. *Multiprocessor Tree-Search Experiments, in Advances in Computer Chess 4 D.F. Beal (editor)*, pages 37–51. Pergamon Press, 1986.

[MP85] T.A. Marsland and F. Popowich. Parallel game tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):pp 442–452, 1985.

[MV87] B. Monien and O. Vornberger. Parallel processing of combinatorial search trees. *Proceedings International Workshop on Parallel Algorithms and Architectures,* Math. Research Nr. 38, Akademie - Verlag Berlin, pages 60–69, 1987.

[New88] M. Newborn. Unsynchronized iterative deepening parallel alpha-beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 687–694, 1988.

[OF88] S.W. Otto and E.W. Felten. Chess on a hypercube. Technical report, California Institute of Technology, USA, 1988.

[Rei89] A. Reinefeld. *Spielbaum-Suchverfahren*. Springer-Verlag, 1989.

[Sch89a] J. Schaeffer. Comment on 'distributed game tree search'. *ICCA Journal*, 12(4):pp 216–217, 1989.

[Sch89b] J. Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6(2):pp 90–114, 1989.

[Sch90] J. Schaeffer. Personal communication. 1990.

[VM87] O. Vornberger and B. Monien. Parallel alpha-beta versus parallel sss*. *Proceedings IFIP Conference on Distributed Processing, Distributed Processing, North Holland*, pages pp 613–625, 1987.