PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

A HIGHLY PARALLEL CHESS PROGRAM

Edward W. Felten and Steve W. Otto

Caltech Concurrent Computation Program
Synchrotron Laboratory, 206-49
California Institute of Technology
Pasadena, CA 91126 U.S.A.

ABSTRACT

We have developed a parallel chess program to run on distributed memory, multiple instruction stream computers. The game tree is decomposed between processors using a recursive version of the principal-variation-splitting algorithm. Search times for related subtrees vary widely (up to a factor of 100) so dynamic reconfiguration of processors is necessary to concentrate on "hot spots" in the tree. A crucial feature of the program is the global hashtable; for this data structure we use a distributed-memory machine in a shared-memory mode. The speedup of the program is conservatively estimated to be 170 on a 512-processor NCUBE hypercube.

1 MOTIVATIONS

It is becoming clear that distributed-memory, multiple-instruction stream (MIMD) computers are successful in performing a large class of scientific computations (Fox 1988)(Hey 1988)(Wallace 1988). These computations tend to have regular, homogeneous data sets and the algorithms are usually "crystalline" in nature. The second generation of problems now being explored tend towards an amorphous structure and asynchronous execution. It is less obvious how well hypercubes or transputer arrays are suited to these new problems.

As an attempt to explore a part of this interesting region in algorithm space, we have developed a chess-playing program which runs on the NCUBE hypercube and on transputer arrays. Besides being a fascinating field of study in its own right, computer chess is an interesting challenge for distributed-memory parallel computers because:

- It is not clear how much parallelism is actually available—the important method of alpha-beta pruning conflicts with parallelism;
- Some aspects of the algorithm require a globally shared data set;
- The parallel algorithm has dynamic load imbalance of an extreme nature.

Before continuing, let us state that our approach to parallelism in computer chess is not the only one. Belle, Cray Blitz, Hitech, and Chiptest have shown that fine-grained parallelism (pipelining, specialized hardware) leads to impressive speeds (Frey 1983)(Ebeling 1985). Our coarse-grained approach to parallelism should be viewed as a complementary, not conflicting, method. Clearly the two can be combined.

2 SEQUENTIAL COMPUTER CHESS

In this section we will describe some basic aspects of what constitutes a good chess program on a sequential computer. Having done this, we will be able to intelligently discuss the parallel algorithm.

2.1 Tree Searching

At present, all competitive chess programs work by searching a tree of possible moves and countermoves. A program starts with the current board position and generates all legal moves, all legal responses to these moves, and so on until a fixed depth is reached. At each leaf node, an evaluation function is applied which assigns a numerical score to that board position. These scores are then "backed up" by a process called minimaxing, which is simply the assumption that each side will always choose the most favorable line of play. If positive scores favor white, then white picks the move of maximum score and black picks the move of minimum score. This is illustrated in Figure 1.

The problem with this brute-force approach is that the size of the tree explodes exponentially. The "branching factor" or number of legal moves in a typical position is about 35. Fortunately, the branching factor can be reduced by alpha-beta pruning, which always gives the same answer as brute-force searching but looks at far fewer nodes. Intuitively, alpha-beta pruning works by ignoring subtrees which it knows cannot be reached by best play (on the part of both sides). This reduces the effective branching factor from 35 to about 6.

The idea of alpha-beta pruning is illustrated in Figure 2. Assume that all child nodes are searched from left to right in the figure. On the left side of the tree (the first subtree searched), we have minimaxed and found a score of +4 at depth 1. Now start to analyze the next subtree. The children report back scores of +5, -1, ... The pruning happens after the score of -1 is returned:

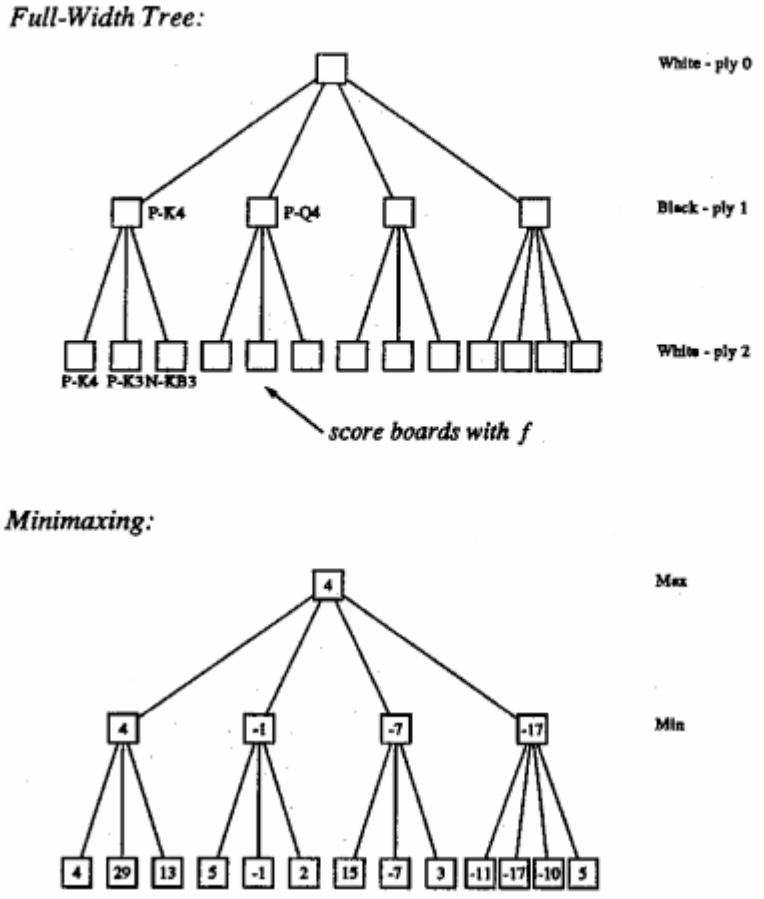

Figure 1: Game playing by tree searching. The top half of the figure illustrates the general idea: develop a full-width tree to some depth, then score the leaves with the evaluation function, *f*. The second half shows minimaxing---the reasonable supposition that white (black) chooses lines of play which maximizes (minimizes) the score.

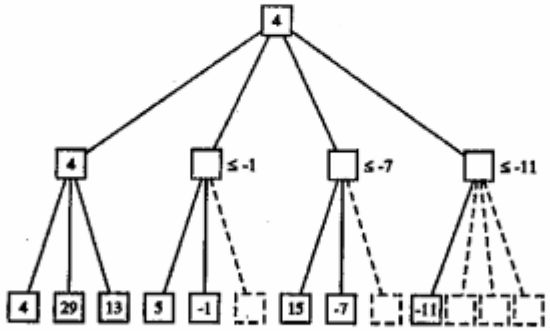

Figure 2: Alpha-Beta pruning for the same tree as Figure 1. The tree is generated in left-to-right order. As soon as the score -1 is computed, we immediately have a bound on the level above (<= -1) which is below the score of the +4 subtree. A cutoff occurs, in that no more descendents of the (<= -1) node need to be searched.

since we are taking the minimum of the scores +5, -1, ... we immediately have a bound on the scores of this subtree—we know the score will be no larger than -1. Since we are taking the maximum at the next level up (the root of the tree) and we already have a line of play better than -1 (namely, the +4 subtree), we need not explore this second subtree any further. Pruning occurs, as denoted by the dashed branch of the second subtree. The process continues through the rest of the subtrees.

The effectiveness of the pruning depends crucially on move ordering. If the best line of play is searched first, then all other branches will prune rapidly.

2.2 Iterative Deepening

Tournament chess is played under a strict time control, and a program must make decisions about how much time to use for each move. Most chess programs do not set out to search to a fixed depth, but use a technique called iterative deepening. This means a program does a depth 2 search, then a depth 3 search, then a depth 4 search, and so on until the allotted time has run out. When the time is up, the program returns the move it currently thinks is best.

Iterative deepening has the additional advantage of facilitating move ordering. The program knows which move was best at the previous level of iterative deepening, and it searches this principal variation first at each new level. The extra time spent searching early levels is more than repaid by the gain due to accurate move ordering.

2.3 The Hashtable

During the tree search, the same board position may occur several times. There are two reasons for this. The first is transposition, or the fact that the same board position can be reached by different sequences of moves. The second reason is iterative deepening—the same position will be reached in the depth 2 search, the depth 3 search, etc. The hashtable is a way of storing information about positions which have already been searched; if the same position is reached again the search can be sped up or eliminated entirely by using this information. As discussed in (Ebeling 85), proper use of the hashtable can effectively give near-perfect move ordering and hence, very efficient pruning.

3 PARALLEL COMPUTER CHESS

3.1 The Hardware

Our program is implemented on an NCUBE/10 system. This is an MIMD (multiple instruction stream, multiple data stream) multicomputer, with each node consisting of a custom VLSI processor running at 7 MHz, 512 kbytes of memory, and on-chip communication channels. There is no shared memory—processors communicate by message-passing. The nodes are connected as a hypercube but the VERTEX message-passing software (NCUBE 1987) gives the illusion of full connectivity.

Users communicate with a "host" or front-end computer which is similar to an IBM PC/AT. Programs are written and compiled on the host, then downloaded to the array at runtime. The array does not communicate directly with the user but a host program must be written to manage this interface.

The NCUBE system at Caltech has 512 processors, but systems exist with as many as 1024 processors. Our program is written in C, with a small amount of assembly code.

Our program also runs on transputer arrays under the EXPRESS operating environment (ParaSoft 1988). We do not presently have access to a transputer array with more than 32 processors. We will soon be able to use a large transputer system; we will report in the future on the transputer performance of our program.

3.2 Remote Procedure Call

In addition to simple message-passing utilities, our program requires some support for interrupt-based communication between processors. In order to make our program portable, we had to base our interrupt-based communication on some simple protocol which we could then implement on each target machine. We chose a "remote procedure call" system.

In this system, any processor can cause a procedure to be called on some remote processor. The called procedure executes immediately on the remote processor, without the knowledge of any program already running on the remote processor. A remote procedure can be called with arguments.

We will see below how remote procedure call can be used to implement a shared hashtable.

3.3 Parallel Alpha-Beta Pruning

Some good chess programs do run in parallel (Schaeffer 1986)(Newborn 1985)(Marsland 1984), but before our work nobody had tried more than about 15 processors. We are interested in using hundreds or thousands of processors. This has forced us to squarely face all the issues of parallel chess--algorithms which work for a few processors do not necessarily scale up to hundreds of processors. An example of this is the occurrence of sequential bottlenecks in the control structure of the program. We have been very careful to keep the control of the program decentralized in order to avoid these bottlenecks.

The parallelism comes from searching different parts of the chess tree at the same time. Processors are organized in a hierarchy with one master processor controlling several teams, each submaster or "team captain" controlling several subteams, etc. The basic parallel operation consists of one master coming to a node in the chess tree, and assigning subtrees to his slaves in a

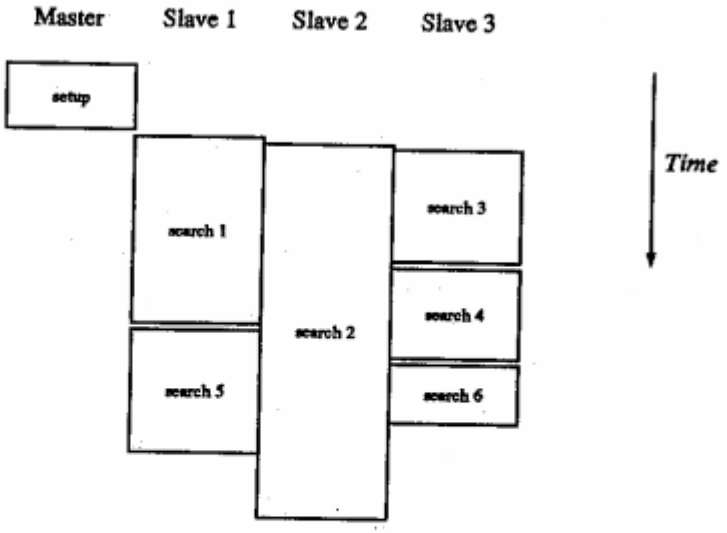

Figure 3: Slaves searching sub-trees in a self-scheduled manner. Suppose one of the searches, in this case, search 2, takes a long time. The advantage of the self-scheduling is that, while this search is proceeding in slave 2, the other slaves will have done all the remaining work. This very general technique works as long as the dynamic range of the computation times is not too large.

self-scheduled way. Figure 3 shows a timeline of how this might happen with three subteams. Self-scheduling by the slaves helps to load-balance the computation, as can be seen in the figure.

So far, we have defined what happens when a master processor reaches a node of the chess tree. Clearly, this process can be repeated recursively. That is, each subteam can split into sub-subteams at some lower level in the tree. This recursive splitting process, illustrated in Figure 4, allows large numbers of processors to come into play.

In conflict with this is the inherent sequential model of the standard alpha-beta algorithm. Pruning depends on fully searching one subtree in order to establish bounds (on the score) for the search of the next subtree. If one adheres to the standard algorithm in an overly strict manner, there may be little opportunity for parallelism. On the other hand, if one is too naive in the design of a parallel algorithm the situation is easily reached where the parallel program searches an impressive number of board positions per second, but still does not search much more deeply than a single processor running the alpha-beta algorithm. The point is that one should not simply split or "go parallel" at every opportunity—as we will see below it is sometimes better to leave processors idle for short periods of time and then do work at more opportune points in the chess tree.

3.4 Analysis of Alpha-Beta Pruning

The standard source on mathematical analysis of the alpha-beta algorithm is the paper by Knuth and Moore (Knuth 1975). This paper gives a complete analysis for perfectly ordered trees, and derives some results about randomly ordered trees. We will concern ourselves here with perfectly ordered trees, since real chess programs achieve almost-perfect move ordering.

In this context, perfect move ordering means that in any position, we always consider the best move first. Ordering of the rest of the moves does not matter. Knuth and Moore show that in a perfectly ordered tree, the nodes can be divided into three types, as illustrated by Figure 5. As in previous figures, nodes are assumed to be generated and searched in left to right order. The typing of the nodes is as follows. Type 1 nodes are on the "principal variation" The first child of a type 1 node is type 1 and the rest of the children are type 2. Children of type 2 nodes are type 3, and children of type 3 nodes are type 2.

How much parallelism is available at each node? The pruning of the perfectly ordered tree of Figure 5 offers a clue. By thinking through the alpha-beta procedure one notices the following pattern:

- all children of type 1 nodes are searched,
- only the first child of a type 2 node is searched—the rest are pruned, and,
- all children of type 3 nodes must be searched.

The implications of this for a parallel search are important. To efficiently search a perfectly ordered tree in parallel, one should perform the following algorithm.

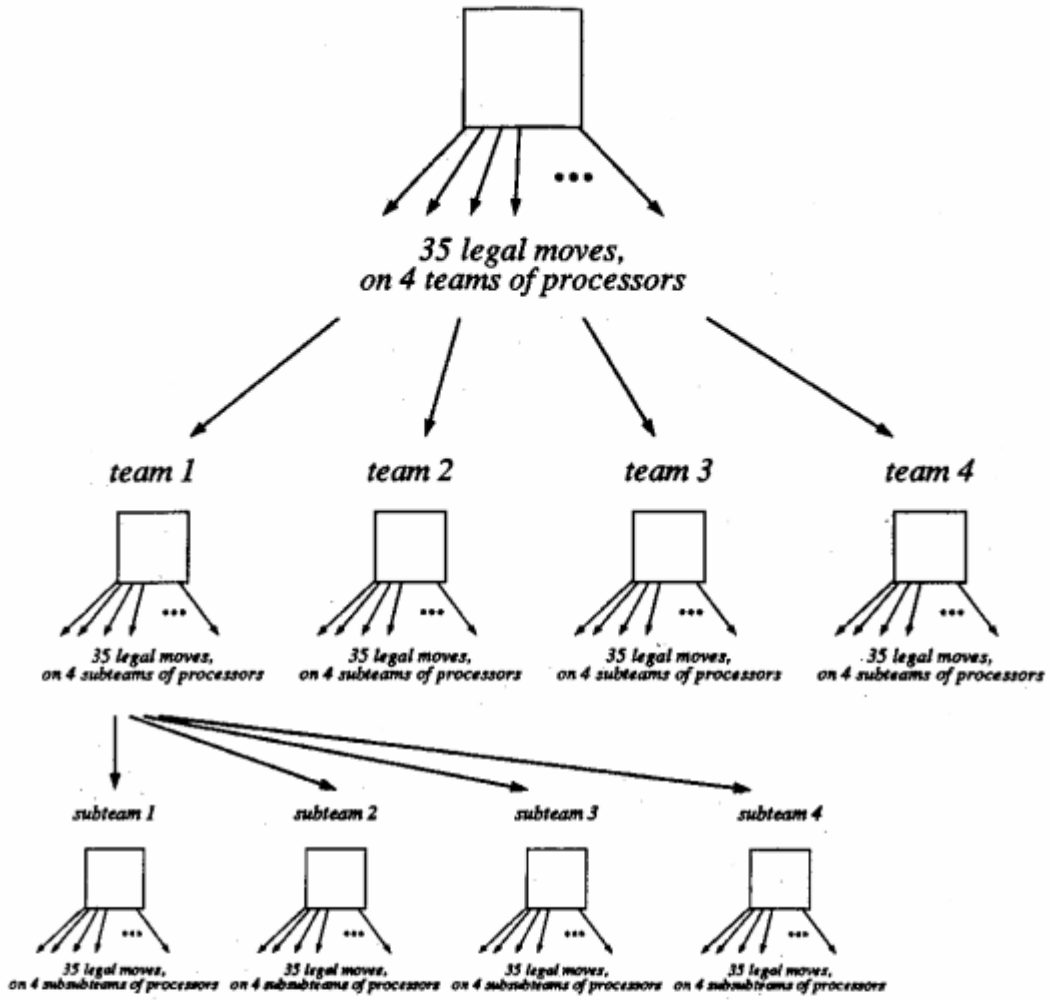

Figure 4: The splitting process of Figure 3 is now repeated, in a recursive fashion, down the chess tree to allow large numbers of processors to come into play. The top-most master has 4 slaves, which are each in turn an entire team of processors, and so on. This figure is only approximately accurate, however. As explained in the text, the splitting into parallel threads of computation is not done at every opportunity but is tightly controlled by the global hashtable.

- At type 1 nodes, the first child must be searched sequentially (in order to initialize the alpha-beta bounds), then the rest can be searched in parallel.
- At type 2 nodes there is no parallelism since only one child will be searched (time spent searching other children will be wasted).
- Type 3 nodes are fully parallel and all the children can be searched independently and simultaneously.

The key for perfectly ordered chess trees, then, is to stay sequential at type 2 nodes, and go parallel at type 3 nodes. In the non-perfectly ordered case, the clean distinction between node types breaks down, but is still approximately correct. In our program, the hashtable plays a role in deciding upon the node type. The following strategy is used by a master processor when reaching a node of the chess tree:

- Make an inquiry to the hashtable regarding this position. If the hashtable suggests a move, search it first, sequentially. In this context, "sequentially" means that the master takes her slaves with her down this line of play. This is to allow possible parallelism lower in the tree. If no move is suggested or the suggested move fails to cause an alpha-beta cutoff, search the remaining moves in parallel. That is, farm the work out to the slaves in a self-scheduled manner.

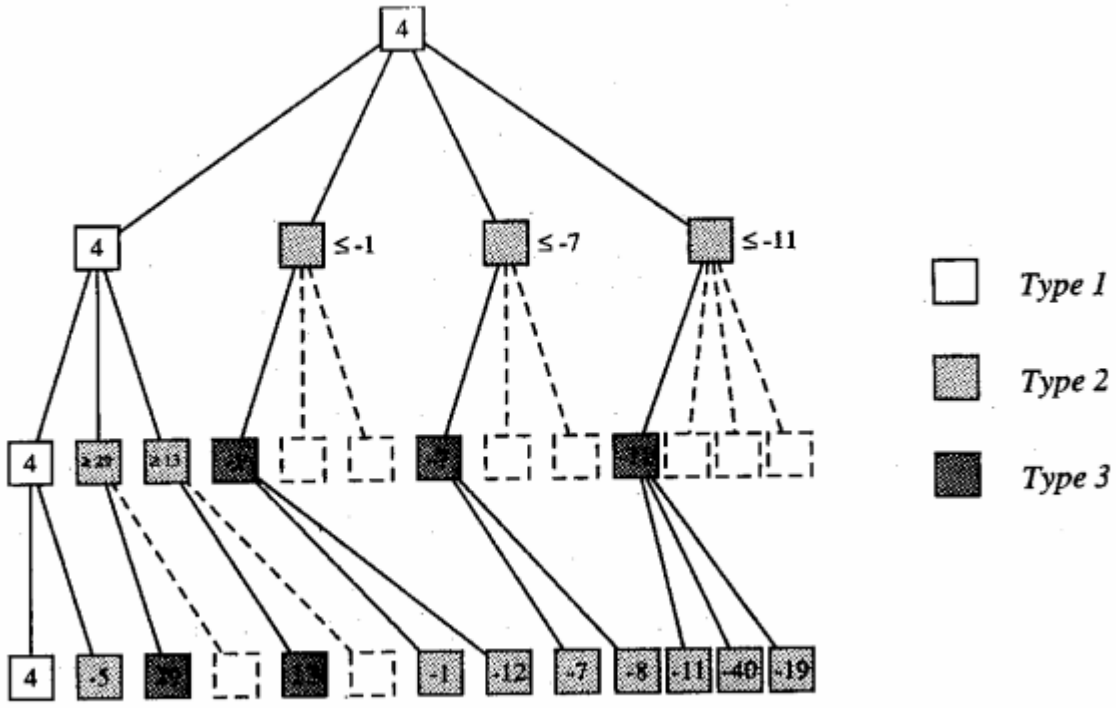

Figure 5: Pruning of a perfectly ordered tree. The tree of Figures 1 and 2 has been extended another ply, and also the move ordering has been re-arranged so that the best move is always searched first. By classifying the nodes into types as described in the text, the following pattern emerges: all children of type 1 and 3 nodes are searched, while only the first child of a type 2 node is searched.

This parallel algorithm is intuitively reasonable and also reduces to the correct strategy in the perfectly ordered case. In actual searches, we have observed the sharp classification of nodes into type 2 and type 3 at alternate levels of the chess tree.

3.5 Global Hashthable

The main value of the hashtable is as a refutation table near the root of the tree. This means that the hashtable carries information about suggested moves in various positions from one level of iterative deepening to the next. The central role of the hashtable in providing refutations and telling the program when to go parallel makes it clear that the hashtable must be shared between all processors. Local hashtables would not work since the complex, dynamically-changing organization of processors makes it very unlikely that a processor will search the same region of the tree in two successive levels of iterative deepening. A shared table is expensive on a distributed-memory machine, but in this case it is worthwhile.

Each processor contributes an equal amount of memory to the shared hashtable. The global hashfunction maps each chess position to a global slot number consisting of a processor ID and a local slot number. Remote memory is accessed by the remote procedure call mechanism. When a processor wants to access a remote entry, it does so by causing a remote procedure to be called; the remote procedure sends back a message containing the desired data. The processor which made the request waits until the answer comes back before proceeding.

Remote writing is also accomplished by remote procedure call. In this case, the remote procedure examines the newly generated hashtable entry and the entry previously occupying the desired slot, then decides which of the two entries is more likely to be useful. If the new entry is more useful it replaces the old.

Experiments show that the overhead associated with the global hashtable is only a few percent, which is a small price to pay for accurate move ordering.

3.6 Coordination of Processors

As indicated above, our program organizes processors in a multilevel hierarchy. A hierarchy with branching factor between 6 and 15 seems to work well in the middlegame; in the endgame a deeper, narrower hierarchy is

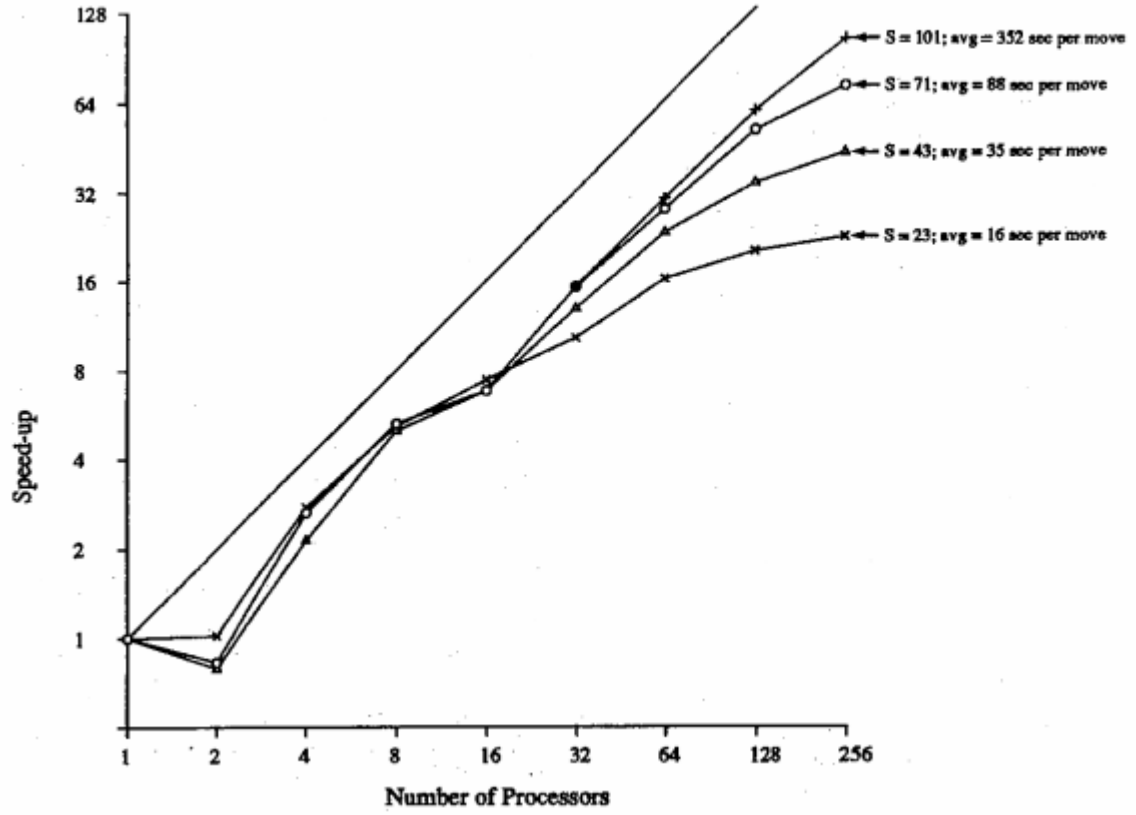

Figure 6: The speed-up of the parallel chess program as a function of machine size and search depth. The results are averaged over a representative test set of 24 chess positions. The speed-up increases dramatically with search depth, corresponding to the fact that there is more parallelism available in larger searches. The uppermost curve corresponds to tournament play--the program runs more than 100 times faster on 256 nodes as on a single NCUBE node when playing at tournament speed. Preliminary results indicate a speedup of at least 170 on a 512-processor machine.

appropriate. Control is decentralized; each processor knows only the identities of its slaves. A control program running on the host takes care of time allocation and communication with the user so that the array serves only as a "search engine."

When a processor's slaves are searching it does not join them but stays where it is, monitoring their progress and passing them relevant information as it becomes available. For instance, if one slave finishes its search and returns a value to the master, this may narrow the alpha-beta bounds of the master and consequently also narrow the bounds of the slaves. The master notifies the slaves whenever new information narrows their alpha-beta bounds. The master also tells the slaves to abort their searches if an alpha-beta cutoff occurs. Aborts and updates of alpha-beta bounds are implmented as remote procedure calls to ensure rapid propagation down through the hierarchy.

3.7 Load Balancing

As we explained in an earlier section, slaves get work from their masters in a self-scheduled way in order to achieve a simple type of load balancing. This turns out not to be enough, however, since the time to search two different sub-trees of the same depth can vary quite dramatically. A factor of 100 variation in search times is not unreasonable. Self-scheduling is helpless in such a situation. In these cases a single slave would have to grind out the long search, while the other slaves (and conceivably, the entire rest of the machine) would merely sit idle. Not only do the search times vary by a large factor, but this all happens at millisecond time scales. Any load balancing procedure will therefore need to be quite fast and simple.

For these reasons, our program has been taught to handle these "chess hot spots" intelligently. The master processors, besides just waiting for search answers, updating alpha-beta bounds, and so forth, also monitor what is going on with the slaves in terms of load balance. In particular, if some minimum number of slaves are idle and if there has been a search proceeding for some minimum amount of time, the master halts the

search in the slave containing the hot spot, reorganizes all his idle slaves into a large team, and restarts the search in this new team. This process is entirely local to this master and his slaves and happens recursively, at all levels of the processor tree.

The payoff of dynamic load balancing has been quite large—the program is approximately three times faster than it was without load balancing. We are convinced that the program is well load balanced and we are optimistic about the prospects for scaling to larger speedups on larger machines.

4 SPEEDUP MEASURMENTS

Speedup is defined as the ratio of sequential running time to parallel running time. We measure the speedup of our program by timing it directly with different numbers of processors on a standard suite of test searches. These searches are done from the even-numbered Bratko-Kopec positions (Bratko 1982), a well-known set of positions for testing chess programs. For each position we chose a depth of search which caused each search to take about the same amount of time on 256 processors. Our benchmark consists of doing two successive searches from each position and adding up the total search time for all twenty-four searches. By varying the depth of search we can control the average search time of each benchmark.

The speedups we measured are shown in Figure 6. Each curve corresponds to a different average search time. We find that speedup is a strong function of the time of the search (or equivalently, its depth). This result is a reflection of the fact that deeper search trees have more potential parallelism. Our main result is that at tournament speed (the uppermost curve of the figure), our program achieves a speedup of 101 out of a possible 256. Not shown in this figure is our latest result: a speedup conservatively estimated to be 170 on a 512 node machine. We believe these numbers can be improved somewhat through further tuning of the code and we are presently pursuing this.

The "double hump" shape of the curves is also understood: the location of the first dip, at 16 processors, is the location at which the chess tree would like the processor hierarchy to be a one-level hierarchy sometimes, a two-level hierarchy at other times. We always use a one-level hierarchy for 16 processors. Perhaps this is an indication that a more flexible processor allocation scheme could do better.

5 PRESENT STATUS

We currently have a robust program which has played in several tournaments against both people and computers. The program searches 50,000 positions per second and we estimate that it plays at a USCF rating strength of at least 2000. A version about ten times slower than the current program finished with a 2-2 record at the 1987 ACM North American Computer Chess Championship. In the next few months we will play the program in several human tournaments to gauge its strength against more varied competition.

ACKNOWLEDGEMENTS

Eric Umland played a key role in the early stages of this effort. We would like to thank Rod Morison, Ken Barish and Rob Fätland for helping with various phases of this project. We are grateful to Geoffrey Fox for his interest and encouragement .

This work is partially funded by the Department of Energy under grant number DE-FG03-85ER25009, the Program Manager of the Joint Tactical Fusion Office, the ESD Division of the USAF, and grants from IBM and Sandia.

REFERENCES

(Bratko 1982) I. Bratko and D. Kopec, "A Test for Comparison of Human and Computer Performance in Chess," in M. Clarke (ed.), "Advances in Computer Chess III", Pergamon Press, Oxford, 1982, pp. 31-56.

(Ebeling 1985) C. Ebeling, "All the Right Moves: A VLSI Architecture for Chess," MIT Press, Cambridge, 1985.

(Frey 1983) P.W. Frey (ed.), "Chess Skill in Man and Machine," Springer-Verlag, New York, 1983.

(Fox 1988) G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, and D.W. Walker, "Solving Problems on Concurrent Processors," Prentice Hall, New Jersey 1988.

(Hey 1988) A.J.G. Hey and D.A. Nicole, "The ESPRIT Reconfigurable Transputer Processor," in proceedings of IEE Seminar "The Design and Application of Parallel Digital Processors," Lisbon 1988.

(Knuth 1975) D.E. Knuth and R.W. Moore, "An Analysis of Alpha-Beta Pruning," Artificial Intelligence, 6 (1975), pp. 293-326.

(Newborn 1985) M. Newborn, "A Parallel Search Chess Program," Proceedings of the ACM Annual Conference, ACM, New York, 1985, pp. 272-277.

(Marsland 1984) T.A. Marsland and F. Popowich, "Parallel Game-tree Search," Technical Report TR-85-1, Department of Computing Science, University of Alberta, 1984.

(Marsland 1987) T.A. Marsland, "Computer Chess Methods," in Shapiro, "Encyclopedia of Artificial Intelligence," John Wiley and Sons, New York, 1987.

(NCUBE 1987) NCUBE Corp., "NCUBE Users Handbook," October 1987.

(Newborn 1985) M. Newborn, "A Parallel Search Chess Program," Proceedings of the ACM Annual Conference, ACM, New York, 1985, pp. 272-277.

(ParaSoft 1988) ParaSoft Corp., "EXPRESS: A Commu-

nication Environment for Parallel Computers," Mission Viejo, CA, 1988.

(Schaeffer 1986) J. Schaeffer, "A Multiprocessor Chess Program," proceedings of ACM-IEEE Fall Joint Computer Conference, 1986.

(Wallace 1988) D.J. Wallace, "The Edinburgh Concurrent Supercomputer Project," in proceedings of IEE Seminar "The Design and Application of Parallel Digital Processors," Lisbon 1988.