An Application Domain Specific Language for Describing Board Games [*]

John W. Romein Henri E. Bal Dick Grune

Vrije Universiteit
Department of Mathematics and Computer Science
Amsterdam, The Netherlands

{*john,bal,dick*}*@cs.vu.nl*

Abstract *Multigame is an implicitly-parallel, domain-specific language for describing board games. The Multigame system automatically exploits parallelism (by searching game trees in parallel), without any involvement from the programmer. The paper describes the language and its implementation on a Myrinet-based distributed system. It also gives performance results for Multigame applications.*

Keywords: application oriented language, parallel, distributed, game-tree search

1 Introduction

Although many high-level parallel programming systems have been designed, writing explicitly parallel programs still is a difficult task. On the other hand, extracting parallelism automatically from a sequential program is extremely hard for the compiler and has so far mainly been successful for applications with regular parallelism. An interesting alternative is to automatically exploit parallelism in programs written in *application domain specific* languages. For such languages, the implementation (compiler and run-time system) can use knowledge about the application domain to exploit parallelism, without any involvement from the programmer. Such languages thus trade general-purposeness for ease of programming and implicit parallelism.

In our research, we study the design and parallel implementation of such application domain specific languages. As a first step, we have designed a language (called *Multigame*) for describing board games. We have chosen this domain because it is well-defined and a large amount of literature already exists about parallel game-tree search. Also, game-tree search is known to be hard to parallelize efficiently [18], making parallelization of Multigame programs a challenging task. In this paper, we describe the design and parallel implementation of Multigame.

The outline of this paper is as follows. In Section 2, we describe the Multigame language. Next, in Section 3 we give some background information about sequential and parallel game tree search. In Section 4 the Multigame system is described. Section 5 gives initial performance numbers, and in Section 6 we describe future work and draw conclusions.

2 Multigame

Multigame is an application domain oriented language for describing board games. To create a manageable problem, we designed a language that is solely intended for board games, so card games (like bridge) and computer-games (like

[*]This research is supported by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

Tetris and Doom) cannot be expressed. Boards are assumed to be rectangular, although it is possible to fold a three-dimensional "board" (e.g., Rubik's cube) to a two-dimensional representation. The game is either played by one player (e.g. the 15-puzzle) or two players (e.g. chess, checkers, connect-4 and tic-tac-toe). Furthermore, the game must have perfect information, so a game like stratego is excluded because one cannot see the identity of the opponent's pieces. Finally, we exclude games that depend on randomness (e.g., backgammon cannot be expressed, because it uses a die).

A Multigame program specifies the rules of a game in a formal way. It first contains declarations that describe the size of the board and the pieces used. The rest of the program describes the legal moves, using a combination of the Logo- and Prolog programming paradigms.

```
knight_move =
    find own knight,
    pickup,
    orthogonal, step,
    either rotate 45 or rotate -45, step,
    not points at own piece, putdown.
```

Figure 1: Example knight move in *chess*

Instead of giving a formal description of the language, we give an intuitive example of a fragment of the well-known game of chess. Figure 1 shows how a knight move can be expressed in Multigame. We assume that the piece names have already been declared correctly. Briefly, the function finds a knight of the player who is to move, picks it up, and takes one step in either the North, East, South, or West direction; next, it changes direction 45 degrees clockwise or counterclockwise, takes another step, and then, if the target field is not occupied by a piece of the same player, puts the knight down.

The language has implicit knowledge about general properties of game playing. For example, if the knight moves outside the edges of the board, the move fails and will no longer be considered. Also, if a statement specifies to place a piece onto a field and the field happens to be occupied by another piece, the original piece will be removed from the board. This is the way to implement capturing moves.

Many statements introduce multiple continuations. For example, if a player still has both knights, the `find own knight` statement results in two possible actions. Likewise, the `orthogonal` statement continues in four different directions. This reflects the similarity to the Prolog programming style. In fact, the code that is generated by the Multigame compiler does full backtracking for these statements.

The language has several other features that are not discussed here. It is general enough to allow many games to be expressed. Most importantly, the language is completely free from any statement that expresses parallelism, work- or data distribution, communication, synchronization, or deadlock prevention, which greatly simplifies the programmer's task.

3 Game tree search

One of the most important aspects of our work is how to efficiently implement Multigame programs on parallel systems. Before we discuss the Multigame implementation, we first outline some general issues of computer game playing. In particular, we discuss sequential and parallel game-tree search.

3.1 Sequential game-tree search

Whenever a game playing program is to make a move, the program searches a so called game-tree to find the best move from the given board position. For one-player games (e.g. the 15-puzzle) the goal is to find the shortest sequence of moves from a problem instance to a target solution. For two-player games, the move to the best obtainable position is searched under the assumption that the opponent replies with the best countermove.

Game-tree search is essentially repeatedly thinking moves ahead. Each node in the tree corresponds to a position; each edge to a child corresponds to a possible move from that position. Unfortunately, most games of interest generate trees that are too large to be fully searched. Although good search algorithms will avoid searching uninteresting parts of the tree, we are still forced to limit the number of moves thought ahead in some way.

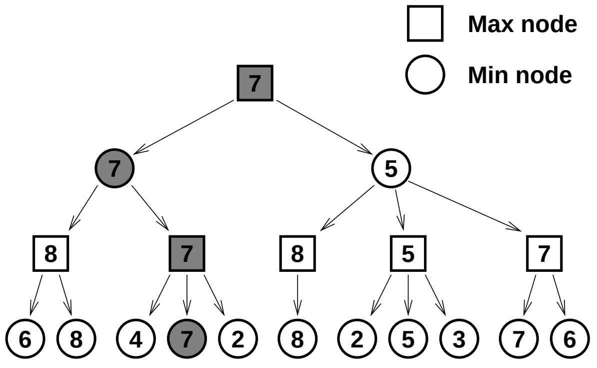

Figure 2: Example game-tree

Figure 2 shows a two-person minimax tree of depth three. The numbers in the leaf nodes of the tree are static evaluation values; a high value indicates that the position is advantageous for the white player. In the interior nodes, white maximizes its children's values, while black minimizes them. From the root position, white decides to do the first move, expecting that black will continue the gray shaded line of play.

Good search algorithms, like alpha-beta [9], MTD(f) [14] and ProofNumber search [1] (for two-player games) and IDA* [10] (for one-player games) do not search all children of a node, but prune sub-trees that are irrelevant for the decision of which move to make from the root position. This can lead up to order square reductions of the number of nodes that are evaluated.

For example, the alpha-beta algorithm prunes children which are proofed to be uninteresting. A node is searched within a so-called (α-β) window, which states that the value of the node is only interesting if it lies within the window. If the value is smaller than α, the position is too bad because the algorithm already found a way to reach a better position; if the value is greater than β, the position is too good because the algorithm has already seen that the opponent has a way to avoid this position. The root node is searched with bounds $(-\infty, \infty)$. The algorithm recursively searches the children of a node one by one. The results of the search of child n might tighten the search window that is used for child $n+1$. In the case that $\alpha >= \beta$ it makes the search of the rest of the children unnecessary, pruning large parts of the search tree.

The search algorithms can use a variety of game-independent heuristics to guide the search, such that most promising children are searched first. Examples of important and well known heuristics are *iterative deepening* [17], *transposition tables* [7], and the *history heuristic* [15].

3.2 Parallel game-tree search

Usually a game playing program can improve its quality by searching the game-tree deeper. Because of the exponential growth of the tree, searching one ply deeper takes an order of magnitude more time. Having multiple processors cooperate is a natural way to reduce the total search time.

One of the most important aspects in our research is to investigate how we can exploit parallelism in Multigame programs in a transparent way. We assume that Multigame programs will be run on distributed-memory multicomputers (e.g. a collection of workstations), so we try to exploit coarse-grained parallelism. The current system can also generate code for shared-memory systems.

Parallelism can be found in several components of the game playing program. However, most parallelism is too fine-grained. Only the search engine provides a reasonable amount of coarse-grained parallelism, which can be exploited by having multiple processors searching multiple children of a node simultaneously. The sequential search algorithms are modified to do parallel search. This is easy for IDA*, but it is hard to parallelize alpha-beta [6] and ProofNumber search efficiently.

There are several ways to parallelize alpha-beta [2]. The problem with parallel alpha-beta is that the search window (α-β) of child $n + 1$ depends on the search result of child n, so alpha-beta search (and MTD(f), which is based on alpha-beta) is inherently sequential. When child $n + 1$ is searched simultaneously with child n, the search window for $n+1$ must be guessed conservatively.

Another problem is how to efficiently implement the heuristics on a distributed-memory parallel system. A transposition table, for example, essentially is a shared data structure that is accessed (read and written) very frequently, by all processors. Implementing such a data structure on a system without shared memory is very difficult.

In general, parallel game-tree search suffers from several kinds of overhead.

- *Search overhead* is caused by the fact that the parallel version of a search algorithm may search more nodes than its sequential variant.
- *Synchronization overhead* occurs when a processor that computes the value of a node has to wait until the values of its children have been computed.
- *Communication overhead* results from the latency of messages that are sent from one processor to another and the time to process them.
- *Load imbalance* might lead to an uneven distribution of the work over processors.

An important problem is that each attempt to lower one kind of overhead generally results in an increase of the other kinds of overhead.

4 The Multigame system

In this section we describe the implementation of Multigame. We first give a general overview of the Multigame system. Next, we discuss parallel search in Multigame in more detail. Finally, we look at the implementation of two important heuristics: transposition tables and the history heuristic.

4.1 Overview of the Multigame system

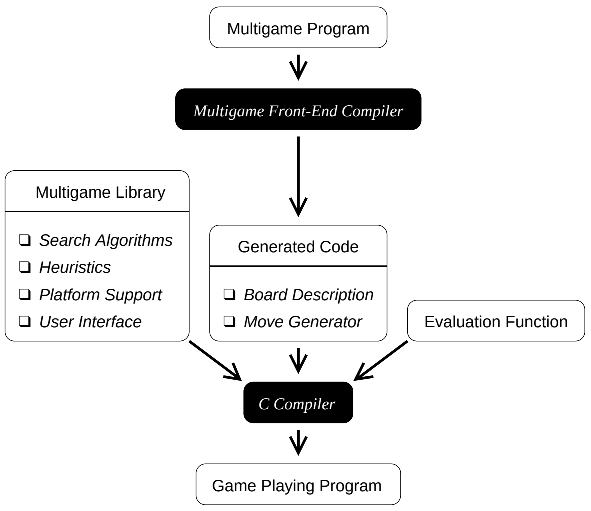

Figure 3: Structure of the Multigame system.

The Multigame system generates a parallel game playing program from a Multigame specification. Figure 3 shows how all components fit together. The *front-end compiler* generates game-specific code. The compiler uses a *library* that provides functions that are common to several or all games. The generated code is ANSI-C, which is portable and efficient[1].

The primary task of the front-end compiler is to generate a move-generator from a Multigame program. The move generator accepts the current board position and computes all positions that can be reached by doing a legal move. The move generator runs sequentially, because the parallelism within the move generator is too fine-grained to be efficiently exploited on off-the-shelf hardware.

The library provides components that are common to several games. The programmer selects which components must be used as com-

[1]Optionally, extensions of the GNU C compiler can be used for even more efficiency, such as controlled register allocation and indivisible read-modify-write instructions.

mand line options when invoking the Multigame compiler.

One of the most important components in the library is the set of search strategies, from which the programmer can select one. Since the search algorithms used by Multigame are part of the library, the knowledge about parallel execution is thus implemented in the library. This is in contrast to parallelizing compilers, which must analyze the program being compiled to discover parallelism.

The Multigame library also contains the heuristics that are mostly game-independent, such as iterative deepening, transposition tables, and history tables. The programmer selects which heuristics are used, and specifies options like table size and update strategies.

Other components in the library include specific code to support a variety of platforms, including Solaris, Amoeba, and the Panda [4] portability layer. It also implements a generic user interface, which looks like a shell and accepts commands like "set search depth to 17", "give me a hint", "move king from `a4` to `b4`", and "print transposition table statistics".

An evaluation function is highly game-dependent. Ideally we would like the front-end compiler to generate appropriate evaluation functions from the rules of the game, but this is very hard to do [13] and is not part of our research. Therefore, the programmer can optionally provide a hand-written evaluation function for a specific game to improve the quality of the playing program. We have ported evaluation functions for checkers and Othello. The checkers evaluation function is ported from Chinook, the current man-machine world champion [16]. The Othello evaluation function is ported from Aïda (a program developed at the University of Leiden). For the 15-puzzle we use the Manhattan distance function with the linear conflict heuristic [8] and the last move heuristic [11].

4.2 Parallel search in Multigame

We have implemented several parallel search algorithms, including alpha-beta search, IDA*, and MTD(f). (We also implemented sequential ProofNumber search [1].) As explained in Section 3.2, the problem with parallel game tree search is how to deal with the different forms of overhead (search overhead, synchronization overhead, communication overhead, and load imbalance). We tried to find a good balance between all overheads and also implemented important optimizations to minimize the search time of a game-tree.

We use a dynamic load-balancing strategy based on work-stealing. We implemented a distributed job-queue to transfer work from one to another processor. Each processor has its own local job-queue. If a processor wants to post a job, it puts it in its own queue. If a processor wants to grab a job, it first looks in its own queue, and if the queue is empty, it sends a request-for-work message to another processor. If the other processor does not have work in its job-queue, yet another processor is tried in round-robin order, until there is a processor that has work. If all processors have been polled unsuccessfully, the processor waits a small amount of time to avoid flooding the network with request-for-work messages before a new polling round is started. A job that has been put into the queue can also be cancelled whenever it turns out to be not of interest any more.

To reduce the search overhead, we use the Youngest Brothers Wait algorithm, described in [5]. With this algorithm, the children of a node may be evaluated in parallel only if its first successor has been evaluated completely. This forces a reasonable bound for α, so that the forked-off children are not searched with an unnecessarily wide (α-β) window.

An important optimization is to send updates of search arguments to jobs that have already been started. For example, in the parallel alpha-beta variant, we send an update message whenever the alpha-beta window for a grabbed job is narrowed. This turned out to be very hard to implement correctly, because the new bound has to be propagated throughout the subtree. Furthermore, all kinds of race conditions had to be dealt with.

Another important optimizations is that a processor does not wait (and become idle) when the result of a parent node cannot be determined while another processor is still busy evaluating one of the children. Instead, it leaves the node behind and grabs another job. This almost eliminates the synchronization overhead, but increases the search overhead because the processor eagerly grabs jobs that might have been cancelled if it would have waited for the child to finish.

4.3 The transposition table

The most important heuristic that speeds up game-tree search is the transposition table. Actually, the name "game-tree" is a misnomer, because the search space is a graph rather than a tree. Many games allow the same position to be reached by different sequences of moves. In chess, for example, the opening moves `d4–Nf6–e3` yield the same position as `e3–Nf6–d4`. The transposition table [20] is a well-known technique to detect and exploit such transpositions.

The transposition table is a large hash-table that caches the results of sub-trees that have already been searched. Each board position is hashed to a *signature* which has a typical size of 64 bits. The signature is subsequently mapped to an index of the table. Each entry in the table contains a tag (to check signature values), and some search-strategy specific fields, such as the result of the position that has been searched (in two-player games this is the minimax value), the best move from that position, and the depth to which the position has been searched.

To illustrate how transposition tables work, assume a program needs to search a given position for n plies (half moves). The program first looks in the transposition table. If it finds an entry whose tag matches the signature, the position has already been searched[2]. If the position was searched at least n plies, the value in the table is used. The entire subtree beneath the position thus is not searched again, which greatly reduces the search time. If the position was searched, but to less than n plies, the information in the table (the "best move" field) is used as a hint for move-ordering, which also reduces the search time.

[2]The mapping between position and signature is not perfect, so there is a chance that the entry corresponds to *another* position with the same signature, albeit very small.

Each time after a position has been searched, the results of the search are stored in the transposition table. We use 2-way associative tables with a replacement scheme described in [3] in the case that a new entry competes with an old one for a place in the table.

We have put much effort in the implementation of a distributed transposition table, because the performance of many games critically depends on the efficiency of the implementation. We implemented three different strategies for a shared transposition table: *partitioned*, *replicated*, and *private*.

- In the *partitioned* implementation, each processor holds part of the transposition table. Whenever a process wants to read or write an entry that is not stored on the same processor, a message is sent to the processor that holds the entry. The communication overhead is large, especially since a read operation requires fully synchronous (blocking) communication.

- In the *replicated* version, all processors keep the entire table in main memory. Processors synchronize their tables by frequently broadcasting new updates to other processors. Read operations are done locally. Write operations are sent to all processors. A disadvantage of a replicated table is that it can hold fewer entries than a partitioned table, since each entry is stored on all machines. Also, the total number of messages that needs to be processed by all processors will be larger than for a partitioned table (unless the number of updates is much smaller than the number of lookups).

- In the *private* version, each processor has its own transposition table: all reads and writes are done locally. There is no communication overhead, but the usefulness of the transposition table is limited, since a processor will never get information about positions searched by other processors, resulting in a larger search overhead.

Unfortunately, the most efficient implementation depends on many (hardware and software) factors. The ratio between processor speed and network speed, the available amount of memory for the table, and the number of CPUs are important factors. Also, the kind of game that is being played has some impact. Some games (e.g. checkers) build trees with many transpositions, thus making the communication overhead of a shared table smaller than the search overhead of a private table. Another game-dependent factor is the time spent in the evaluation function (an intricate evaluation function yields a high computation/communication ratio). A high ratio between transposition table reads and writes may favor the replicated implementation. On the other hand, a deep search which requires many entries to be stored favors the partitioned implementation, because it can store many more entries than the replicated implementation, given the same amount of memory.

Yet another factor is the interrupt latency time, which can become dominant. We observed that polling for messages instead of sending an interrupt upon message arrival can greatly improve performance under some circumstances [12]. One way to avoid interrupts for transposition table reads and writes is to use dedicated transposition table servers. In this case, the transposition table is stored on only a few processors, which do not participate in the tree search. These processors can poll for read and write requests and return replies immediately.

4.4 The history heuristic

Another heuristic that is used for move ordering is the history heuristic. The heuristic takes advantage of the fact that the best move from one position is often the best move from a similar position. The history table isolates the moving piece from the surrounding pieces and records how frequently a possible move is the best move. When a position is searched, its children are sorted in descending order of frequency. This way, the most promising move is searched first.

Sharing a history table on a distributed system is less critical for the performance than sharing a transposition table. Each processor keeps its own history table, and once per second we synchronize these tables by sending the changes of the last second to other processors. This does not involve much communication.

5 Performance results

We have done several performance measurements on Multigame. Our primary development platform is the Amoeba system, which consists of 50 MHz MicroSPARC processors connected by a Myrinet network. Myrinet is a gigabit switched network. On Myrinet, we use fast message passing software [12], with a 50 μs latency for unicast messages. The message passing software is based on the Illinois Fast Messages library which was extended with multicast primitives and interrupt driven message delivery [19], and is integrated with the thread scheduler. Sending messages is done from user-space, avoiding expensive kernel boundary crossings.

In the following sections, we show performance results for the 15-puzzle, checkers, and Othello.

5.1 15-puzzle performance

Table 1 shows the execution times for analyzing a random position of the 15-puzzle (position 66 from Korf's test set [10]). Normally, parallel IDA* search has very nondeterministic search times: due to the random work distribution, it is possible that one processor finds the solution almost immediately, so that the execution terminates at a random

#procs	time	speedup	#evals
1	1850	1.00	20567802
2	967	1.91	20773619
4	517	3.58	21015093
8	290	6.38	21109169

Table 1: Results for the 15-puzzle. Time is in seconds. #evals shows the number of calls to the evaluation function.

moment. It is even possible to obtain a superlinear speedup in this way. For our measurements, we avoid this behavior by reading the search timer *before* the last iteration begins. This way, the tree that is searched (or rather the graph) is almost constant. The fact that 8 processors search a few more nodes than 1 processor is explained by the fact that some transpositions are searched by multiple processors, despite the shared transposition table. We use a replicated transposition table with 2^{20} entries. Without the shared table, the search overhead is much worse, and does not compensate for the reduced communication overhead.

The performance numbers in Table 1 show that we obtain good speedups. The reason that we obtain less than 100% efficiency is that we use a shared transposition table which causes communication. However, without the table the sequential version performs much worse, and we feel that we should compare the speedups to an optimal sequential version.

5.2 Checkers performance

We also measured the performance of the checkers application. We use the MTD(f) algorithm as search strategy, because it outperforms alpha-beta. The trees built by checkers have many transpositions, so the application benefits heavily from the shared (replicated) transposition table. The history heuristic is also very useful for this application. Compared to the 15-puzzle, the evaluation function we use for checkers is much more intricate, so a large fraction of the run-time is spent in the evaluation function.

We repeatedly measured the execution time for one board position on multiple processors and noticed that the execution times differed from run to run. Due to the random work distribution and speculative search, a non-deterministic number of nodes is searched, hence the execution times differ. This problem can be worked around by doing a large number of runs and averaging the execution times. However, we also noticed that some test positions result in better average speedups than other positions. We therefore repeatedly took one of the twenty test position from Plaat's test set [14] at random, measured the search times and the number of times the evaluation function was called on 1, 2, 4, and 8 processors, computed the speedup and search overhead for this test position, and finally averaged all speedups and search overheads.

#procs	speedup	search overhead
1	1.00	0%
2	1.55	27%
4	2.33	55%
8	3.25	91%

Table 2: Results for checkers.

Table 2 show that we do obtain speedups, but not as good as with the 15-puzzle. The last column shows that we suffer from a substantial search overhead; on 8 processors 91% more evaluations are performed than on 1 processor. As is explained in Section 3.2, the search overhead is inherent to two-player search algorithms like MTD(f) due to the speculative way work is stolen.

5.3 Othello performance

We also did performance measurements on Othello, again using the MTD(f) algorithm and using the same heuristics as were used with checkers. However, some properties of Othello are different from checkers. Where checkers builds trees with many transpositions, transpositions in Othello are rare (although the transposition table is still useful for move ordering). Furthermore, checkers trees are narrow and deep, while trees built by Othello are much wider. Naturally, we use another evaluation function for Othello.

#procs	speedup	search overhead
1	1.00	0%
2	1.44	35%
4	2.18	71%
8	2.74	141%

Table 3: Results for Othello.

In Table 3 we show the average speedups and search overheads for Othello. Again, the test positions are randomly chosen from Plaat's Othello test set [14]. The speedup is somewhat worse than for checkers, because the search overhead is higher (141% vs. 95%).

6 Conclusions and future work

In this paper we described Multigame, an application domain specific language used for playing board games. A Multigame program formally expresses the rules of a game. From these rules the Multigame compiler automatically generates a parallel program that can play the game, using the knowledge it has about its application domain. The programmer's task is simplified because he or she is freed from thinking about parallel issues like communication, synchronization, work- and data distribution, and deadlock prevention. Instead, the programmer only has to choose a few parameters, which influence the (parallel) behavior of the program.

Performance measurements show that we obtain good performance. We are in the process of porting Multigame to a cluster with 200 MHz Pentium Pro processors connected with a fast Myrinet switching network. This cluster uses much faster processors than the 50 MHz MicroSPARC processors we use now, and will finally consist of 64 nodes, being a much more modern and interesting platform than our present cluster.

We are also working on a new, more efficient distributed transposition table by customizing the network coprocessors on the Myrinet network interfaces. The idea is to run application specific code on the network processor to relieve the host processors. This way, the network processor services transposition table accesses from other processors without interrupting the host processor. Initial results are promising.

Future research will focus on other application domain oriented languages. We recognized that it was hard to implement distributed search strategies efficiently and correctly, and we will investigate whether programming distributed search strategies is easier with an application domain specific language that is especially designed for this purpose.

Acknowledgments

Aske Plaat provided useful comments on a draft version of this paper. Arnold Geels devised Gamespeak, a predecessor of the Multigame language. We thank Jonathan Schaeffer for providing us with the sources of Chinook and Jacco Gnodde and Dennis Breuker for providing us with the sources of Aïda.

References

[1] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Rijksuniversiteit Limburg, Maastricht, the Netherlands, September 1994.

[2] H.E. Bal and R. van Renesse. A Summary of Parallel Alpha-Beta Search Results. *ICCA Journal*, 9(3):146–149, 1986.

[3] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement Schemes and Two-Level Tables. *ICCA Journal*, 19(3):175–180, 1996.

[4] H.E. Bal e.a. Orca: a Portable User-Level Shared Object System. Technical Report IR-408, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, July 1996.

[5] R. Feldmann. *Game Tree Search on Massively Parallel Systems.* PhD thesis, University of Paderborn, August 1993.

[6] R.A. Finkel and J. Fishburn. Parallelism in Alpha-Beta Search. *Artificial Intelligence*, 19:89–106, 1982.

[7] R.D. Greenblat, D.E. Eastlake III, and S.D. Crocker. The Greenblat Chess Program. In *Proceedings of the Fall Joint Computing Conference*, pages 801–810, San Fransisco, 1967.

[8] O. Hansson, A. Mayer, and M. Yung. Criticizing Solutions to Relaxed Models yields Powerful Admissible Heuristics. *Information Sciences*, 63(3):207–227, 1992.

[9] D.E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[10] R.E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[11] R.E. Korf and L.A. Taylor. Finding Optimal Solutions to the Twenty-Four Puzzle. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1202–1207. AAAI Press / MIT Press, August 1996.

[12] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating Polling, Interrupts, and Thread Management. In *Proceedings of Frontiers'96*, Annapolis, MD, October 1996.

[13] B.D. Pell. *Strategy Generation and Evaluation for Meta-Game Playing.* PhD thesis, University of Cambridge, August 1993.

[14] A. Plaat. *Research, Re: Search & Re-Search.* PhD thesis, Erasmus Universiteit Rotterdam, the Netherlands, June 1996.

[15] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[16] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53:273–289, 1992.

[17] D.J. Slate and L.R. Atkin. CHESS 4.5 — The Northwestern University Chess Program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer Verlag, 1977.

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

[19] K. Verstoep, K.G. Langendoen, and H.E. Bal. Efficient Reliable Multicast on Myrinet. In *1996 Int. Conf. on Parallel Processing (Vol. III)*, pages 156–165, Bloomingdale, IL, August 1996.

[20] A.L. Zobrist. A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, University of Wisconsin, Madison, 1970. Reprinted in: *ICCA Journal*, 13(2):69–73, 1990.