PARALLEL SEARCH OF GAME TREES

T. A. MARSLAND

M. S. CAMPBELL

and

A. L. RIVERA

Technical Report TR80-7
November 1980

Department of Computing Science

The University of Alberta

EDMONTON, Canada, T6G 2H1

PARALLEL SEARCH OF GAME TREES

T. A. MARSLAND

M. S. CAMPBELL

and

A. L. RIVERA

Technical Report TR80-7
November 1980

Department of Computing Science

The University of Alberta

EDMONTON, Canada, T6G 2H1

ABSTRACT

With the increasing availability and computing power of contemporary microprocessors there is a growing trend towards the construction of unique computer architectures for specialized applications. This is especially true for compute-bound tasks requiring minimal amounts of expensive storage, particularly if the problem may be partitioned into autonomous functions for parallel execution. For some problems, such as the tree structured decision-making process, this partitioning is easily done.

A classical tree problem is the minimax evaluation of zero-sum games. Although the choices may appear straightforward, there are many important design decisions relating not only to the use of efficiency refinements such as the alpha-beta algorithm, transposition tables and iterative deepening, but also to fundamental interprocessor communication problems involving the use of shared memory for message passing, and choice of hardware configuration. This report therefore serves as a working design document for the construction of multiprocessor systems for performing fast alpha-beta searches, especially as related to the computer chess problem.

1. INTRODUCTION

The alpha-beta algorithm is a well known procedure for computing the minimax value of tree structures whose leaf (terminal node) values are expressed by simple numeric quantities. Because of its efficiency, the algorithm is commonly employed to search lookahead trees for such games as chess and checkers. The application of parallel processing to the alpha-beta algorithm is also receiving attention. In this paper an examination is made of various alternatives in hardware, interprocessor communications, and searching methods, as part of a proposed design for the implementation of parallel search.

The paper is essentially in two parts. It opens with a brief summary of the alpha-beta algorithm, and a discussion of the possible overheads involved in a parallel implementation of the algorithm. A number of searching methods suitable for a parallel system are reviewed, and some suggestions for variations and potential enhancements are made. In the second part, the characteristics of two contemporary microprocessors are examined to determine their suitability for the given application. In particular, interprocessor communication schemes from both the hardware and organizational points of view are studied. Finally, based on our assessment, the preferred system architecture is described and our conclusions, with directions for future research, are presented.

2. THE ALPHA-BETA ALGORITHM

A complete description of the minimax alpha-beta tree searching algorithm can be found elsewhere [Knuth75]. Rather than duplicate that work we will simply clarify some relevant facts and terminology used in our paper.

In two-person zero-sum games a tree of all possible sequences of moves can be envisioned, from either the initial or any intermediate position. Nodes in this so-called game tree correspond to game positions, while branches correspond to moves. All leaf nodes are terminal positions in the game (i.e. wins, draws or losses). All other nodes correspond to non-terminal (intermediate) positions.

Generally it is impractical to examine the entire tree to the terminal nodes, and so some arbitrary criteria are used to abbreviate the search. The simplest approach is to truncate the game tree at a fixed depth, and assign a numeric measure to estimate the relative goodness of the nodes at this depth. It is assumed that increasing the maximum depth improves the quality of the search result. Thus alpha-beta pruning is a useful optimization of the minimax algorithm, since it can allow smaller trees to be searched while still locating the "best" path to the terminal nodes. In a uniform game tree with fixed depth **d** and branching factor **b**, the minimax algorithm examines **b ** d** terminal nodes, while in the best case of alpha-beta, only

$$b ** \lceil d/2 \rceil + b ** \lfloor d/2 \rfloor - 1$$

terminal nodes are examined. Though a great improvement over straight minimaxing (reducing search to about the square root), even the best case of alpha-beta is still exponential with tree depth.

```
alphabeta(p, alpha, beta)
position p;
int alpha, beta;
{
      int n, v, i, s;
                           /*  determine successor positions  */
      n = generate(p);     /*  p.1 ... p.n and return number  */
                           /*  of moves as function value     */

      if (n == 0) return (f(p));
      v = alpha;
      for i = 1 until n
      {
            s = -alphabeta(p.i, -beta, -v);
            if (s > v) v = s;
            if (v >= beta) goto done;  /*  cutoff  */
      }
done: return(v);
}
```

Figure 1: Basic alpha-beta algorithm

The basic structure of the alpha-beta algorithm is very simple, as illustrated by the pseudo-C [Kern78] representation in the negamax [Knuth75] framework, Figure 1. The algorithm maintains two bounds, traditionally called alpha and beta, as it carries out a search. Alpha represents the score for the best alternative found so far for the first player, while beta is the best alternative value for the opponent. The interval enclosed by (alpha,beta) is referred to as the alpha-beta _window_. Since the evaluated score of the root position must be between alpha and beta, the window is often set to (-infinity,+infinity) to be sure of enclosing this (unknown) score.

Generally speaking, however, the narrower the initial window, the better the algorithm's performance. Hence there is some incentive for choosing an initial window which spans a suitable range about an expected value for the position. If the best score returned is less than alpha or greater than beta, then the search has _failed low_ or _failed high_ [Fish80] respectively. Thus the window must be expanded and the search repeated until the returned score falls within the window. For example, if the initial search fails high, then with a window of (beta,+infinity) the true score can be found in one more iteration (provided all the opponent's relevant continuations have been examined in the first pass).

3. PARALLEL SEARCH OVERHEADS

There are two major forms of overhead in a parallel alpha-beta search - interprocessor communication and search overhead. As will be seen, these two overheads tend to be inversely related, and a suitable balance between them must be found for practical systems.

3.1 _Search overheads_

Without alpha-beta pruning a concurrent implementation of a tree search would be a relatively trivial matter. Sub-trees could be explored independently with no loss of efficiency. However the power of the sequential alpha-beta algorithm is based on the fact that, at any given point in the search, all previously acquired information is used to minimize the tree size (i.e. to generate the narrowest possible guaranteed alpha-beta window). When different subtrees are explored concurrently, the best information is not necessarily available to every sub-tree search, with consequent reduction in the efficiency of the algorithm. Thus, one requirement for a a good parallel alpha-beta implementation is that it minimize the adverse effects of having tree information distributed over a number of processors.

3.2 <u>Communications overheads</u>

Another type of overhead in the parallel alpha-beta algorithm results from the necessity for interprocessor communication, in order to update alpha-beta windows across all processors. Whether this is carried out via a shared data structure (i.e. common memory), or message passing (with consequent sending and receiving costs), a delay is unavoidable. This delay would come from the execution of procedures which would either set semaphores, execute critical areas of monitor procedures or perform confirmation and checking for protocol procedures. There is also a tendency, if communications are minimized, to reduce the sharing of global tree information (using wider alpha-beta windows), perhaps lowering the efficiency of each processor.

4. BASIC METHODS OF PARALLEL ALPHA-BETA SEARCH

A number of schemes for introducing parallelism to the alpha-beta algorithm can be conceived. Our review of these methods will make the choices more clear, and will provide some suggestions for further research.

4.1 <u>Naive method</u>

One possible parallel implementation of the alpha-beta algorithm uses a static decomposition of the game-tree. This involves splitting the tree into sub-trees and assigning these sub-trees to different processors. The depth at which the division takes place will be referred to as the <u>common depth</u>. There are some serious drawbacks to such a scheme.

(i) A static decomposition of the tree with tasks of varying sizes is poorly suited to execution on a multiprocessor system. Once all the sub-trees at any common depth have been assigned to processors, then any processor returning from the search becomes idle until the last one completes.

For example, consider a 10-processor system and a 19-processor system, both exploring a tree with 20 subtrees. Though the 19 processors have almost double the computing power of the 10, both systems will take approximately the same time to complete the search. This is because in the 19 processor case the twentieth subtree can be explored by only one processor, leaving the others idle.

(2) The tradeoff between search and communications overheads is particularly unfavorable. In the naive method, interprocessor communication is exponential with common depth, which encourages minimization of the depth in order the keep the communication at a manageable level. However search overhead becomes excessive at low common depths, especially with a large number of processors. A striking example of this can be seen in the exploration of a typical chess game tree. A round number estimate of the branching factor is about 40, with alpha-beta pruning reducing the effective tree size to one with a branching factor of about 7. Thus, if 40 processors are used to conduct a search with common depth 1, the resultant speedup will be about 7 (equivalent to the effect of removing one ply from the tree) [Gil172]. The serious problems involved are illustrated by the fact that a 40-fold increase in computing power provides only a 7-fold speedup.

4.2 Baudet method

A second approach to parallel alpha-beta has been developed [Baud78], one which does not involve decomposition of the game-tree. Instead, the initial alpha-beta window is divided into subintervals, and each processor searches for a solution over a different subinterval. Since considerably smaller windows can be used, the individual searches usually proceed more quickly. Processors whose windows do not contain the actual score of the position either fail high or low. The processor whose window does enclose the position value is able

to find the correct move and score.

The maximum speedup of the method is believed to be only 5 or 6, regardless of the number processors available. This is because the cost of a partial search, i.e. a search over a restricted alpha-beta window, has a certain fixed overhead regardless of the width of the window (provided, of course, that alpha < beta) [Baud78].

An important result, however, is that when the degree of parallelism (i.e. the number of processors) is small (k=2 or 3), the speedup obtainable is greater than k. It is this fact, and the upper limit on the speedup, that leads Baudet to suggest combining his method with a tree decomposition scheme, particularly when a large number of processors are available.

4.3 Akl method

Akl et al. describe a strategy for tree decomposition which attempts to take advantage of certain regularities in the behaviour of the sequential alpha-beta algorithm [Akl80]. The essential idea is that, even in the best case, there are certain subtrees that alpha-beta must explore. Therefore these subtrees can be examined independently and concurrently as the first stage of a parallel algorithm. The subsequent stages of the algorithm can make use of the alpha-beta windows (set up by the first stage) to search the remaining subtrees independently, with minimal search overhead.

To describe the algorithm further, some additional terminology is useful [Akl80]. The first son of a node is called the left son, and is contained in a left subtree. All other sons of a node are right sons, and are contained in right subtrees.

The initial stage of the algorithm fully searches the left subtree of the root node, but only the left subtrees of the right sons of the root node. When this phase is complete, the left son will have a final value, while all the right sons will have temporary values (i.e. the values of their left sons). In Figure 2, those subtrees explored in the first phase are marked by the solid lines. If these temporary values are insufficient to cause a cutoff, as in Figure 2, then successive second sons of each remaining right node is searched, Figure 3, until either all right sons have been cutoff or fully explored[1].

The algorithm was implemented through simulation of physical parallelism. In the simulation, processes are generated to carry out subtree searches, and processors assigned to them from a pool according to a priority scheme. It is not clear whether this method is suitable for an actual realization on a multiprocessor. However, the importance of the point that certain subtrees must always be examined should be kept in mind. The algorithm's effectiveness was demonstrated on game-trees with randomly assigned terminal node values, and branching factors of five and less [Akl80].

[1] Actually the above description is an oversimplification, ignoring the fact that the algorithm is recursively applied at deeper levels in the tree. The ideas presented are essentially correct, however.

4.4 Tree-splitting algorithm

Fishburn and Finkel describe a parallel alpha-beta algorithm which they call the tree-splitting algorithm [Fish80]. It involves a relatively simple game-tree decomposition, and the use of a tree of processors. The algorithm has been implemented on the Arachne distributed operating system, and a theoretical model has been developed for certain simplified cases.

The tree of processors takes the form of an N-ary tree, in which nodes represent independent processors, and branches the communication lines between them. Parent nodes are called masters and children are called slaves. Thus the root of the tree is a master, all terminal nodes are slaves, and all intermediate nodes are either masters or slaves depending on the context. One important condition on the tree of processors is that it be shallower than the game-tree to be searched.

There are two separate algorithms run on the processors, dependent on whether they are a terminal node or not. Terminal nodes run the slave algorithm. These processors are passed a position and an alpha-beta window, and search the subtree (sequentially) down to the maximum depth. The calculated value is returned to the master.

All non-terminal nodes run the master algorithm. They are also given a position and a window from their master (or from the input, in case of the root processor). The successors of the position are generated and assigned to the slaves one by one. When the subtree search is done (i.e. all successors have

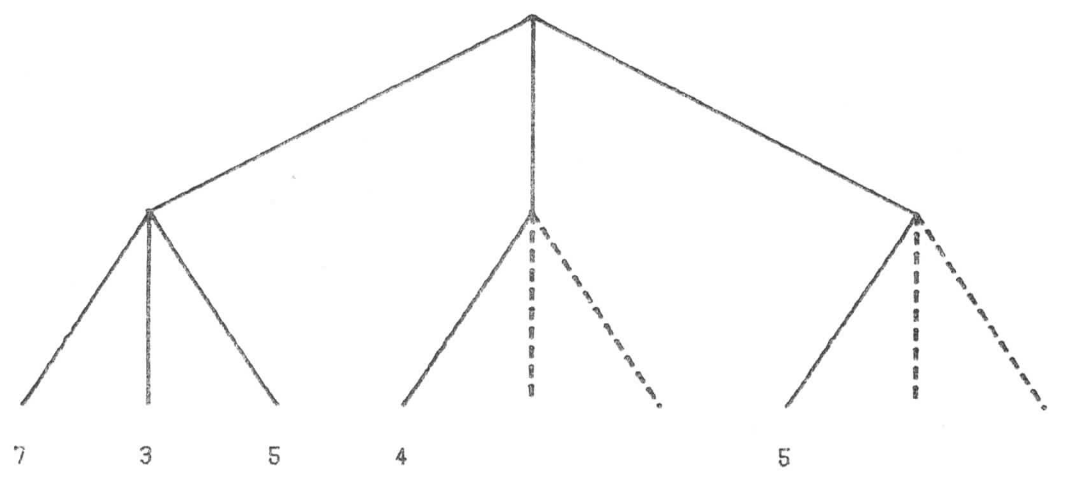


FIGURE 2: FIRST PHASE OF SEARCH

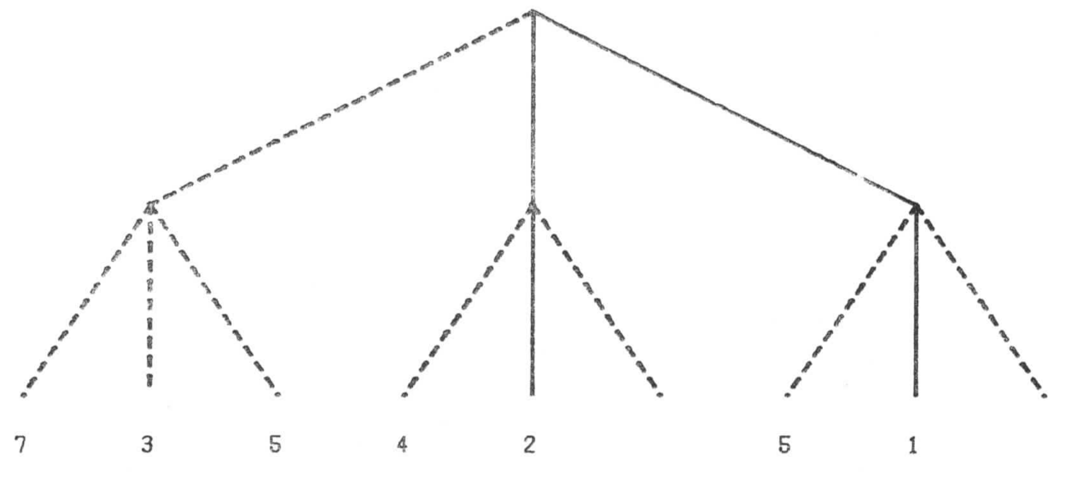


FIGURE 3: SECOND PHASE OF SEARCH

been scored by slaves, or the search has been cut off), the calculated value is returned to its master.

An important feature of this method is the provision for dynamic updating of alpha-beta windows. If a master gets a return value from a slave which narrows the window, a method has been described whereby the new window can be transmitted to slaves whose search is in progress. Our pseudo-code version, adapted from [Fish80], to carry out dynamic updates for the negamax variant of alpha-beta is given in Figure 4. It is assumed that the alpha and beta values are maintained in a global array, rather than on the stack, and that procedure update is called asynchronously by means of an interrupt.

```
int alpha[MAXDEPTH], beta[MAXDEPTH];

update(newalpha, newbeta)
int newalpha, newbeta;
{
      int k, tmp;
      for k = 0 until (MAXDEPTH - 1)
      {
            alpha[k] = max(alpha[k], newalpha);
            beta[k] = min(beta[k], newbeta);
            tmp = newalpha;
            newalpha = -newbeta;
            newbeta = -tmp;
      }
}
```

Figure 4: Interrupt invoked update of alpha-beta window

There are a number of optimizations possible for this searching scheme.

(1) Since most of a master's time is spent waiting for messages, the deepest masters can join in the search by running the slave algorithm, in addition to their master function.

(2) It is possible to group several higher level masters (the least active processors) onto a single processor.

(3) Masters can assign successor's successors to slaves. This is equivalent to increasing the common depth in the naive method. The results are that the processors have less idle time, and less search overhead, at the cost of increased communication.

A problem inherent in the tree-splitting algorithm arises as a result of the restriction that the processor tree be shallower than the game-tree. In chess, for example, adding 1 ply to the game-tree increases it's size by a factor of 6 (approximately). However, several layers of, say, a binary processor tree would be needed to allow a 6-fold speedup. In other words, the processor tree will quickly catch up in depth with the game-tree as more processors are available. One way around the problem is to increase the branching factor of the processor tree. This is not entirely satisfactory, as efficiency is reduced due to increased search overhead.

4.5 Enhancements and Hybrid Systems

Some unexplored possibilities of potential use in the parallel search of game trees will be considered now. These include variations on the already presented methods, a minimax algorithm which does not use alpha-beta, and other techniques not directly related to the particular searching strategy chosen.

4.5.1 Transposition Table : The use of a transposition table to aid tree search has been successfully demonstrated [Green67], and in improved forms is now used by all except the small-computer chess programs. A transposition table is basically a large hash table containing positions and their values. Whenever a position is generated during a search, a check is made to see if it appears in the table and has already been fully evaluated. If a useable value exists, it is taken and the search is cut off. Otherwise the position is scored in the normal way, and the value is entered into the table. Thus identical positions in different parts of the tree need not be searched twice.

The advantages of a transposition table in a multiprocessor chess-playing program are twofold: (1) it provides the search reduction available to a sequential program, and (2) it can be handled independently and concurrently. The potential problem involved with a transposition table is that it will become the object of contention between processors. There are various ways to reduce the contention. The table can be divided into N sections, reducing the table queue length by a factor of N. Also, processors can be assigned as table controllers. The processors carrying out a search can send their requests to the appropriate controller, and continue their search. The controller can interrupt the search if a useful value is found.

4.5.2 Iterative Deepening : Another technique employed with great success in current chess programs is iterative deepening, which uses a search to depth N-1 or N-2 to obtain improved move-ordering for a depth N search. Iterative deepening is particularly effective when used in conjunction with a transposition table, and can be expected to provide an equivalent speedup in either sequential or parallel programs.

4.5.3 Hybrid Systems : Since the parallel aspiration search provides a greater than k-fold speedup for k processors, where k is small (2 or 3) [Baud78], it seems logical to combine this technique with tree decomposition methods. The tree-splitting algorithm is particularly suited for the amalgamation, since it suffers from the restriction that the processor tree must be shallower than the game-tree. Therefore, Baudet's method provides a limited alternative to increasing the branching factor of the tree of processors when a large number of processors are available.

A parallel aspiration search is of somewhat less use than might be expected in the search of chess game-trees. This is because a fairly narrow window is usually available at the start of the search with a good probability of being correct. The method of choosing a window varies, but if it is correct sufficiently often, parallel aspiration searches can be somewhat wasteful. It is reasonable, however, to have a 'backup' search being conducted in parallel with the regular search, in case the estimated window was incorrect. The backup search would use a (-infinity,+infinity) window to ensure success.

4.5.4 SSS* : SSS* is a minimax algorithm which is more efficient than alpha-beta in terms of number of terminal nodes explored [Stock79]. The algorithm seems applicable to parallel search, since it maintains alternate search paths simultaneously throughout the tree. However to do this requires the maintenance of a large data structure (of order (b ** (d/2)), where d=depth, b=branching factor). In a parallel system, the synchronization overheads involved with such a shared data structure can eliminate the gains made due to concurrency. We are working on ways to combine SSS* with alpha-beta, in order to reduce this problem.

4.5.5 Parallel Evaluation : Since most current chess-playing programs spend roughly half their time evaluating terminal nodes, parallelism could be usefully introduced by employing a number of processors to evaluate concurrently different terms in the scoring function. This technique has numerous advantages:

(1) evaluation time would be reduced;

(2) small, cheap processors could be used to evaluate single positional features;

(3) there is no obvious limit to the concurrency possible [Scher80];

(4) the scoring function can become more sophisticated without slowing the program. It has been noted that more complex evaluation functions can simulate the effect of deeper searches [Slate77], and this is also the basic premise of the knowledge based, selective search, programs with their complex capture analyzers [Mars74].

5. HARDWARE CHOICES: ZILOG Z8000 AND THE MOTOROLA MC68000

In order to bring out the desirable properties of the hardware necessary for this application, the general architectural features of two 16-bit microprocessors will be summarized. Emphasis will be placed on features directly relevant to multiprocessing and chess game-playing.

The Z8000 and MC68000 are both high-throughput systems designed to solve a wide range of applications. Both have abundant resources which include numerous registers, support for many data element types, a large instruction set and a large address space. A common characteristic of the Z8000 and the MC68000 is regularity in the design of the instructions, addressing modes and data types. This greatly simplifies the work of the programmer and significantly reduces the program length.

The designers of both systems have tailored their machines to support compiler and operating system code efficiently by providing several mechanisms for handling interrupts and traps. Multi-processing is also supported by both designs. The architectural features of the two systems, to be presented, are summarized in Tables 1 and 2.

5.1 CPU Resources

The Z8000 has sixteen 16-bit general-purpose registers which can be used as index registers and as accumulators. These registers may also be reconfigured as 8-bit (byte) registers, as 32-bit registers and even as 64-bit registers. The Z8000 supports seven main data types: bits, BCD digits, bytes, words (16-bit), long words (32-bit), byte strings and word strings. Additionally, memory addresses, I/O addresses, segment table entries and program status words are also provided.

There are five main addressing modes: Register, Indirect Register, Direct Address, Indexed and Immediate. For certain instructions, there are other addressing modes: Base Address, Base Indexed, Relative Address, Autoincrement and Autodecrement. With very few exceptions, almost all of the 110 instructions can work on byte, word and long-word quantities.

In addition to the 32-bit program counter and 16-bit status registers, the MC68000 has seventeen 32-bit registers. Half of which (the first eight registers D0-D7) can be used as 8-bit, 16-bit and 32-bit data registers. The other half (registers A0-A6) and the dual system stack pointer (user and supervisor) can be used as software stack pointers and base address registers. All of the seventeen registers may be used as index registers. The MC68000 supports bits, BCD digits, bytes (8-bits), 16-bit words and 32-bit long words. In addition it can work with memory addresses and status word data.

There are six basic addressing types: Register, Indirect Register, Absolute, Immediate, Program Counter Relative and Implied. Together with the register indirect addressing modes is the capability to do postincrementing and predecrementing, offsetting and indexing. With very few exceptions, each of the fifty-six different instruction types can work with all six basic addressing types.

5.2 <u>Address Space and Memory Management</u>

Both the Z8000 and the MC68000 have large address spaces compared to 8-bit microprocessors and even to minicomputers currently in use. For example, the Z8000 has six separate data spaces: code, data and stack for both the system mode and normal (user) mode. Each of these can have up to 8 megabytes of storage. For small applications which do not require the full use of the address space, a short offset form is available wherein the high order 8 bits are set to zero, and so point to the first segment. With a separate memory management unit, relocation and segment protection can be provided. The segmented addressing feature alleviates the need for long instruction formats and the demand for register pairs. On the segmented version of the Z8000, the user can designate up to 128 segments that reference areas of memory varying in size from 256 bytes (minimum) to 64 kilobytes (maximum) in 256-byte increments. The Memory Management Unit of the Z8000 comes in a separate LSI package.

The MC68000 on the other hand, has a 24-bit address bus which provides an address range of more than sixteen megabytes. With the use of the numerous addressing modes and a flexible instruction set, the large address space is easily accessed and managed. Basic memory protection is provided by instructions which allow checking of address values against preset bounds. The memory management operations are transparent to the programmer when he is in the user mode, and can only be changed or updated when the processor is in supervisor mode.

5.3 Code Density and Speed

The speed of a processor is dependent upon the number of instruction words executed. The number of words needed to execute the most frequent instructions has been minimized in both the Z8000 and the MC68000. The short offset mechanism of the Z8000 allows the address to be reduced to a single word. The increased regularity and consistency of the two machines contribute greatly to this end.

Of greatest importance in this aspect are the instructions which are most frequently used in programs. For example, only one word is used on the Z8000 to implement the commands JUMP RELATIVE and CALL RELATIVE. Similarly the TRAP, move multiple registers (MOVEM), link stack (LINK), unlink stack (UNLK) and check limit instructions on the MC68000 significantly reduce the code requirements for subroutines, operating system calls and stack operations. In both machines arithmetic and data manipulation instructions have been implemented efficiently.

The MC68000 has the advantage of speed over the Z8000. The reasons stem from the use of a faster clock (8MHz versus 4MHz on the Z8000), the use of a more uniform instruction set, the separation of address and data lines, and larger internal registers (32-bit versus 16-bit for the Z8000).

5.4 Operating System and Multiprocessing Support

Numerous interrupts and task-switching features (traps) are present in both machines, including non-maskable and vectored interrupts. On the Z8000, vectored interrupts cause a 16-bit value to be read from the data bus. This 'vector' value is used to select a particular interrupt procedure located in the Program Status Area in a reserved area of memory. Similarly there are 255 vector locations available on the MC68000 for interrupts, hardware traps and software traps. On both machines, instructions are available to load groups of registers and to examine and set Status Registers on both machines. Also TEST-and-SET instructions provide a mechanism for the synchronization of cooperating processes.

In addition to the large address space, these two machines provide unique built-in features which simplify their use in multi-processing applications. The Z8000 CPU can exclude all other asynchronous CPU's from any critical shared resource by using Micro-In input and Micro-Out output, in conjunction with the REQUEST, RELEASE, TEST uI, SET uO, and RESET uO instructions. The MC68000, on the other hand, contains bus arbitration logic for a shared bus and shared memory environment (shared with other MC68000 CPU's, DMA devices, etc.), which is the counterpart of the special multiprocessor instructions found on the Z8000.

Table 1. General Features

	Z8000	MC68000
Manufacturer	Zilog	Motorola
Clock Frequency	4 MHz	8 MHz
Bus Cycle Time	750 nsec	500 nsec (read) 750 nsec (write)
Vectored Priority Interrupt	vectored interrupt- no on-chip priority arbitration	yes - 7 levels
Virtual Memory and Memory Protection	yes - with separate Z8010 memory management unit(s)	yes - with separate memory management unit
Maximum Address Space per Process	with memory management 128*64 kilobyte code, 128*64 kilobyte data, and 128*64 kilobyte stack without memory management 8 megabyte code, 8 megabyte data, 8 megabyte stack	with memory management unknown without memory management 16 megabyte code, 16 megabyte data
Maximum Amount of Physical Memory	with memory management 16 megabyte per Z8010	with memory management unit unknown
Interlock Instructions for Multiprocessor Operations	yes	yes

Table 2. High-level language support

	Z8000	MC68000
Data Types	bits 2-digit BCD numbers 8- and 16-bit logicals 8-, 16-, and 32-bit signed integers 32-bit addresses (23 bits used) byte strings word strings	bits 2-digit BCD numbers 8-, 16- and 32-bit logicals 8-, 16- and 32-bit signed/unsigned integers 2-bit addresses (24 bits used)
Registers	15 general-purpose 1 procedure-call stack pointer 1 status	data (32-bits) address (32-bits) procedure-call stack status
Address Modes	immediate reg direct reg indirect reg indirect indexed reg indirect with offset reg indirect with post- in/decrement absolute absolute indexed relative	immediate reg direct reg indirect reg indirect indexed reg indirect with offset reg indirect indexed with offset reg indirect with pre-decrement reg indirect with post-increment absolute
Procedure Call Support	procedure call stack; save/restore registers, call, return, and load effective address operations push and pop operations	procedure call stack; save/restore registers link/unlink stack call, return, load effective address and push effective address operations; move instructions in combination with auto-increment/decrement instructions
Other Stack (besides procedure call stack)	yes - 7 register pairs can be used as stack pointers	yes - any of the 7 address registers can be used as stack pointers

6. ARCHITECTURAL CHOICES FOR MULTIPROCESSING

Architectures which can be used in a multiprocessing environment have their advantages and disadvantages depending upon the application, the characteristics of the processor used, the peripheral devices attached to the system and the software installed on the system (the operating system and the 'user' programs).

Since the emphasis of the project is to use 16-bit microprocessors to explore the computer-chess problem, only implementations which can be applied to microprocessors and existing memory capabilities will be discussed in detail. Our presentation in this section owes much to the recent book by Weitzman [Weitz80], which is also the principal source of our diagrams. Only passing mention will be made of architectures designed for large general-purpose computers. The basis for comparison will be the speed with which small amounts of information, such as alpha-beta windows and transposition table values, can be shared between the processors.

6.1 Shared Memory Schemes

One primary means of interaction between processors can be provided by common memory. In this case the manner in which memory is shared must be clear. In the past multiprocessor systems, like the Univac 1110 and the IBM 370/168 MP, had tight coupling, i.e. memory and I/O peripherals were shared through a common operating system. In more recent systems, processors have their own private memory and executive software. Shared memory is used mainly for passing data to be processed and to

inform other processes of the status of program execution (not for the program to be executed itself). This type of configuration would be most suitable for dedicated applications. Some ways in which memory can be accessed easily by all processors in the system include: shared bus, matrix switch, virtual memory and mailbox schemes.

Common memory is a simple, inexpensive and flexible means of interconnecting several processors. Serious problems with this scheme include contention of the different processors for the main bus and the inherent limitation imposed by the bandwidth of the bus on the overall speed of the system. Bus arbitration logic is a way of streamlining the data traffic. The amount of data being transferred can also be reduced by providing each process with a small private memory for local intermediate results. Figure 5 illustrates a basic shared memory configuration.

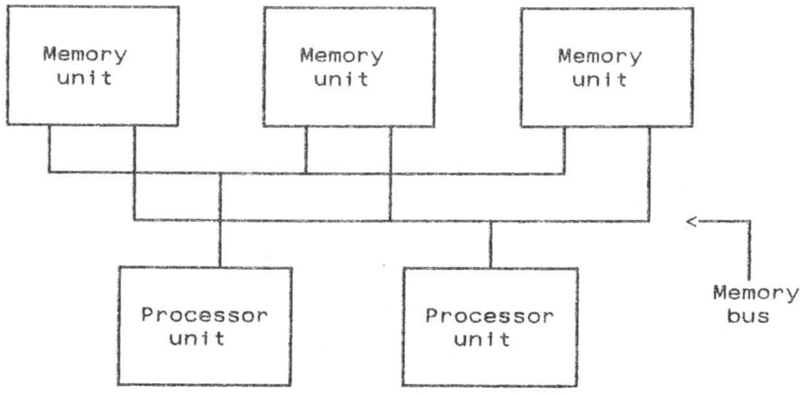
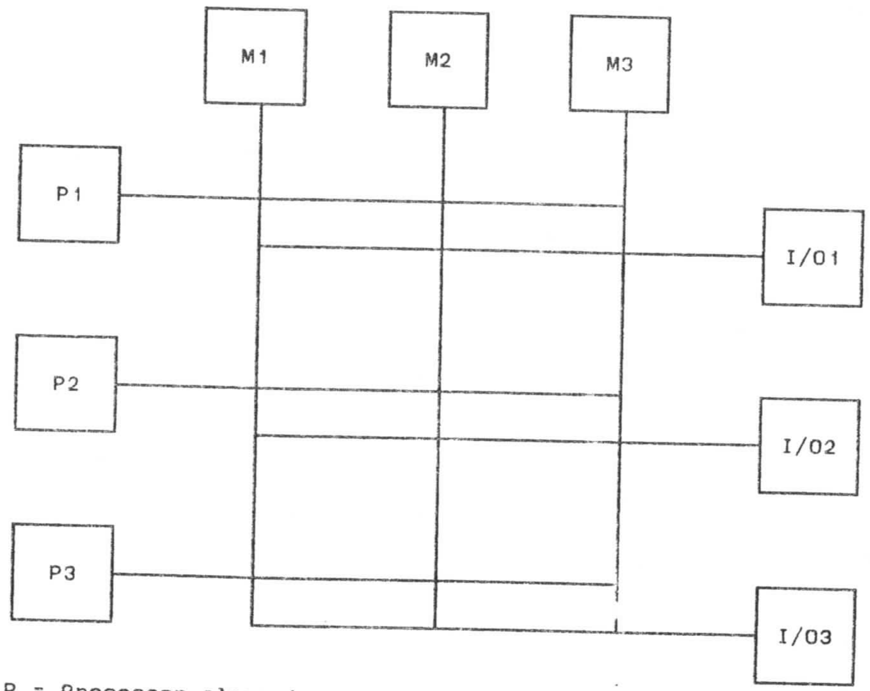


Figure 5: Basic shared memory configuration

In cases where processors need to share not only each other's memory but also all available I/O units, provision of separate buses and control logic to interconnect the devices constitutes a matrix switch configuration, Figure 6. While this scheme can support a large volume of simultaneous data transfers, it is quite costly to implement and the number of buses grows exponentially with the number of devices connected. This might be suitable for a system with ten devices (processors and peripherals) or less, if the volume of shared data warrants its use. An example of this would be C.mmp [Weitz80].



P = Processor element
M = Memory element
I/O = Input/output controller

Figure 6: Matrix switch configuration

Similar to the matrix switch, the multiport memory arrangement places arbitration logic in the individual devices themselves. This configuration shares the high volume (of data transfer) capacity with the matrix switch, but suffers from inflexibility. Additional processors would entail more ports in the memory.

Processors in a system can share memory by allowing their virtual addresses to map into the same location in physical memory. This gives the advantage of protection provided by the address translation mechanism. As typified by Cm*, a special (Kmap) microprogrammed processor performs the memory allocation and handles all communication between different modules [Swan77].

In order to reduce contention problems, use of common memory may be restricted to the exchange of messages. Virtual address translation is no longer needed because each processor would be assigned a segment in memory to be used as its mailbox. Control of shared memory and the frequency with which it can be shared are paramount considerations in this scheme. Controlled access can be accomplished through the use of locks, priority levels and interrupts. Synchronization software can be implemented with the use of TEST-and-SET instructions, and interlocks can be ensured if the memory is capable of read-modify-write cycles before permitting further accesses.

The most promising choices from this group seem to be the common memory, the virtual memory and the mailbox memory schemes. Both microprocessors can accommodate any of them, since the instruction sets and the peripheral devices can handle the requirements of these configurations.

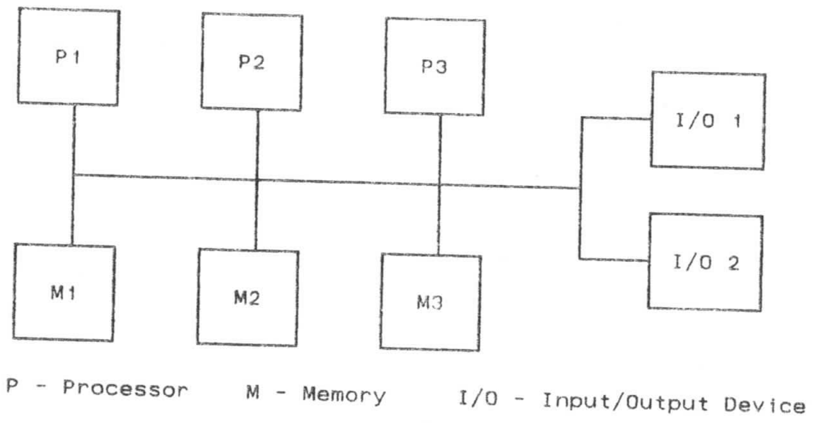


Figure 7: Time-shared common bus system

6.2 Shared Bus

This technique employs a common I/O channel to connect each processor. Control of the bus may be centralized into one global switch which can poll the requests and allocate the use of the bus in a predetermined allocation strategy. Already there are some logic devices available which can perform bus arbitration for a multiprocessing environment using 16-bit microprocessors. Control can be decentralized by assigning finite time periods or 'slots', during which different processors can control the access of the bus, Figure 7. In other cases, a multiple access bus may be shared using collision control software algorithms. The bus can be of the

polled, interrupt-driven, time-multiplexed or multiple-access type.

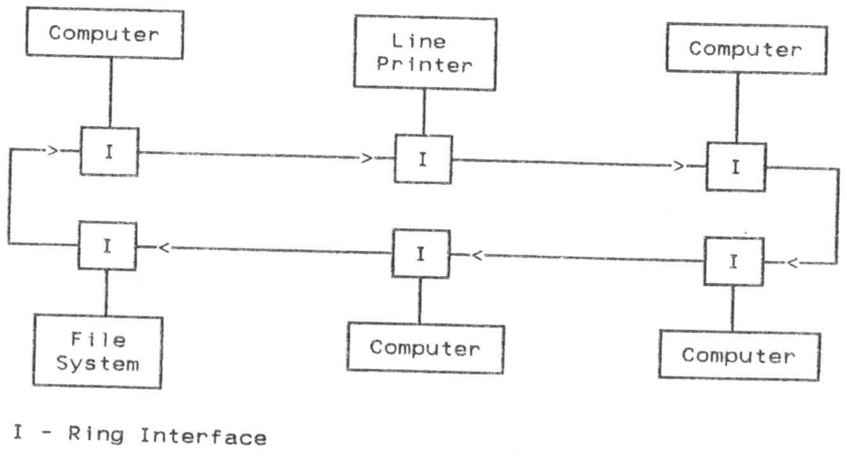


Figure 8: Loop-organized system

6.3 Broadcast Architecture

Although conceptually very similar to the shared bus scheme, a loop multicomputer system, Figure 8, typically consists of a high-speed uni-directional digital communication channel which is arranged as a closed loop. Nodes such as micro- or mini-computers can be attached through the use of a loop interface. A message between nodes is placed on the loop and is transferred from node to node until it reaches its destination. Either the sender or the receiver may be the one responsible for removing the message from the loop. The main advantages of this technique are: the elimination of message routing problems; the simplification of the interconnect structure; and the use of a passive broadcast medium for reliability. However, the cost of creating a loop interface for each processor may be quite high. Additional software is needed to drive these interfaces. Furthermore, this organization is characterized by long access times, making it more sutable for long message transfers.

6.4 Star Architecture

In this type of interconnection, a central processor acts as a communication device between all the other processors in the system. These processors either ask to send messages or acknowledge their receipt. The centralized switch, with queuing of communication requests, gives this system better response-time characteristics than the more general loop systems. However, no partial operation is possible since 100% reliability of the central switch is needed.

6.5 Tree Architecture

Similar to other hierarchical structures, well-defined specialized tasks are performed at the base, whereas the top of the organization has a more general-purpose capability. The processors at the base have few peripherals and may perform repetitive and tedious functions of the algorithm. I/O processing, program execution control and report generation are performed at the top. Typically, shared data bases are also stored at the top, rather than being distributed throughout the

system. The main advantage of this scheme lies in its applicability to problems using trees as data structures and the orderly control within the system.

For example, two-man game-playing applications employ trees to represent all the possible moves that each side can make. These trees can be searched systematically in parallel by assigning the search of different branches to distinct processors and then having the results of all the searches collated by a single 'master' processor which occupies the top of the hierarchical structure. See for example the three-level structure shown in Figure 9.

This section has dealt with interconnect structures suitable for multi-microprocessor systems. The final system configuration for this project may be a combination of some of the schemes which were discussed, rather than any specific organization.

7. MULTIPROCESSOR COMMUNICATION SOFTWARE

A multiprocessor system must provide not only all the services needed by a uniprocessor, but also a level of control beyond the management of local resources. The extent of this control depends upon the system architecture. In a centrally-controlled system, using bus, star or hierarchical structures, the processors are usually organized in a master-slave configuration. Control flows from upper levels of the structure down to the subordinate levels. On the other hand, broadcast and point-to-point configurations create a master-to-master

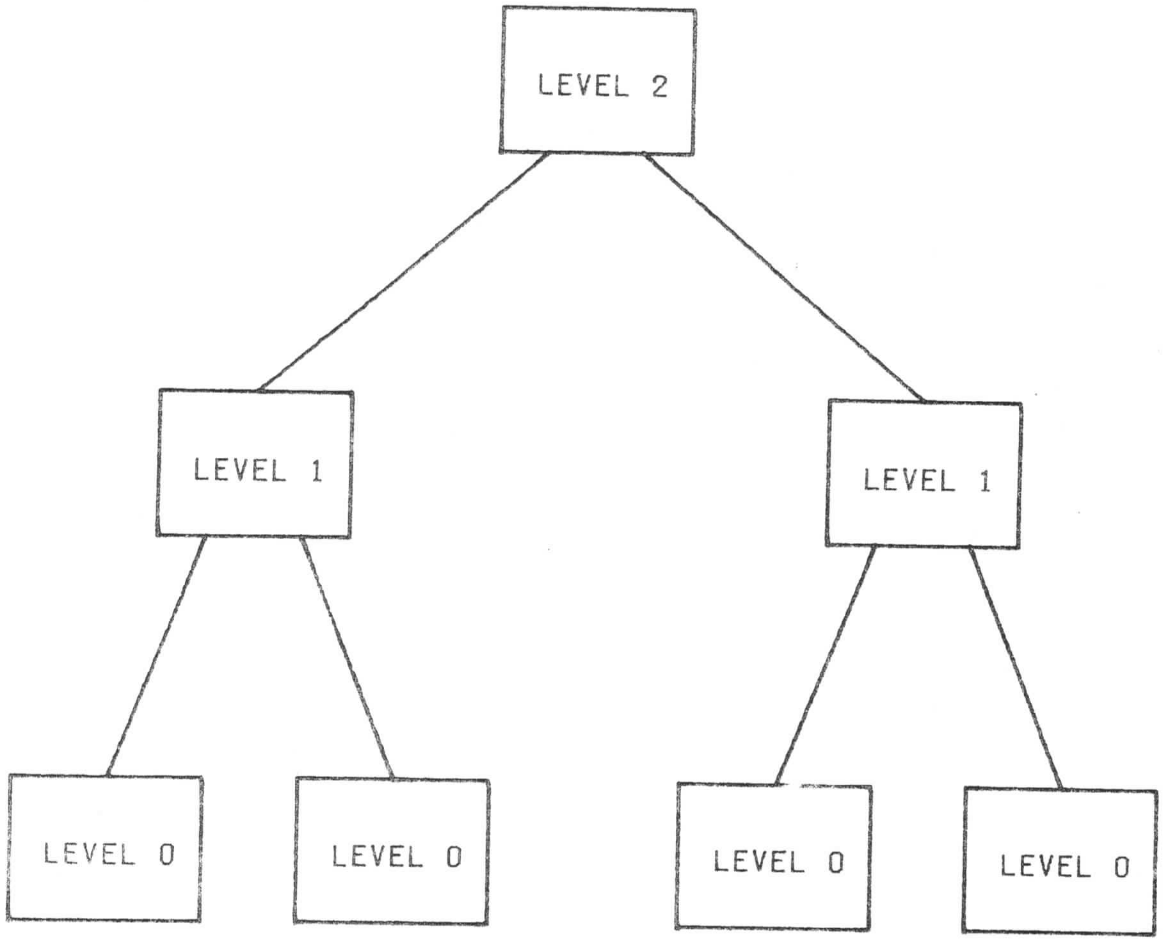


FIGURE 9: THREE-LEVEL HIERARCHICAL SYSTEM

relationship among the processors within the system.

In any of the above cases, the control of processors and resources must always ensure that there will be an efficient and consistent allocation method, and that race and deadlock conditions are eliminated. To be more specific, the control function must answer the questions of task handling and partitioning, diagnostics and failure recovery procedures, and provision of coherent communication between system components. On top of these would be a capability for down-line loading of programs, program development support, and overall system security.

However, as with the case of microcomputer applications used for dedicated tasks, one need not bother to have all the features necessary for robust operation. Game-playing applications perform a limited variety of tasks and so the support software can be simpler. In any case, with the use of numerous processors and specialized logic modules, the implementation of a general-purpose operating system may not be practical at all.

Fundamental to multi-microcomputer applications is the need for interprocessor communication. Two basic schemes are possible: either use shared memory or employ a message-passing protocol.

7.1 Shared Memory

This scheme makes controlled use of a common memory. All the processors which require sharing of data, communicate by accessing the same memory region. Usually no acknowledgement is necessary, since techniques such as semaphores and monitors (with and without mutual exclusion) may be used to ensure the integrity of the data in memory. No acknowledgement of the receipt of a message is necessary and since only write accesses need be monitored, this method may be quite practical for a system with modest message-passing requirements. To achieve speed and accuracy, the processors should be within close proximity of one another, as would be the case in the computer chess application.

7.2 Message passing protocols

The use of message-passing protocols is more common in distributed systems. This scheme will work well for systems ranging in size from large network-type architectures to small twin-processor micros communicating through a high-bandwidth bus link. Normally such protocols are arranged hierarchically. Communications are normally carried out between adjacent layers only. Above the physical layer, there are still three possible protocol levels: the communications link (logical connection), the network control (interprocess communication), and the user-layer [Zimm80].

The communications link protocol layer contains rules by which data can be reliably transferred from one system component to another. Its main functions include: data transfer control; error checking and recovery; information coding and transparency; optimization of line utilization; maintenance of line synchronization; communications facility transparency; and bootstrapping. Information transparency in this case means being able to correctly interpret incoming signals as data or as code for the protocol program. Communications facility transparency means being able to make use of different types of communication paths without having to change the protocol format or procedure.

Interprocess-communication layer protocols are concerned with the bookkeeping required to allow several processors, each with a valid logical end-to-end conncection, to share a single physical link. This may very well be called the <u>logical</u> communications link control layer. A major component of this layer would include message-routing procedures, and flow control.

User-layer protocols are actually high-level functions which are packaged together to perform tasks specifically requested by the user. A particular example would be protocols which allow movement and alteration of files residing within the different parts of the system. This involves exchange of information: particularly commands, data and even state variables needed for synchronizing the system. User-layer protocols are usually built on top of interprocess communication protocols.

For a specific application, all of these layers must be provided by the user. The hardware linking all the processors and their associated memory must be provided by the designer.

In addition to the hardware, the software must also be present to recognize and interpret the signals that the hardware has been designed to pass around. The software must also be able to recognize the commands that a programmer running code in the system issues.

A system with several independent processors will require the physical interconnection between all of them. Software in this system must handle 'packaging', routing and recognition of messages within the system. Lastly, these functions must be performed efficiently and without the risk of deadlock. For our application the packaging and routing problems are minimal, and each processor is responsible for accepting and updating information provided by the others.

8. DESCRIPTION OF POSSIBLE SYSTEM

In order to implement a parallel alpha-beta search for a game-playing program, certain goals must be kept in mind in the system design.

(1) The system should be small enough to build easily, yet provide a reasonable improvement in performance over a uniprocessor.

(2) The system should be extendable, in that additional processors can be added to it in a regular manner without the introduction of excessive search overhead.

(3) The system should employ a transposition table because of its known benefits and to obtain empirical data on the effectiveness of this technique in a chess/multiprocessor

environment.

(4) For the sake of both cost and convenience, the system should employ 16-bit microprocessors for tree searching, move generation and transposition table operations, but could use groups of very cheap processors to compute the evaluation function in parallel.

8.1 Choice of Search Algorithm

Of the different parallel search algorithms discussed, it appears that the tree-splitting algorithm of Fishburn and Finkel is most useful for our application. Baudet's method is not particularly effective for chess programs, because a small window can be estimated rather easily before the search takes place. The tree-splitting algorithm searches in a very regular and consistent manner, making implementation easier than, say, Akl's method. The problems of the naive method are also circumvented to a large extent; search overhead is small when the processor tree has fanout of 2 and uses dynamic updating of alpha-beta windows.

The searching algorithm will be augmented by a transposition table, known to be useful in sequential chess programs. The effectiveness of this enhancement in a parallel system will have to be studied, but should be efficient since it is easily partitioned into an autonomous parallel function.

8.2 System Organization

The system described here will be a simple one. A functional description will be given as well as the proposed hardware implementation. A schematic of a triple processor prototype is given in Figure 10. One processor is designated the master, and the others are its slaves. The master provides the user interface, assigns subtree searches to it's slaves, and controls access to the transposition table. The slaves search their assigned subtrees, accessing the transposition table indirectly through the master.

Certain information must be passed between the master and it's slaves. Initially the master sends a position and an associated alpha-beta window to the slave. Subsequently updates of the alpha-beta window, and data obtained from the transposition table are sent. The slaves must return the score of a position that has been searched. The slaves must also send the master their requests for access to the transposition table, either for score information or to update the table when a position is scored.

Basically the master assigns subtree searches to the slaves. The slaves, while carrying out their search, send requests for transposition table lookups, but proceed concurrently with their search. Only if the master finds something useful in the table will it interrupt the slave with the new information to terminate the search. The dynamic update of the alpha-beta window is also handled via interrupts. Updates of the transposition table are simpler, requiring no communication back to the slave.

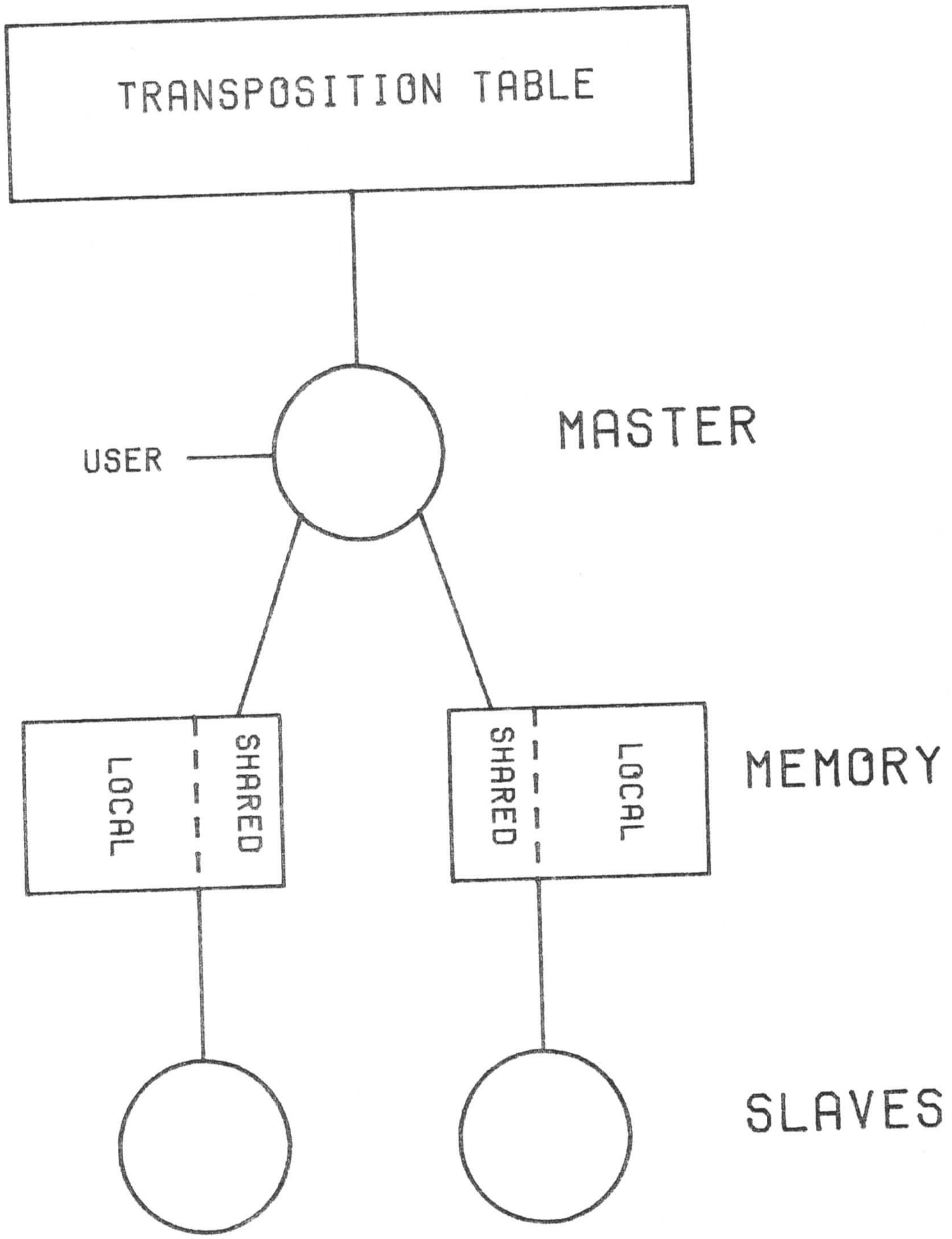


FIGURE 10: SYSTEM ORGANIZATION

8.3 Hardware Package

Inasmuch as the transposition table needed by the chess program can be accommodated by any of the two machines under consideration, the hardware package design depends mainly on the availability of appropriate instructions suited to the chess problem, and the speed at which the most common instructions are executed.

As noted in Section 5, the MC68000 has a distinct advantage in terms of speed of execution of instructions needed to implement subroutine calls. The MC68000 also has the advantage when it comes to logical operations and manipulation of data in registers and in memory. Hence the MC68000 seems to be the logical choice at the moment to perform the function of the master and the slave. It can handle the communication problem through the use of shared memory.

In future expansions, if the system demand exceeds the 16M address space of the MC68000, the Z8000 can be used. This may be in the form of a (transposition) table-master. The Z8000 along with some memory management chips can handle the address mapping and allocation of a larger memory.

It should be noted that the software for the system is being written in the C-language [Kern78] and is being compiled with the use of software support we have developed for a UNIX system [Bell78]. At the moment, the C-compiler for the MC68000 produces more 'efficient' code than that for the Z8000. This may be attributed to the fact that a code optimizer has not been installed in the Z8000 compiler, and that the architecture

of the MC68000 has a greater similarity to the PDP 11's (for which the portable C-compiler was designed), than does the Z8000.

8.4 Communication Scheme

The communication requirements of the proposed system can be conveniently handled through the use of common memory. Slave processors will share at least 4 Kilobytes of memory, with the balance available for storing intermediate calculations. The master will allocate its memory to system code (command handlers, interrupt handlers, slave-schedulers, memory management) and for the transposition table itself.

This scheme can be implemented by reserving areas of slave-processor memory for communication. The master processor can send interrupts to the slaves and must be able to lock the memory for writing. The same locking capability is available to the slave processors. All I/O interfacing will be done with the master processor. Slave processors will not be accessible from the user end.

8.5 System Extensibility

The use of the tree-splitting algorithm [Fish80] allows the tree of processors to be extended in depth arbitrarily, as long as it remains shallower than the game tree to be searched. Also there is the possibility of increasing the branching factor of the processor tree, although this technique has limited effectiveness. In both cases however, a problem arises in the use of the transposition table. With increasing numbers of processors searching concurrently, the contention for this table becomes very high. Even with a processor controlling the table, and queueing requests and updates, the system will reach a point where information cannot be accessed fast enough to be useful.

There are various solutions to this problem. Table hash indices can be calculated by the slave processors. This reduces the amount of information to be transferred, and lessens the work necessary for the table controller, thereby shortening the queue. Furthermore, it is quite likely that the slave will be able to use the calculated hash index in a variety of ways. Also, the table can be split into sections with a controller for each section. Unfortunately this implies that all slaves must be able to communicate with all controllers. It is possible to have, say, a master controller through which requests are routed. This reduces the processor interconnection requirements considerably. If the master controller's task is made simple enough, it need not become a bottleneck.

9. SUMMARY

In this paper we have presented various alternatives for the design of a multi-microprocessor chess playing system. The choices made for each system component can be summarized as follows:

Processor: Given constraints previously discussed, the Z8000 and M68000 were the alternative processors possible. The M68000 was chosen on the basis of its greater speed and more

flexible instruction set, with its somewhat smaller address space not a serious factor.

Communications: Though not entirely independent of the searching algorithm (different search methods have different interprocessor communication requirements), our system is based on the use of shared memory. It's advantages include its speed, and the suitability of the M68000 processor for such a scheme.

Searching Algorithm: From the four basic searching methods previously presented, the two most useful for our purposes were [Akl80] and [Fish80]. The naive method's disadvantages have already been discussed, and a parallel aspiration search [Baud78] is not particularly advantageous in a chess environment, nor extensible enough when more processors are added to the system. The tree-splitting algorithm of Fishburn and Finkel was the choice finally made, based on a simpler and more uniform control structure than that of Akl's. This desire for simplicity overshadowed the greater extensibility, and possibly greater searching efficiency of the latter method.

The reasons for the addition of a transposition table to the system include the excellent results in sequential chess programs, a desire to increase the amount of interprocessor communication to reduce search overhead, and the fact that it can be naturally partitioned into an asynchronous function.

REFERENCES

[Akl80]
S. Akl, D. Barnard, and R. Doran, "Design, Analysis, and Implementation of a Parallel Alpha-Beta Algorithm", Technical Report No. 80-98, Department of Computing and Information Science, Queen's University, Canada (1980)

[Baud78]
G. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University (1978)

[Bell78]
The Bell System Technical Journal, July-August 1978, Vol. 57, No. 6, PART 2

[Fish80]
J. Fishburn, and R. Finkel, "Parallel Alpha-Beta Search on Arachne", Technical Report No. 394, Computer Sciences Department, University of Wisconsin-Madison (1980)

[Gill72]
J. Gillogly, "The Technology Chess Program", **Artificial Intelligence 3**, pp. 145-163 (1972)

[Green67]
R.D. Greenblatt, D.E. Eastlake and S.D. Crocker, "The Greenblatt Chess Program", Fall Joint C. C. 1967, Vol 31, pp. 801-810

[Heer80]
J. Heering, "The Intel 8086, the Zilog Z8000, and the Motorola MC68000 Microprocessors", **Euromicro Journal**, No. 6, pp. 135-143 (1980)

[Hoj79]
K. Soe Hojberg, "Queue Handling Arbiter Solves Shared Resource Conflicts", **Computer Design**, pp. 129-131 (November 1979)

[Jones78]
M. Jones, "Parallel Search of Chess Game Trees", M.Sc. thesis, School of Computer Science, McGill University (1978)

[Kern78]
B. Kernighan and D. Ritchie, "The C Programming Language", Prentice-Hall (1978)

[Knuth75]
D. Knuth and R. Moore, "An Analysis of Alpha-Beta Pruning", **Artificial Intelligence 6**, pp. 293-326 (1975)

[Mars74]
T.A. Marsland, "A users guide to WITA, a chess program", TR74-2, Computing Science Dept., Univ. of Alberta, Edmonton (1974)

[Motor80]
MC68000 16-bit Microprocessor User's Manual - Second Edition, Motorola Semiconductor Products (1980)

[Nadir80]
J. Nadir, and B. McCormick, "Bus Arbiter Streamlines Multiprocessor Design", **Computer Design**, pp. 103-109 (June 1980)

[Scher80]
T. Scherzer, Private conversations about the BEBE chess program, (Sept/Oct 1980)

[Slate77]
D. Slate, and L. Atkin, "Chess 4.5 - The Northwestern University chess program", **Chess Skill in Man and Machine**, pp. 82-118, Springer-Verlag (1977)

[Stev79]
D. Stevenson, "An Introduction to the Z8010 MMU Memory Management Unit:Tutorial Information", Zilog Corporation, (August 1979)

[Stock75]
G. Stockman, "A Minimax Algorithm Better than Alpha-Beta?", **Artificial Intelligence 12**, pp. 179-196 (1979)

[Swan77]
R. Swan, "Cm* - A modular, multi-microprocessor", **Proc. AFIPS**, pp. 637-644 (1977)

[Weitz80]
C. Weitzman, "Distributed Micro-Minicomputer Systems", Prentice-Hall (1980)

[Zilog78]
AMZ8000 Advanced Specifications Manual, Zilog Corporation (1978)

[Zimm80]
H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for open systems intercommunication", **IEEE Transactions on Communications**, Vol. COM-28, #4, pp. 425-432 (1980)