Chess and supercomputers: details about optimizing Cray Blitz

Robert M. Hyatt
University of Alabama at Birmingham
Birmingham, AL 35294

Harry L. Nelson
Lawrence Livermore Laboratory
Livermore, CA 94550

Abstract

The Cray YMP (or XMP) computer system offers particular advantages for computer chess programs. This paper describes the architectural features that Cray Blitz [5,6] uses to search approximately 200,000 chess positions per second. It also describes the programming and algorithmic changes required to take advantage of each of several architectural features of the Cray family of computer systems.

1. Introduction

The architecture of the Cray YMP (or XMP) computer system offers particular advantages for computer chess programs. 64-bit registers match perfectly with the number of squares on the chessboard. Advanced architectural features, such as pipeline instruction execution with many functional units, offer very high performance with proper program coding strategies. Large memory sizes (up to 128 million 64-bit words) support tremendous hashing/transposition tables that reduce the computational requirements of various parts of a chess-playing program. Finally, with as many as eight processors (Cray YMP), parallel algorithms produce significant performance increases that further improve the level of play exhibited by chess-playing programs.

The following sections discuss particular architectural features of the XMP/YMP machines and describe how Cray Blitz exploits the machine to search trees at approximately 200,000 positions per second normally, and sometimes beyond 500,000 positions per second in favorable cases.

2. General Hardware Overview

The XMP/YMP machines differ significantly from typical general-purpose machines such as Suns and Vaxes in several important ways. The simplicity of the instruction set is a major departure from most architectures (although new RISC architectures such as the Sun-4 machines now have this characteristic also), and it affects the design and development of chess programs. All instructions other than those that load from and store to memory operate on register operands. This effectively provides a small but fast cache for data. In many cases operands can be block loaded to registers at a rate of one per clock period (6.0 nanoseconds), but the primary storage for often-used variables must be in registers for optimum performance.

Vector operations included in the XMP/YMP architecture offer excellent performance when it is possible to use them. Though less than 5% of all instructions issued by Cray Blitz are vector instructions, and most of these are shorter than the maximum length (64), twice the speed is attained as compared with the non-vector version.

3. Vector Operations

The hash table lookup/store routines make frequent use of vector operations. Since the table is not large enough to store every position encountered during the search, and since any hashing algorithm produces identical hash values for different hash keys (board positions), some method of resolving these collisions must be used.

The method used in Cray Blitz is frequently referred to as "double hashing." Using this method, the program reduces the chess board to a 64-bit hash value. The program uses the low-order N bits (N depends on the size of the hash table chosen) to directly address the initial hash table location. The lookup/store routines append a 1-bit to the next nine bits (adjacent to the N bits mentioned above) to form a secondary hash increment, P, (where 0<P<1024 and P odd), to describe a linear array of alternative table locations. This constant increment vector (of

CH2916-5/90/0000/0354/$01.00 © IEEE

length eight for the current hashing algorithm) of table entries has addresses N, N+P, ..., N+7P.

Using a vector load operation to fetch these table entries, reduces the delay for the entire set to 18+7 clock periods where 7 is the number of extra entries to be fetched. Using a scalar type of operation, most chess programs probe exactly one entry in the hash table, Cray Blitz uses vector processing to examine several entries, allowing more flexibility in choosing which entry it overwrites (if necessary). Notice that the overhead of fetching seven additional entries is less than one-half of the time required to fetch the first entry. Several heuristics control which positions are kept (since one table entry contains information about only one chess position).

The only unusual requirement for using this mechanism appears when the initial hash value points to a position near the end of the table. Vector operations do not "wrap around" back to the start of the table (when the original position is near the end of the table). A solution is found by making the program append extra entries to the end of the table as shown in Figure 1. Since the rehash increment is ten bits long, this solution requires 1024*7 extra entries. Understanding this is made easier by visualizing a single table entry as containing 1+7 entries that are spread out evenly with a constant distance between them. Since one table entry can have many different hash values that point to it (by taking the low order N bits of many unique values), a single hash table entry can be a member of many different "sets" since there are 512 different rehash increments.

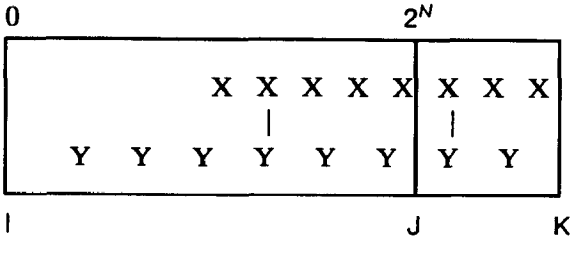

Figure 1: Hash table + vector overflow area

In Figure 1, the primary hash table contains $J-I+1=2^N$ words located between I and J. The vector overflow area is between words J and K. Notice that for the set of positions labelled X , three are in the overflow area. Notice also that for the set of positions labelled Y the spacing is different and that two entries in Y have the same addresses as entries in X (these X and Y entries are connected by a | in the figure). For the current hashing algorithm in Cray Blitz, a total of eight positions make up these sets. This requires an additional 7*1024 entries, a minor cost considering the performance gained by using vector operations.

This hashing algorithm is also used by the scoring functions to reduce their computational requirements. As an example, the pawn scoring routines do not consider anything but the placement of pawns. Since relatively few different pawn positions are encountered during a single tree search, saving the score for each different position saves a significant amount of time. This becomes even more important because it effectively reduces the time taken to score pawn positions to almost zero. Because of this, any new scoring code has almost no effect on the execution speed of the program. As a result, important bits of chess knowledge do not get "culled" to keep the search speed above a given level. This is also used for king safety computations for the same reasons.

4.0 Functional Unit Parallelism

Even on a single-processor XMP/YMP computer system, the architecture of the machine still offers an impressive level of parallelism if a programmer is willing to spend the effort to fully utilize the available hardware.

There are two distinct levels of parallelism (again, ignoring the multi-processing aspect of more than one cpu) that offer significant performance capabilities. The first type of parallelism results from the independent functional units within a single CPU. The second results from the internal pipelining of each of these units.

Assuming that the instruction stream avoids all conflicts, the instruction decoding hardware makes it possible to begin one new operation every clock cycle. After the instruction issues, an independent functional unit takes over and completes the operation regardless of the number of clock cycles

it actually takes. Moreover, using "chained vectors" it is possible to utilize multiple functional units simultaneously to further improve the execution speed of the program.

Although the Cray Blitz code is written in CFT77 FORTRAN, in order to effectively take advantage of the various hardware features, it is often necessary to code a replacement module in CAL (Cray Assembly Language). These modules are exact duplicates of the FORTRAN code in terms of results, but execute three to five times faster.

4.1 Instruction Shuffling

The XMP/YMP machines include the concept of "free cycles" which occur frequently in sequential programming examples. These free cycles are the cycles between the issue of two instructions that somehow conflict. Figure 2 illustrates a simple operand conflict.

target register	operand register	timing start	end
S1	S2+S5	0	3
S3	S1-S4	3	6
S5	S2+S7	4	7
S6	S5-S6	7	10
S7	S2-S4	8	11

Figure 2

In this example, the second instruction must wait for the scalar add functional unit to compute S2+S5 and place the result in S1. This takes three clock periods, therefore the second instruction will have to wait for that length of time before it can begin. The same sequence of events happens for the third and fourth instructions. After these two instructions issue, the fifth instruction executes with no delays. The entire sequence can be reordered as in Figure 3 to eliminate these delays.

target register	operand register	timing start	end
S1	S2+S5	0	3
S5	S2+S7	1	4
S7	S2-S4	2	5
S3	S1-S4	3	6
S6	S5-S6	4	7

Figure 3

In the preceding example, instructions are shuffled to eliminate the operand conflict waits completely. After this shuffling, each instruction issues in succeeding clock periods so that the next instruction is reached after 5 clock periods, rather than the original 9.

This might seem rather tedious, particularly analyzing the operand conflicts to properly shuffle the instructions. However, there is a program developed at Lawrence Livermore National Laboratory by Harry Nelson and Rollin Harding that automates this analysis task. The program, CYCLES, prints a timing analysis for a code pointing out where there are operand conflicts, memory wait delays, and functional unit delays. The programmer then uses this information to shuffle the instructions and move them around to eliminate as many delays as possible.

Using CYCLES, together with the programming tricks discussed below, it is possible to reduce the running time of a code substantially by reducing the number of cycles where the hardware is doing nothing but waiting.

4.2 Instruction Lifting

A simple example concerns the typical memory_load, add, and memory_store sequence of instructions that work efficiently on most computer architectures. On a YMP, the memory_load instruction, although issued in one clock period, will take at least 18 clock periods to be completed by the functional unit, making the target register unavailable for that length of time. However, other instructions (including other loads) not requiring this register may proceed. The add instruction, needing the result, must wait for 18 clock periods before it can begin execution. The store operation will also be delayed from starting until the result from the add instruction is available but will proceed to completion concurrently with the instructions that follow the store. It, thus, becomes essential to find other (needed) instructions which can be performed during the wait for memory.

In this example, the memory_load can, perhaps, be compared with an I/O operation on most machines because the memory operation is so slow compared with the computational section.

For I/O operations, programmers quickly learn to read data prior to the point where it is required, so that, hopefully, it will be available by the time it is needed with no delay. This same concept applies to memory_loads on the Cray and the associated programming "trick" is sometimes called instruction lifting. The idea is to move the load instruction back up the instruction stream at least eighteen clock periods in much the same manner as the I/O read is moved in traditional computer programs.

This concept is simple in theory, but it causes two important difficulties that must be addressed. By "anticipating" memory_loads and doing them well before the value is required, a register must be allocated to the memory_load. This register becomes unavailable to any instructions between the start of memory_load operation and the instruction that uses the value. See Figure 4 for an example.

	target register	operand register	timing start	timing end
	S1	CON1	0	1
	S2	CON2	1	2
ENTRY	=	*		
	S3	S2+S1	2	5
	V3	S3+V0	5	80
	S1	COUNT,A1	6	24
	S1	S1+S3	24	27
	COUNT,A1	S1	27	45

Figure 4

Our discussion mainly concerns the last three instructions, those that load from memory to S1, update S1, and then store S1 back to memory. The suggested optimization technique would move the load from COUNT into S1 instruction back up the stream eighteen clock periods. Note, however, that S1 has already been used. Moving this instruction requires changing the register name for one or the other of these instructions and all associated subsequent intervening instructions. This process is likely to result in errors if the renaming process is done without extreme caution. Figure 5 illustrates one possibility.

	target register	operand register	timing start	timing end
	S4	COUNT,A1	-12	6
	.			
	.	additional instructions not using		
	.	S4 may appear here		
	.			
	S1	CON1	0	1
	S2	CON2	1	2
ENTRY	=	*		
	S3	S2+S1	2	5
	V3	S3+V0	5	80
	S1	S4+S3	6	9
	COUNT,A1	S1	9	27

Figure 5

The problem is made more difficult when there is a label between the place where the instruction is presently located and the new location where it will be moved backward in the instruction stream. Figure 5 also illustrates this problem. The label ENTRY implies that somewhere in the instruction stream a branch will/might transfer control to the label. If the memory_load instruction is moved above the label, the code sequence that branches to ENTRY will not do the memory_load instruction and may produce erroneous results. There are two approaches: 1) don't "lift" instructions past labels, or 2) "lift" above labels and then "copy" the lifted instruction and place it before all jump instructions that reference the label. Case 1 is simple but can penalize performance if there are not enough cycles between the label and where the word is required, forcing the add instruction (in this example) to wait. Case 2 is more difficult and produces confusing code where apparently extraneous memory reads appear in the instruction stream. Nevertheless, the payoff is so great that case 2 memory lifting is used extensively throughout Cray Blitz.

4.3 Address/Scalar Arithmetic Operations

The XMP/YMP computers have two separate integer scalar computation units. The normal 64-bit integer values use one set of functional units and the previously mentioned S-registers. For computations where the values represent memory addresses, the XMP/YMP has separate functional

units and registers (A-registers) These have a length of 24-bits (XMP) or 32-bits (YMP).

This yields two advantages. First, the address functional units operate faster using short values than the corresponding 64- bit functional units. Second, these two sets of functional units are completely independent. This allows parallel execution of certain types of calculations and tests when the values are acceptable for shorter arithmetic operations.

A common example is the following FORTRAN if statement:

```
if(a.gt.5 .or. b.gt.5) go to 100
```

On the XMP/YMP, one way to do the two comparisons is to fetch the value of A, subtract six from it and jump if the sign of the result is positive. This is repeated for the variable B. The hardware test procedure requires the result be put in scalar register S0 or A0 which then sets an appropriate flag so that the jump instruction executes correctly. Traditional XMP/YMP code would look like figure 6.

	target register	operand register	timing start	timing end
	S1	A,	0	18
	S2	B,	1	19
	S3	6	2	3
	S4	S1-S3	18	21
	S5	S1-S3	19	22
	S0	S4	21	22
	JSP	L100	25	
	S0	S5	27	28
	JSP	L100	31	
NEXT	=	*		

Figure 6

While the preceding code produces correct results to match the FORTRAN "if" statement, the number of cycles required to arrive at NEXT when both A and B are not greater than 5 is 33. (There is a 3 cycle delay after the result reaches S0 before it is available to the branch unit.)

The following code shows the timing when the values of A and B have been lifted and makes use of the knowledge that either a or b (or both) are small enough so that shorter arithmetic produces accurate answers. The modified code is in Figure 7; NEXT is reached at cycle 12 (note that A and B have been pre-loaded into T40 and B40 by "lifting" the memory load instructions to a point above where where the example code starts).

	target register	operand register	timing start	timing end
	S1	T40	0	1
	S3	6	1	2
	S0	S1-S3	2	5
	A1	B40	3	4
	A2	6	4	5
	A0	A1-A2	5	7
	JSP	L100	8	
	JAP	L100	10	
NEXT	=	*		

Figure 7

The important point to note is that there may be separate instruction streams using two separate sets of functional units and registers. This is an example of an optimization that compilers do not find because it is very difficult to determine if an integer value fits within a 32-bit register without being intimately familiar with the code. For programs with large numbers of if_tests, this can easily double the execution speed with no penalty other than making the assembly code somewhat less readable.

5.0 Attack Detection

Another feature of vector operations on the XMP/YMP concerns the so-called gather/scatter operations. Some programmers refer to these operations as indirect vector loads and stores. The idea is to first produce a vector of random memory addresses and then do a load or store operation using this list of addresses. There are potential performance problems since two or more of the random address can address the same memory bank, resulting in a bank-busy delay (five cycles).

In Cray Blitz, the search often needs to know if a particular square is under attack (particularly when determining if a king is in check.) The 64-bit registers of the XMP/YMP make this test extremely quick and easy. In practice, the routine

"attack(i,j)" is used to ask if square "i" is under attack by any piece of side "j".

The attack detection code requires two data structures. The first is a sixty-four bit representation of the board where any occupied square is represented by a 1-bit and an empty square is represented by a 0-bit. This bit-board is updated dynamically each time a piece is moved on the game board.

The second data structure is quite large, and is generated before the game starts and remains constant throughout the course of the game. It is a 3-dimensional array of sixty-four bit words, AT(i,j,k). (It is logically an array of size (64,64,6) but is actually somewhat larger because Cray Blitz uses border squares to aid in move generation and these must be accounted for in this array.)

The first subscript identifies the square that the code is testing to determine if it is under attack. The second subscript represents the square of a piece on the board. Since this square can contain one of six different pieces, the third subscript identifies the piece type (1=pawn, 2=knight, 3=bishop, 4=rook, 5=queen, and 6=king.)

The program maintains separate lists of the squares occupied by white and black pieces for use by this routine (and others). The way that the code uses these data structures to detect attacks is both fast and easy to understand. The algorithm works for sliding and non-sliding pieces equally, but they are described separately for clarity.

For kings, knights and pawns the question is: does such a piece on square "j" attack square "i"? Intuitively, this is a set of 64x64 flags that answer the question as true or false for any pair of squares "i" and "j". If the king, knight, or pawn can move from square "i" to square "j", then the array value for AT(i,j,piece) is set to 0 (all zeroes.) If a knight cannot move from square "i" to square "j", then AT(i,j,piece) is set to -1. The reason for -1 will become clear after the next step is explained.

For sliding pieces, the question is two-fold. First, does a sliding piece on square "i" bear on square "j" (or vice-versa, it makes no difference) and secondly, is an attack blocked by an intervening piece between square "i" and square "j"? The first part of this can be answered exactly like the previous example, but a second part is now required and is equally important to a correct answer. The question, more properly phrased, is: if a bishop (or rook or queen) on square "i" bears on square "j", what squares must be unoccupied for an attack to be true? If a bishop on square "i" does attack square "j" when the board is empty, then AT(i,j,3) is set to the bit pattern where each square between "i" and "j" (along a diagonal, naturally) are represented by ones. If this pattern is then "anded" with the bit board identifying the empty/occupied squares, and if the result is zero, then the path between the two squares is empty and the attacking condition is true. If the result of the "and" is non-zero, this indicates that one of the intervening squares is blocked and that the attacking condition is false.

To determine if square "i" is under attack by ANY black piece (recall that a list of black pieces (k) and squares they stand on "j" is maintained), the code, once for each piece, "ands" AT(i,j,k) with the bit-board representing the current state of the chess board pieces. Whenever any of these "and" operations produces a zero result, the code returns with the attack condition set to true; otherwise, all piece locations are tested before returning a false condition. Notice that for kings, knights and pawns, intervening pieces are not possible. Therefore, if a piece on square "i" attacks square "j", then the corresponding mask is zero guaranteeing that the result of the and will be a zero. If a piece on square "i" does not attack square "j", then the corresponding mask is all ones, guaranteeing that the result of the "and" will be non-zero since the bit-board can never be all zeros because the board always contains at least kings.

In summary, this attack detection method requires N "and" operations where N is set by the number of pieces for the side being tested. The gather/ scatter operations of the XMP/YMP are used to load the N masks into a vector register. This is then "vector anded" with a scalar containing the bit-board representation of the chessboard. Any zero value in the resulting vector register indicates that the attack condition is true. This code is so simple that the CFT77 FORTRAN

compiler produces code that is nearly as good as that produced by hand-coding in CAL.

Any discussion of chess programs always raises the issue of the "bit-board" representation (already described) and how it compares to the more traditional "mailbox" representation where each square on the board is stored in one computer word.

Both representations have advantages and shortcomings, but in the past, Cray Blitz has used the "mailbox" approach because it simplifies the evaluation coding significantly. However, because of the vector processing hardware on the Cray machines, the program now takes advantage of both representations. The "mailbox" board representation simplifies the evaluations while the "bit-board" representation takes advantage of the vector hardware to speed things up further. Cray Blitz represents a hybrid mixture of these two well-known techniques.

6.0 Register Usage

Another architectural feature valuable for Cray Blitz is the fast B and T registers available on the XMP/YMP. For 64-bit integers, the 64 T registers are used as temporary storage areas to back up the eight S registers used by the computational section of the machines. For 24/32-bit operations (24 on XMP's and 32 on YMP's), the 64 B registers are used in the same manner.

However, a problem occurs when mixing CAL and CFT77 and/or C code because both compilers use B and T registers for code optimization. As compilers become better at optimizing register usage, FORTRAN/C routines alter more and more of the B and T registers for temporary storage. This prevents a CAL routine from calling a FORTRAN/C routine and expecting that all B/T registers will be unaltered when the routine returns.

Our somewhat unscientific method of analyzing register usage is to fill all registers with some unusual bit pattern and then execute the FORTRAN/C code for several tests. After each test completes, the B/T registers are dumped to determine which ones are safe to use across FORTRAN/C calls. Any register with its contents unaltered is considered safe (for the current version of the compiler, at any rate) and is therefore available to contain temporary values that are required frequently (ply, depth, scores, etc.) While such an empirical method for determining which registers are volatile is quite ugly, Cray Blitz is running in an environment of operating system and compiler software development where the operating system and/or FORTRAN compiler change daily. Without such a tool, continual phone interaction would unduly prolong debugging when a new optimization renders a previously safe register volatile.

If the complete program were written in CAL, additional optimization would be possible with this large set of registers. Specifically, the entire board could be resident, avoiding the delays that occur when accessing main memory. The only draw-back to such a scheme is that the hardware requires explicitly specifying the registers used, preventing indirect references. While this does not completely prevent such usage, it would eliminate any loops that access the board, making the code somewhat longer.

The biggest shortcoming of using registers in this manner appears when a new version of a compiler is released. Suddenly, registers that were safe with the previous compiler change in random ways, leading to debugging problems and re-coding to avoid using the newly expanded set of registers altered by the compiler's new and better code.

7.0 Parallel Processing

The design of Cray Blitz takes maximum advantage of shared memory multiprocessing computer systems. The messages and requests (described later) passed from processor to processor are simply flags stored in a shared memory word that all processors test at the beginning of a node expansion. The overhead to send such a message is therefore nearly zero, and the maximum delay before a processor responds to a request is the amount of time that it takes one processor to expand a single node (roughly 40 microseconds on a Cray YMP, which includes move generation, evaluation, and updating the various data structures used to support the tree

search itself). Additional details can be found in Hyatt's thesis [7].

The current algorithm, called Dynamic Tree Splitting (DTS), extends the Principal Variation Split (PVS) algorithm described by others [1, 3, 10, 13, 15, 18]. Like DPVS introduced by Schaeffer [19], DTS addresses two major problems exhibited by the PVS algorithm: (1) PVS requires that all processors split work up at the nodes on the Principal Variation (PV), and (2) all processors synchronize at the end of these parallel searches before any can proceed to other work.

7.1 Dynamic Tree Splitting (DTS)

Whenever a processor exhausts the work (sub-tree) that it is working on, it broadcasts a help request to all busy processors. These processors make a quick copy of the *type* of each node they are searching in the current sub-tree and the number of unsearched branches at each node, and give this information to the idle processor. The busy processors then resume searching where they were interrupted. The idle processor (or processors if more than one is idle) examines the data and picks the most likely split point based on the amount of work left, and the depth of the node. The following description uses the terminology of Marsland and Popowich [13] to describe these actions . Subtrees are made up of PV, CUT and ALL type nodes (these correspond exactly to the minimal game-tree notion of type=1, 2 and 3 nodes used by Knuth and Moore [8]). Thus the idle processor selects a PV or ALL node, but never a CUT node, and then forces the selected processor to split at the chosen node. The busy processor arranges to share the data at the selected split point and then both processors continue searching from that point in parallel.

In a normal termination of these parallel searches, each processor compares its results with those from other parallel searchers and if it has a better value, it copies its search path and score over the best so far. As well as a normal completion, other outcomes are also possible. If a processor discovers a refutation to the branch that leads to a split point node, other processors working at that split point are doing unnecessary work. The processor informs the others and they immediately stop and try to find more useful work to proceed with, by broadcasting a help request. As a processor finds a new best score for a split point, it shares the value with other parallel searchers at that split point to improve their AB cutoff performance. These issues are dealt with more deeply in Hyatt's thesis [7].

7.2 Parallel Performance

Performance can be measured in many ways. One metric is to measure the search overhead introduced by parallel processing. This overhead represents work that is not done by a sequential program. In the case of Cray Blitz, the search overhead is measured in terms of number of nodes searched. Figure 8 illustrates the search overhead measured as a percent of the original sequential tree size. Note that this figure is for typical trees and does not show the occasional search overhead explosion mentioned previously. This data was produced by running a set of test positions using varying numbers of processors. Some of the problems produce almost no additional search overhead, and others sometimes produce a tremendous search explosion.

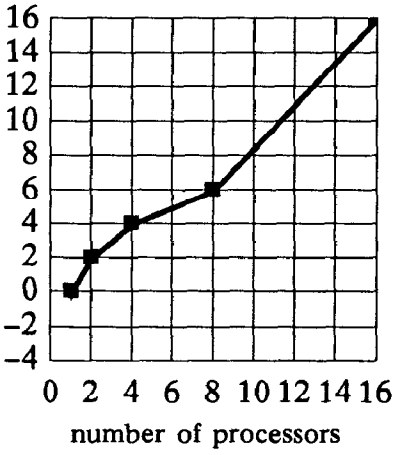

Figure 8: DTS search overhead

This metric points out the deficiency that established parallel algorithms exhibit. In theory, the search analyzes PV and ALL nodes completely and searches only one descendant from a CUT node. In practice such analysis breaks down because the perfect move ordering assumption only applies to minimal game trees, although recent analysis of average game tree search by

Reinefeld and Marsland now makes it possible to address this issue directly [17].

The DTS algorithm always tries to split ALL nodes, a reasonable goal since they are normally completely searched. What happens, however, is that when the branch chosen from a CUT node is not best, allowing one of the branches from the successor ALL node to refute it, then the ALL node behaves like a CUT node, since almost any branch searched reveals the bad move ordering at the previous supposedly CUT node. If the search chooses such an ALL node for a split point, processors will do unnecessary work since any one branch might refute the previous branch. This increases the search overhead significantly.

Some tests with Cray Blitz produce trees that increase in size very little, and for typical positions each processor seems to add less than ten percent extra nodes to the total searched. However, other positions produce trees that are sometimes two to three times the size of the sequential search. This keeps the processors busy, but they stay busy searching unnecessary parts of the tree. In summary, the only significant points are that most positions run much faster using a parallel search, and also no positions require more time for the parallel search than the sequential search.

Testing shows that a four-processor machine provides an average speedup of about 3.2 over the entire game (there is not yet enough YMP data to determine average performance for eight processors, although an educated guess would place the average speedup between 5 and 6). Some moves are near a factor of 4.0 and others drop even lower. Occasionally a move actually speeds up by more than a factor of 4.0 producing the so-called "super-linear speedup" anomaly.

Figure 9 illustrates the performance of Cray Blitz when run on a Sequent Balance 21000 computer system. This machine is dramatically slower than a Cray XMP/YMP machine, but offers more processors for parallel performance analysis. The performance curve for the Balance 21000 is not as impressive as that of the Cray machines due to the difference in search depth

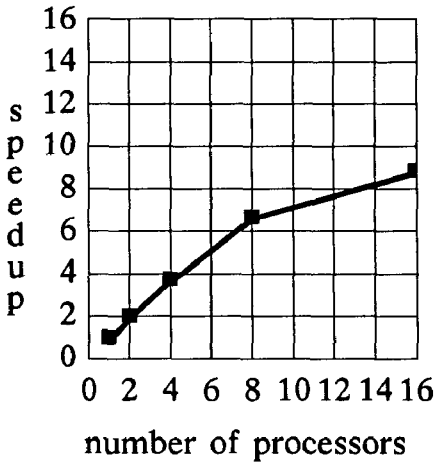

Figure 9: DTS speedup

capability between the two machines. It has been shown that deeper searches yield better parallel performance [7, 18, 19] giving a significant advantage to faster parallel machines such as the Cray YMP.

These tests produce reasonably similar performance results through eight processors although the YMP has a small but noticeable edge. As the number of processors increases, the curve quickly flattens out on both machines (Cray and Sequent), but the flattening is more noticeable on the Sequent due to the difference in search depths it attains (five plies on the Sequent, nine to ten plies on the Cray YMP, given the same search time.)

8.0 The Future

The Cray product line has distinct architectural features that were easily used to improve the performance of Cray Blitz. Sometimes the original code was modified extensively while other features were simple to utilize. While Cray Blitz is no longer the fastest computer chess program around (searching "only" some 200,000 nodes per second on a Cray YMP), it is probably in second place behind Deep Thought, a special purpose chess machine developed at Carnegie-Mellon University. It is impressive that a general purpose machine exceeds the speed of special purpose hardware machines like Belle and HiTech, to name a few.

References

1. Campbell, M.S. (1981), "Algorithms for the Parallel Search of Game Trees," M.Sc. thesis, Department of Computing Science, University of Alberta, Canada.

2. Campbell, M and T. Marsland (1983), A comparison of minimax tree search algorithms, *Artificial Intelligence* 20, pp. 347-367.

3. Finkel, R. and J. Fishburn (1982) , Parallelism in Alpha-Beta Search, *Artificial Intelligence* 19, pp. 89-106.

4. Finkel, R. and J. Fishburn (1983), Improved Speedup Bounds for Parallel Alpha-Beta Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5, pp. 89-92.

5. Hyatt, R.M., A.E. Gower and H.L. Nelson (1985), "Cray Blitz", in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, Oxford, pp. 8-18.

6. Hyatt, R.M., H. Nelson, A. Gower (1986), Cray Blitz - 1984 Chess Champion, *Telematics and Informatics* 2, pp. 299-305.

7. Hyatt, R.M (1988), *A High-performance Parallel Algorithm to Search Depth-first Game Trees*, Ph.D. thesis, Department of Computer Science, University of Alabama at Birmingham.

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

9. Marsland, T.A., M. Campbell, and A. Rivera (1980), "Parallel Search of Game Trees," Technical Report TR 80-7, Computing Science Department, University of Alberta.

10. Marsland, T.A. and M. Campbell (1982), Parallel Search of Strongly Ordered Game Trees, *ACM Computing Surveys* 14, pp. 533-551.

11. Marsland, T.A. and F. Popowich (1983), A Multiprocessor Tree-searching System Design, Technical Report TR 83-6, Department of Computing Science, University of Alberta.

12. Marsland, T.A. and M. Campbell (1983), Relative Efficiency of Alpha-Beta Implementations, *International Joint Conference on Artificial Intelligence*, Karlsruhe, pp. 763-766.

13. Marsland, T.A. and F. Popowich (1985), Parallel Game-tree Search, *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7, pp. 442-452.

14. Marsland, T.A., M. Olafsson, and J. Schaeffer (1986), Multiprocessor Tree-Search Experiments, in: *D. Beal, Ed., Advances in Computer Chess 4*, Pergammon Press, pp. 37-51.

15. Newborn, M.M. (1985), "A Parallel Search Chess Program," ACM Annual Conference, Denver, pp. 272-277.

16. Newborn, M.M. (1988), "Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 5, pp. 687-694.

17. Reinefeld, A. and T.A. Marsland (1987), "A quantitative Analysis of Minimal Window Search," 10th International Joint Conference on Artificial Intelligence, pp. 951-954.

18. Schaeffer, J. (1987), Experiments in distributed game-tree searching, Technical Report TR87-2, Computing Science Department, University of Alberta.

19. Schaeffer, J. (1989), "Distributed Game-Tree Search," to be published in the *Journal of Parallel and Distributed Computing* in 1990.