NEWS, INFORMATION, TOURNAMENTS AND REPORTS

PARALLEL CHESS ON THE CRAY X-MP/48

Robert M. Hyatt
University of South Mississippi

INTRODUCTION

The chess program Cray Blitz is the current World Computer Chess Champion and became the current North American Computer Chess Champion in the 1984 ACM tournament. The program has also played in human chess tournaments and has the strength of a chess master. At speed chess, where an ability to perform very accurate analyses is particularly important, it has maintained a performance rating of over 2600 for the past two years. This indicates that, at speed chess, the program is one of the top players in the world, either electronic or human. It is currently running on a Cray X-MP/48 computer system and has been designed around the parallelism that the X-MP architecture provides.

There have been three major versions of the program that have used Cray computers. Version one was created for the Cray-1 and took advantage of some of the special Cray architectural features such as vector processing, instruction overlap, and its large number of registers. Version two was designed for the X-MP/24, the first multiprocessing Cray computer system. Version three was designed for better use of the X-MP/24 and the new X-MP/48 computers since version two failed efficiently to utilize both processors.

VERSION ONE

Version one of Cray Blitz was originated in 1980 by a conversion of Blitz, a computer-chess program written in 1976, to the Cray computer system. The initial effort resulted in a substantial speed-up of the program and eventually yielded a program that could examine approximately 1,000 chess positions per second. Additional reprogramming to eliminate certain programming practices known to be time-consuming on the Cray-1 (mod function, integer divide and others) resulted in the present pure Fortran speed of approximately 4,500 nodes per second. The most successful chess program of the time was the program Chess 4.x written at Northwestern University and run on a CDC Cyber 176. It was examining approximately 2,600 nodes per second which made version one of Cray Blitz look quite good. However, study revealed that while CFT (Cray Fortran) was producing good code, careful hand coding of the routines frequently used could result in even further improvement.

After considerable timing analysis was done, we carefully isolated those routines needing hand coding and methodically attacked the difficult and laborious task. Each new CAL (Cray Assembly Language) routine resulted in a measurable performance increase, and the end result was that the program analyzed over 20,000 nodes per second on a single processor. This effort is still ongoing and is continously increasing the performance of the program. However, there was a point beyond which additonal hand coding had no effect on the search time. As this point was reached, the older CAL routines were re-examined for better algorithmics, since the CAL routines were simply 'conversions' of the FORTRAN routines they replaced. Better algorithmics resulted in further increases in search speed, and in the area of CAL programming, this effort is still underway.

Additionally, algorithms were developed specifically for the Cray (using CAL) effectively to take advantage of the vector hardware. By using the unusual properties of the vector hardware and of the parallelism of the functional units, further improvements were made as we discovered that some things could be added to the CAL, with no penalty whatsoever since a fair amount of time is spent waiting for memory accesses.

Early in 1982, a new computer called Cray-X became available at the Mendota Heights computer center. As it was compatible with the Cray-1, we quickly moved the program to the new system and were happy to observe a 30 percent performance improvement with little work on our part. However, the machine had two processors and we were letting 50 percent of the total machine idle away when we were playing chess. Since there was little hope of more CAL or additional algorithmic improvements giving us a performance improvement of 50 percent, we began to think about ways to use both processors in a parallel manner.

VERSION TWO

As the Summer of 1983 passed, we progressed along our projected path of improvements while preparing for the upcoming World Computer Chess Championship in October. Since Cray multitasking software was still under development during this time, our thoughts about parallelism were wishful at best. In late August, with the World Championship two months away, we decided to test our latest code by playing speed chess in Pasadena, California. We were using a Cray-1M, which seemed to be approximately 50 percent slower than the X-MP/24 (one processor, since we had no parallel algorithm) we would be using for the World Championship. Our goal was to test the program for bugs or weaknesses, not really caring how we would do in the speed tournament. The program performed quite well and lost only two games, showing that even in its 'slower' state on the Cray-1M, it was still quite dangerous, as several International Chess Masters found out.

After the tournament, we were quite surprised when the people at Cray Research software development started asking if we could use two processors simultaneously. It turned out that, while we were improving and debugging the program, Cray Research had developed the multitasking operating system and Fortran interface to it and were looking for applications to test it. Our dilemma was that we had only two months before the World Championship, and we had not even started on a parallel version.

After discussing the possibilities for a day, and keeping the idle CPU in the back of our minds, we decided to emulate Rocky and go for it. Due to the extremely short time for writing, testing, and debugging a parallel program, with a brand-new operating system, and no experience in parallel programming, we decided that we had a chance of getting it together. Since planning dooms most projects, we felt that we were a cinch to succeed in the conversion since we had <u>no</u> time for planning or discussing our approach.

Due to the short time available, we decided to take the simplest approach to multitasking. The program had a list of n moves to examine in order to choose the move it would make. We elected to divide this search process between the two processors and overlap examining two moves at a time. In order to avoid load-balancing problems, we also elected to use a self-scheduling algorithm where each processor took a move from the move list, and after examining it, took another. When no further moves were left, the results of the two processors (each one had found the best move of the ones it searched) were compared and the best move was selected.

In order to implement such a parallel search on short notice, we decided to completely duplicate the tree-search code, and affix a 'Z' to each module name that was duplicated and to each common block that was duplicated. We were wary of trying to debug a re-entrant code with task common, since we had a significant number of CAL routines that would have to be extensively modified to become re-entrant. This approach worked and actually gave very few problems that had not been anticipated.

The first problem was performance and had been anticipated. With a chess position, the first move searched takes longer to examine than any of the remaining moves. The explanation is that when the first move is examined, a program (or a human) has no idea of how good or bad the position is. After examining the first move, and discovering that a pawn can be won, other moves can be examined much more quickly because if one of them seems to lose a pawn, no further analysis is needed since the first move wins one. However, if the first move loses a pawn, the program (or the human) must still examine it carefully as ALL moves might lose a pawn, and it is still necessary to play the best move. Since each processor analyzed a different first move, they both had to search them carefully. After each had finished its first move, things proceeded quite rapidly and the remainder of the moves were examined essentially twice as fast as with one processor. The drawback is that the entire move list is not examined twice as fast due to the first two moves taking a substantial amount of time.

This algorithm in some rare cases was no better than one processor (viz. when there was only one legal move), but was generally 1.5 to 1.8 times faster than one processor. In certain rare cases, this program was actually many times faster than one processor. If one processor selected its first move and had to spend minutes examining it due to the complexity of the resulting position, the other processor could finish its first move and continue processing the remaining moves. It might actually find a crushing move while the first processor is still working on the first move. Since a single processor would have to work through the first move to get to the winning move, this case looked quite attractive for the parallel design. However, this was a rare case.

Figure 1 illustrates typical search improvements obtained with the parallel search by comparing total time (wall clock) required for one and two processors. The exact chess positions are irrelevant in this story. Case 1 is the worst-case improvement where there is more than one legal move since only one move will result in no improvement. Since the program never changes its mind, a fair amount of time is wasted by the second processor while it is examining its first move. Case 2 is the optimum case, where the program changes its mind several times. This example results in a performance improvement of 2.0 over one processor. Again, this is optimum and reality lies between 1.5 and 1.8 during the course of a complete game.

Case 1 (no multiprocessing)

```
Depth  Time   Eval  Variation

 5a    0:01  0.383  Bxf3 Qxf3 0-0 0-0 c5
 6a    0:07  0.251  Bxf3 Qxf3 0-0 Nc3 c5 0-0
 7a    0:30  0.305  Bxf3 Qxf3 0-0 Nc3 c5 0-0 h6
 8a    2:46  0.221  Bxf3 Qxf3 e5 Be2 0-0 0-0 Ne4 fxe5 Nxe5
Time: 3:25 100% Nodes: 4,981,731 H 31% 99% 95% NPS: 24,242
```

Case 1 (with multiprocessing)

```
Depth  Time   Eval  Variation

 5a    0:01  0.383  Bxf3 Qxf3 0-0 0-0 c5
 6a    0:04  0.251  Bxf3 Qxf3 0-0 Nc3 c5 0-0
 7a    0:19  0.305  Bxf3 Qxf3 0-0 Nc3 c5 0-0 h6
 8a    1:48  0.221  Bxf3 Qxf3 e5 Be2 0-0 0-0 Ne4 fxe5 Nxe5
Time: 2:13 199% Nodes: 6,545,639 H 25% 96% 92%.NPS: 49,123
```

Net improvement with two processors: 1.5

Case 2 (no multiprocessing)

Depth	Time	Eval	Variation
6a	0:01	0.504	Nd5 e4 fxe4 fxe4 Nf6 Re1
6a	0:01	1.160	h6 Nh3 Re6 Nf4 Rxd6 Ke1
7a	0:09	1.310	h6 Nh3 Re6 Nf4 Rxd6 Ke1 Rd8
8a	0:44	1.120	h6 Nh3 Rac8 Nf4 Rc6 Ke1 Rxd6 Rc1
8a	1:43	1.237	Bc4 Ke1 h6 Nh3 ...
8a	3:07	1.280	Rac8 a3 Rc2 Rc1 Ra2 Rc7 Ra1+ Bc1 Rxe3 Rxa7
Time: 3:20 100% Nodes: 3,887,306 H 40% 99% 96% NPS: 19,606			

Case 2 (with multiprocessing)

Depth	Time	Eval	Variation
6a	0:01	0.504	Nd5 e4 fxe4 fxe4 Nf6 Re1
6b	0:01	1.160	h6 Nh3 Re6 Nf4 Rxd6 Ke1
7a	0:04	1.310	h6 Nh3 Re6 Nf4 Rxd6 Ke1 Rd8
8a	0:23	1.120	h6 Nh3 Rac8 Nf4 Rc6 Ke1 Rxd6 Rc1
8b	0:51	1.237	Bc4 Ke1 h6 Nh3 ...
8a	1:31	1.280	Rac8 a3 Rc2 Rc1 Ra2 Rc7 Ra1+ Bc1 Rxe3 Rxa7
Time: 1:41 197% Nodes: 3,987,721 H 40% 99% 96% NPS: 39,877			

Net improvement with two processors: 1.97

Figure 1

From the first box:

Time: 3:25 is equivalent to 100% as fast as single processing;

Nodes examined: 4,981,731;

Use of hash tables: 31% (Transposition table), 99% (Pawn table), 94% (King table); (cf. H.L. Nelson (1985). Hash Tables in Cray Blitz. ICCA Journal, Vol. 8, No. 1, pp. 3-13). NPS means nodes per second.

We were succesful in developing, coding, testing and debugging the new parallel program during the next two months. We did experience substantial problems with the CAL routines. Removing the CAL slowed the program by a factor of 4.5, so it was obvious that in making a choice between CAL or a parallel search, we would choose CAL.

With the first round game scheduled for Saturday night, Friday morning found us without a working version of the program. A concentrated effort Saturday afternoon allowed us to get enough of the CAL working in the parallel version such that we were faster with some CAL and parallelism than we were with all CAL and no parallelism. Without having played a game of any kind, and only having tested on a set of problem positions, we started round one. Two dozen packs of Lifesavers, 18 soft drinks, and a gross of Tylenol later, we had picked up our first win. After the round-one game, we stayed up all night and got all of the CAL operational (minus one fairly important routine) for the next round. The rest is history as Cray Blitz went on to defeat Belle (from Bell Labs) and won the tournament. It looked FAR easier than it actually was!

VERSION THREE

Early in 1984, the newly announced X-MP/48 was installed in the Mendota Heights computer center. The availability of this machine had two important features that would assist us. First, it was a four-processor machine with eight million words of memory. Second, it had two new instructions, GATHER and SCATTER which we felt would be of benefit in the CAL routines.

It was obvious to us that the algorithm used to win the world championship was poorly suited to more than two processors. We were already experiencing load-balancing problems when one processor started examining a move that required a tremendous tree search while the other processor would finish the rest of the moves and have nothing left to do. Since the work was being divided up into rather coarse chunks, we began to redesign the tree search to be more efficient on four (and more) processors.

Since maintaining two separate copies of the tree-search code was inconvenient and resulted in numerous problems where one copy was modified and the other was delayed (and eventually forgotten), the decision was made to design a re-entrant code using the features of CFT, such as task common, local data, and shared or globol common. The program was reduced from approximately 20,000 lines of Fortran and 20,000 lines of CAL to approximately 14,000 lines of Fortran and 10,000 lines of CAL. This was much more convenient than having two of everything to change.

The algorithm we finally used allowed all processors to work on the first move together, and then look at the remaining moves independently as before. This prevented much duplication of effort and also made the analysis of the first move proceed four times faster than before.

To explain the algorithm, assume the program is doing an eight-ply exhaustive search. That is, it is searching everything to a depth of eight half-moves and then searching beyond that point very selectively to make sure it is not overlooking something. The new algorithm forces one processor to follow the tree through the first seven levels while the other processors remain idle. At depth eight, the other three processors join the search and each processor looks at the next available move at depth eight and searches as deeply as is necessary. When all moves at depth eight have been examined, all processors synchronize and compare results. The best move is reported back to the previous level. This move completes the analysis for the first move at the previous level, and now the three remaining processors join the first in examining the remaining moves at depth seven. When they finish, they back up to depth six and search the remaining moves there. This proceeds until they finally get back to depth one and divide up the remaining moves there after they have all worked on the first move.

Early performance measurements with this algorithm on the X-MP/48 yielded performance improvements of 3.5 to 4 over the one-processor code. It was apparent that the new algorithm was far superior to the old one, and was actually smaller in terms of lines of code. Figure 2 illustrates the results obtained by running the same two tests of Figure 1 on the new algorithm. As expected, case two shows no improvement, since it was already optimal, but case one shows that the factor 1.5 could be improved upon quite a bit.

Notice that there is a fairly uniform improvement when adding processors, although each processor accumulates some idle time due to improper load-balancing and conflicts with other processors over shared-memory data structures.

Case 1

CPUs	time	improvement factor
1	3:25	----
2	1:44	1.92
3	1:13	2.80
4	0:55	3.71

Case 2

CPUs	time	improvement factor
1	3:20	----
2	1:42	1.95
3	1:09	2.88
4	0:52	3.85

Figure 2

Load balancing is now the major concern of our new programming developments. The tree being searched is not a perfectly symmetrical tree where each branch goes to exactly the same depth (or height), and each node does not have the same number of moves to examine. The probability is high that if there are exactly four moves at a level, and each processor examines one in parallel, the processors will not complete at the same time, and the cumulative wait times can actually become quite large. As an example, in the first round of the 15th Annual North American Computer Chess Championship, we saw a total wait time for the game of 40 percent of one processor, which corresponds to running approximately 3.6 times faster than a single processor would have. However, in the next round, the game was more complicated which aggravated the load-balancing problem and resulting in the performance dropping off to 'only' 3.1 where we lost 90 percent of one processor due to lack of work for it.

The major problem seems to center around the concept of each processor looking at a different ply-one move (after they all look at the first one). In the case where they each start on one of the last four moves at ply one, it is highly probable that they will not finish at the same time. If the difference in times between the longest and shortest is substantial, a severe performance penalty is incurred, as we saw in the second round of the tournament. To help minimize this discrepancy in times between moves, each processor searches its move with a very narrow search window. This tends to minimize the timing differences among the moves. If one processor then finds that its move is better than the current best move, the other processors are stopped and then they all work on this new candidate before continuing on the moves they were working on.

A final technique that is being used is to count the number of nodes examined when searching each move when doing a depth n-1 search. Then, when the depth n search starts, we order the ply-one moves in order of decreasing node counts. This puts the complicated moves at the top of the list and minimizes the difference in time required for the moves near the bottom of the list. Of course, a move that appeared simple at depth 7 might turn out to be extremely complicated at depth 8 and disrupt this planning, but in general the idea has proven effective in actual tests.

THE FUTURE

It is apparent that additional parallelism is around the corner with a sixteen-processor Cray coming soon and no limit in sight. The concept of each processor working at the same depth begins to fail beyond four processors. For example, the average number of moves at any level averages 38 for the entire game. If more than 38 processors are available, the probability is high that many of them will reach positions where there are not enough moves to go around and they must sit idle. Tests with more than four processors on an X-MP/48 have verified that there is a point beyond which additional processors will yield no performance improvement with the present algorithm.

An attempt at using a large number of processors is currently under investigation and is based upon the idea of dividing the large group of 'n' processors into 'm' smaller groups of 'n/m' processors. Each group would be given a sub-tree to examine, and then divides that sub-tree up into 'n/m' sub-sub-trees for analysis by each processor. The effect is to reduce the number of processors evaluating a particular node to a number far less than the number of moves to be examined.

Each attempt at better load balancing seems to bring out two harmful side effects; namely, more overhead due to multiprocessing synchronization, and less efficiency where additional nodes are examined that would not be examined by a uniprocessor search. The extra nodes have historically remained well below one percent of the total nodes searched, but the more recent algorithms seem to let this percentage grow unnecessarily. Some extra work is permissible as long as the extra work done does not consume the processor time gained by reducing the cumulative idle times through trying to perform better load balancing.