Applied Numerical Mathematics 12 (1993) 285–295
North-Holland



APNUM 398

# Bits and pieces: constructing chess endgame databases on parallel and vector architectures *

Burton Wendroff
*T-Division, Los Alamos National Laboratory, Los Alamos, NM 87454, USA*

Tony Warnock
*C-Division, Los Alamos National Laboratory, Los Alamos, NM 87454, USA*

Lewis Stiller **
*Advanced Computing Laboratory and Center for Nonlinear Studies, Los Alamos National Laboratory, Los Alamos, NM 87454, USA; and*
*Department of Computer Science, The Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218-2686, USA*

Dean Mayer
*University of California at San Diego, San Diego, CA, USA*

Ralph Brickner
*C-Division, Los Alamos National Laboratory, Los Alamos, NM 87454, USA*

*Abstract*

Wendroff, B., T. Warnock, L. Stiller, D. Mayer and R. Brickner, Bits and pieces: constructing chess endgame databases on parallel and vector architectures, Applied Numerical Mathematics 12 (1993) 285–295.

An endgame database for chess encodes optimal lines of play for a specific endgame involving a small number of pieces. The computation of such a database is feasible for as many as six pieces provided the inherent parallelism in the problem is fully exploited. Two computer architectures which can do this are the SIMD CM-2 and the vector multiprocessor YMP. For each machine the computer programs operate on sets of chess positions, each position represented by a single bit. The high-level algorithm is an iterative backtracking

*Correspondence to*: B. Wendroff, T-Division, Los Alamos National Laboratory, Los Alamos, NM 87454, USA.
* Supported by the U.S. Department of Energy under contract W-7405-ENG.36.
** Partially supported by NSF/DARPA Grant CCR-8908092 and ONR Grant DAALOS-92-G-0345.

procedure. After describing endgame databases and classifying the complexity of endgames we outline the algorithms and give some details of their implementation in Fortran for parallel and vector architectures. For endgames with five or more pieces it is important to use the symmetries of the chess board to reduce storage requirements, and we indicate briefly how this can be done for the vector architecture. Some timing comparisons are presented.

## 1. Introduction

Computers are having a significant impact on the game of chess. There are many computers which play chess, and some of these have even achieved the skill of a weak grandmaster [2]. There is software which stores games in a database which can then be used to study various openings or can be used to analyze the style of play of a future opponent [1]. What we will consider in this paper is the use of computers to create an endgame database [9]. In the endgame there are a small number of pieces left on the chess board, and one can imagine describing all the correct lines of play. For the chess community such databases provide unequivocally correct move sequences from any position in the database, and have been used to challenge accepted wisdom about the outcome of various games. For the computer scientist endgames provide an interesting and open-ended computational problem—when all $n$-piece endgames have been solved we need only go on to consider $(n + 1)$-piece games. All five-piece endgame databases [5] and almost all pawnless six-piece databases [4] have been created, although archival storage of the latter is not available.

This paper is primarily directed at an audience normally dealing with scientific computations such as computational fluid dynamics. In contrast to those problems, endgame database construction involves parallel logical operations on sets of bits. In the hope that it will be instructive for those doing large scale scientific computations, we present the basic algorithms in some detail in the form of code fragments, and we discuss how these perform for smaller endgames on two supercomputers, the Connection Machine CM-2 [6], and the Cray YMP8/128. The algorithm was originally devised for the Connection Machine [5], but, as we shall show, it has a natural implementation on the Cray. Since the time the work described here was implemented, the CM-2 upgraded to a CM-200 [7]. The CM-200 is a generally similar architecture to the CM-2, except that it has a faster clock and improved routing microcode: the CM code runs approximately 40 per cent faster on the CM-200 than on the CM-2. However, since the code was developed for the CM-2, and most of the runs occurred on the CM-2, we will discuss the CM-2 version in the remainder of the paper.

The CM-2 is a single-instruction multiple-data (SIMD) distributed-memory machine. It has 64K processors connected on a 16-dimensional hypercube. Each processor has a 1-bit bandwidth and a 8-kilobyte memory, and operates at approximately 7 mh. During any one clock (machine cycle) each processor is either doing the same instruction as every other processor or is idle. However, for scientific computation, which are often floating-point-intensive, the 1-bit processors are typically not used. Instead, the CM-2 is configured as 2K 64-bit wide Weitek vector units interconnected to form an 11-dimensional hypercube. However, as we shall see, the bit-serial architecture is convenient for the chess endgame problem, so we did not use the Weitek units explicitly in our endgame code.

The YMP8/128 is an 8-processor 128-million-word shared-memory machine operating at 166 mh. The bandwidth for both machines is 64 bits. The processors operate asynchronously on

independent instruction streams. They communicate with each other primarily through the shared memory, but also through special registers. The processors have vector and scalar functional units which operate in parallel.

Using an analysis similar to that in [3], we can get a very rough idea of the kind of relative performance we might expect on these two machines by the following argument. On the CM-2 we could think of each processor as representing a chess position. Take as a performance measure the clock rate of 7 mh times 64K, the number of positions on which an operation can be performed simultaneously. This number is 448. On the YMP we will think of each bit in the 64-bit word as representing a chess position, so an operation on a single word treats 64 positions simultaneously. In addition, the parallel vector functional units can do two logical operations, two loads, and a store simultaneously. With eight processors working the YMP does $8 * 5 * 64 = 2.56$K simultaneous operations. Multiplying by 166 mhz we get the performance measure of 425. Thus, it is not unreasonable to expect similar performance on this problem for the two machines. Of course, the actual behavior depends critically on programming details.

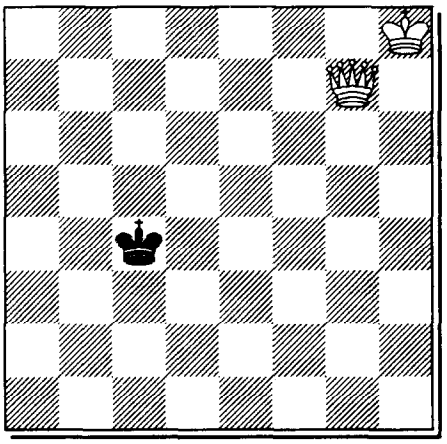
## 2. Endgame databases

Each particular collection of pieces defines a particular endgame. Piece names are abbreviated by the first letter of their names, except for knights which are denoted by the letter N. In describing endgames, the white pieces are listed followed by the black ones. Thus, KRK denotes the endgame in which white has king and rook, and black has only his king. KBNKN is the ending with the white king, bishop, and knight and the black king and knight.

A chessboard has 64 squares and there are $64^n$ ways to put $n$ distinct objects on a chessboard. A database which contains information about every such arrangement need have at most $64^n$ items. In a chess endgame database the objects are chess pieces and the information stored is a small integer defined as the number of moves to "win" for the strong side (usually white) assuming optimal play by both sides, that is, assuming that the winning side is trying to win in as few moves as possible while the losing side is trying to put off the loss as long as possible. The term "win" may have several different meanings for different databases depending on the chess goal. Typically, white has a win if black can be mated with the given pieces on the board or if white can capture a black piece leaving a lost position for black in an endgame with fewer pieces. Logically, then, the database is an array of integers $M(p_1, \ldots, p_n)$, where each $p_i$ represents the location of the $i$th piece. This together with a program which can generate moves from a given position and can do a shallow minimax search is all that is needed to play perfect chess in the endgame.

For example, suppose the ending is KQK. The position in Fig. 1 will be found in the database as a mate in 10. Now, order white's possible moves in some way and create the position $P$ resulting from the first move. $P$ is a black-to-move position. Create the set of white-to-move positions resulting from all of black's possible moves from $P$. If any one of these is in the database as a mate in more than nine moves, try the next white move. Eventually a white move will be found with the property that black has no choice but to move to a position which is a win in nine for white. This is repeated after black makes his response, but now looking for a win in one less move, and so on until black is mated. A move sequence found this way is shown in Fig. 1.

Table 1 shows some typical max-to-win values for different sizes of endgames.



**1. ♕g7-f8, ♚c4-b5 2. ♕f8-a3, ♚b5-c6 3. ♕a3-b4, ♚c6-d5 4. ♔h8-g7, ♚d5-c6 5. ♕b4-a5, ♚c6-d7 6. ♕a5-c5, ♚d7-e8 7. ♕c5-a7, ♚e8-d8 8. ♔g7-f6, ♚d8-c8 9. ♔f6-e6, ♚c8-d8 10. ♕d7 checkmate**

Fig. 1. A win-in-10 position from the endgame KQK. White and black each are playing optimally, although equally good alternatives have not been shown. The rows of the chessboard are numbered from 1–8 starting at the bottom, and the columns of the chessboard are lettered from a–h starting at the left. Thus, white's first move above is to move the queen one square northwest.

## 3. Complexity

We give a rough classification of endings according to their complexity.

**Definition 1.** An endgame is *trivial* if neither side can mate from any position of the given game.

Examples of trivial endgames are KK, KBK, and KNK.

**Definition 2.** A *subgame* of an endgame is any endgame resulting by deleting one or more pieces.

Thus, a subgame of KBNK is KBK.

**Definition 3.** A *simple* endgame is one with no pawns all of whose subgames are trivial.

Table 1
Some typical max-to-win values for different-sized endgames. The "Nodes" field is the number of nodes in the endgame, but it can be reduced slightly by more aggressive treatment of symmetry. As the max-to-win value increases, the endgames become increasingly difficult to comprehend. The last two had been considered draws before computer analysis

| Endgame | Nodes | Max-to-win |
|---|---|---|
| KQK | 40960 | 10 |
| KRKN | 2621440 | 27 |
| KBBKN | 167772160 | 66 |
| KRBKNN | 10737418240 | 223 |

Examples of simple endgames are KQK, KRK, KBNK, KBBK, and KNNK.

**Definition 4.** An *n-simple* endgame is a pawnless endgame all of whose subgames are $(n-1)$ simple, where we identify the 0-simple endgames with the trivial ones and the 1-simple endgames with the simple ones.

For example, KBNKN is a 2-simple endgame since each subgame is either simple or trivial. On the other hand, KRBKR is not 2-simple since it has 2-simple subgames. From the chess point of view these two endgames are equally complex, but not from the programmer's viewpoint, the latter being more difficult. KBNKN is the most complicated ending we will consider.

**Definition 5.** A *complex* ending is any ending with pawns.

Because pawns can change type by promotion, these endings are difficult for both chess players and chess programmers.

## 4. Retrograde analysis

A procedure for constructing an endgame database for chess was described precisely in [8]. The idea, called retrograde analysis, is to work backward from the goal of checkmate using *unmoves* instead of moves. A simple unmove of a chess piece follows the same rules as a move except that after an unmove, say by white, the resulting position must be legal for white to move. Captures cannot be made on an unmove, so one cannot unmove to an occupied square. Permitted are uncaptures, wherein a piece of the opposite color is left behind. This happens when pieces of the opposite color occupy the same square, which is the way that captured pieces are represented.

The database for simple endgames is done as follows. We start with a set $B_0$ of all positions in which black is checkmated, that is, these are all the (chessically) legal black-to-move positions for which black is in check and has no legal moves. By Definition 2, all pieces will be present. For all the positions in $B_0$ find all the positions reached by all the white unmoves. There will be no uncaptures since in legal positions no two pieces can be on the same square. If $K_1$ is this set, then $K_1$ is the set of all legal positions from which white can mate in one move. Now, let $C$ be the set of all positions reached by all the black unmoves from positions in $K_1$. Again, there will be no uncaptures. Each position in $C$ has to be tested to see if black can escape, that is, if there is a black move which creates a position not in $K_1$. Note that a capture by black automatically escapes, which is correct for simple endgames. Each position with an escape has to be removed from $C$. With what is left form $B_1 = C \cup B_0$. These are the legal positions from which black loses in one move or less. Form $K_2$ by performing white unmoves on $B_1$. Let $W_2 = K_2 - K_1$. These are the positions from which white wins in exactly two moves. This process is repeated until a $K_{i+1}$ is reached such that $K_{i+1} - K_i$ is empty. Setting $W_1 = K_1$, the database consists of the collection $\{P,j\}$ for all positions $P$ in $W_j$, $j = 1, \ldots, i$.

In order to determine the positions in $C$ from which black can escape it seems necessary to try all the *moves* from a given position to see if an escape exists. This step can be eliminated by having black unmove from the set of positions *not* in $K_1$ (which must include the positions with one of white's pieces captured). The result is the set of positions which give black an escape [4]. Call this set $E$. Now white can unmove from the set of positions *not* in $E$ to get $K_2$. The actual computation is slightly different, since stalemates must be taken into account.

For nonsimple endgames the situation is much more complicated since all possible subgames formed by captures on either side must be included.

## 5. Two implementations

Let us look further into the implementation of retrograde analysis. For any chess position $P$ let $\Pi_W(P)$ be the set of all positions reached by white's unmoving from $P$. For example, then,

$$K_2 = \bigcup_{P \in E} \Pi_W(P).$$

This will be done in a loop like,

**for all** positions $P$
  **if** $P \in E$ **then**
    $K_2 = K_2 \cup \Pi_W(P)$.

However, each unmove forming $\Pi_W(P)$ can be constructed from elementary unmoves such as "unmove the rook one square to the east". Letting $\pi_m$ represent an elementary unmove, we have

**for all** positions $P$
  **if** $P \in E$ **then**
    **for all** elementary unmoves $\pi_m$
      $K_2 = K_2 \cup \pi_m(P)$.

We could just as well interchange the loops to get

**for all** elementary unmoves $\pi_m$
  **for all** positions $P$
    **if** $P \in E$ **then**
      $K_2 = K_2 \cup \pi_m(P)$.

The point to this is that in this form the loop over $P$ is fully vectorizable with the proper data structure, whereas the first form offers little opportunity for vectorization. The second form was actually designed with the CM-2 in mind: on that machine each processor represents a group of positions and the processors simultaneously perform $\pi_m$. On the Cray, $\pi_m$ involves motions of words or shifts of bits within a word; on the CM-2, $\pi_m$ involves motions of bits from processor to processor or shifts of bits in the memory of each processor. Additional parallelization can be achieved on the Cray by asynchronous splitting of the loop over $P$ among its processors.

On both machines the test for $P$ in $E$ is not done explicitly but is taken care of by the data structure. This is necessary in order to produce the needed regularity of the algorithm and means that a lot of the computation is redundant. Each iteration going from $K_n$ to $K_{n+1}$ requires exactly the same amount of computation. For that reason it is clear that the predecessor of the complement version is appropriate. This is not the case with the first form of the algorithm, where the amount of work depends on the number of positions being processed. For some endings the complement of $K_1$ will be very dense and it will be more efficient to compute black's forward moves.

Our experience with the first algorithm is that it is at least an order of magnitude slower than the second.

## 6. Elementary moves as motions

Chess pieces move according to definite rules. On an otherwise empty board a rook can move any number of squares along its row or column (rank or file), bishops move diagonally any number of squares, queens move as both a rook and a bishop, and kings move as queens but only one square. Knights jump to the next column and then two rows away or vice versa. No piece can unmove to or past an occupied square.

For the moment let us think in terms of Fortran 77 and, for definiteness, the ending KRK. The objects we want the compute are sets of positions such as $B_0$, $K_i$, etc. These can be treated as two-dimensional arrays of 64-bit words with dimension

dimension $B_0(64, 64)$,

with the convention that the first argument is the position of the white king, the second argument is the position of the black king. Here, we have supposed that the squares of the chess board have been numbered from 1 to 64 in some way. The third piece, the white rook, is represented by the appropriate bit position in the word. Let $A(i,j)_k$ be the $k$th bit in the word $A(i,j)$. For example, the set $B_0$ is equivalent to the array $B_0(\cdot, \cdot)$ if

$$B_0(i,j)_k = \begin{cases} 1, & \text{if position } (i, j, k) \text{ in } B_0, \\ 0, & \text{otherwise.} \end{cases}$$

In other words, the array $B_0(\cdot, \cdot)$ is the characteristic function of the set $B_0$.

We can also look at this geometrically, thinking of the universe of all possible chess positions as a cube of $64^3$ boxes. A particular cube represents a set of positions if and only if whenever position $(i, j, k)$ is in the set the box at $(i, j, k)$ contains a 1. To find the result of unmoving the rook one square to the east for all the positions in the set we need to take the contents of every box and move along the rook axis of the cube. Thus,

```
do 10 i = 1,64
  do 10 j = 1,64
    temp(i, j) = shiftr(A(i, j), 1)
10 continue
```

creates the set *temp* of all positions obtained by unmoving the rook one square to the east (if the bit positions in the word are appropriately ordered) from all the positions in $A$. There will

be some illegal positions in *temp*, but these can easily be removed by a masking operation. We can then repeat this loop with *temp* on the right to get the two-square moves, and so on. Taking the union of the results produces the set of all rook unmoves to the east positions. Then the whole thing is repeated for west, north, and south. To unmove the white king to the east we move contents of boxes along the white king axis. The loop is

```
do 10 i = 1,64
  do 10 j = 1,64
    temp(i, j) = A(i − 1, j)
10 continue
```

and so on.

More generally, with more pieces one piece will be "in word" and the others will be "in array". We should mention that we could just as well have used a two-dimensional representation of the chessboard rather than a one-dimensional one for either or both of the two kings.

The data structure on the CM-2 is very similar. In an *n-piece* endgame $n-1$ pieces will be "in processor", that is, the set of all possible configurations of $n-1$ pieces will be mapped 1–1 onto the set of virtual processors. Here, however, the two-dimensional chessboard representation is appropriate. The $n$th piece resides "in ram", that is, 64 bit locations in the memory of each processor define the possible positions of the $n$th piece. We will use a the CM Fortran dialect of Fortran 90, except that CM Fortran does not yet support 64-bit integers. The statements

```
INTEGER * 8 A, TMP,
DIMENSION A(64, 64), B(64, 64),
CMF$LAYOUT A(:NEWS, :NEWS), B(:NEWS, :NEWS)
```

do the mapping, while

```
temp = 0,
a = 0
```

initialize the arrays. Then a white king unmove is done with

```
forall (i = 1:64, j = 1:64) temp(i, j) = A(i − 1, j)
```

which moves the 64 bits in each processor at location $A$ to location *temp* in the "next" processor along the white king column axis. The "in ram" rook unmove is taken care of with

```
temp = shiftr(a, 1)
```

which shifts the 64 bits at location $A$ in each processor over one place and puts the result in location *temp*. As with the Fortran 77, masks are used to filter the illegal positions.

## 7. Symmetry

We indicated at the beginning of Section 2 that there are at most $64^n$ possible configurations of $n$ chess pieces. It is well known that for pawnless endgames many of these positions are equivalent under the symmetries of the chessboard. For example, if $P$ is a position which is

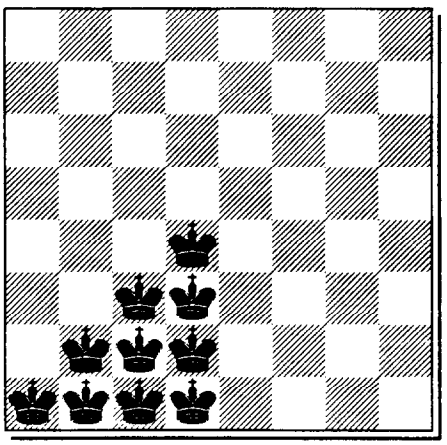


Fig. 2. The ten possible positions of the black king. No matter where the black king is in a position, the position can be reflected and rotated so that is brought into the 10-square octant shown.

checkmate for black then so are all the positions obtained by reflecting the position about either main diagonal or about the vertical or horizontal midlines or any composition of these transformations, so long as the position does not contain any pawns. In fact, the group of symmetries of the chess positions is simply $D_4$, the dihedral group of order 8.

By exploiting this symmetry, the storage requirements and the time requirements of the computation can be reduced. In the most abstract formulation, we are trying to reduce the computational requirements of a search on a state space invariant under symmetry. Although this leads to deep and interesting theory, the essential characteristics of the treatment of symmetry can be understood without recourse to the group-theoretic machinery.

The key observation is that any chess position without pawns can be rotated and reflected in such a manner that the black king is on one of the ten squares in the lower left-hand octant of the board (see Fig. 2).

Therefore, we have reduced the number of possible positions of the black king from 64 to 10. Instead of using a $64 \times 64 \times 64$ array of 64-bit integers to store the set of positions in a four-piece endgame, in which the first axis encodes the position of the black king, we only need to use a $10 \times 64 \times 64$ array of 64-bit integers.

Unfortunately, there is a price to pay in terms of code complexity for this data layout. Basically, all the pieces other than the black king can be moved just as before. However, sometimes when the black king moves, it will move out of the 10 squares to which it is restricted. When this happens the entire chessboard, including the other pieces, must be rotated and reflected so as to bring the black king into the octant to which it is restricted.

This transformation process is difficult to do on both architectures. On the wide-word architecture implementation, some piece is stored "intra-word". The transform on this piece therefore induces a bit permutation of the bits of each word. Furthermore, this permutation must be done for each of the 7 non-identity transforms at each iteration of the main loop. A naive method to do this would be to actually move the bits one at a time. However, such 1-bit operations are inefficient on a wide-word machine. A much more efficient way is to use lookup tables to implement the transformation. Each 64-bit word within which the permutation is to be

implemented is divided into four 16-bit fields. For each permutation, for each of the four 16-bit fields, a lookup table is used that gives the 64-bit word to which that 16-bit field is mapped. The final transform will simply be the logical bitwise OR of the four values that are looked up. The "inter-word" permutations use a gather operation on a pre-computed list of addresses.

On the CM-2 architecture, the bit permutation is fairly straightforward. Since the architecture is bit-serial anyway, it is easy to perform this; just as important, the system software supports the variable-length bit-fields we use. Unfortunately, on that architecture, the motion of the other pieces induces a general communication operation among all the processors, which operates at lower bandwidth than intra-processor operations.

## 8. Timing

The parallel CM-2 code requires between about 1 and 2 seconds per iteration for a five-piece endgame [4], depending on the load on the front end of the CM-2 and depending on the particular endgame and certain configuration choices in the program. We think that the vector implementation could require a comparable amount of time. We have a code for the endgame KBNKN running on one CPU of a YMP. It does all the unmove operations, including the permutation operation needed for the symmetry constraint, in about 3 seconds per iteration. It does not handle all the induced subgames correctly, so we cannot quote a time for the complete endgame. We also do not have a multitasking version for the YMP, but the outer loops of the Fortran code are easily multitasked.

For the four-piece endgame KBKN one CPU of the YMP takes 0.12 seconds per iteration, while one sequencer (16K processors) of the CM-2 uses more than 0.16 seconds.

However, by running in slicewise mode [3] and using the vector implementation on each node, we think a significant speedup in the CM-2 code could be obtained.

## Acknowledgment

We would like to take this opportunity to thank Professor Noam Elkies of the Department of Mathematics at Harvard University for his ongoing contributions to the chess endgame project.

This work used the computing resources of the Advanced Computing Laboratory at the Los Alamos National Laboratory, Los Alamos, NM 87545.

## References

[1] F.-H. Hsu, T. Anantharaman, M.S. Campbell and A. Nowatzyk, A grandmaster chess machine, *Sci. Amer.* 263 (4) (1990).

[2] F.-H. Hsu, Large scale parallelization of alpha-beta search: an algorithmic and architectural study with computer chess, Tech. Report CMU-CS-90-108, Carnegie-Mellon University, Pittsburgh, PA (1990).

[3] D. Smitley and K. Iobst, Bit-serial SIMD in the CM-2 and the Cray-2, *J. Parallel Distributed Comput.* 11 (1991) 135–145.

[4] L. Stiller, Group graphs and computational symmetry on massively parallel architecture, *J. Supercomput.* 5 (2–3) (1991) 99–117.

[5] L. Stiller, Parallel analysis of certain endgames, *Internat. Comput. Chess Assoc. J.* 12 (2) (1989) 55–64.
[6] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary* (Thinking Machines Corporation, Cambridge, MA, 1990).
[7] Thinking Machines Corporation, *Connection Machine CM-200 Technical Summary* (Thinking Machines Corporation, Cambridge, MA, 1991).
[8] K. Thompson, Retrograde analysis of certain endgames, *Internat. Comput. Chess Assoc. J.* 9 (3) (1986) 131–139.
[9] H.J. van den Herik and I.S. Herschberg, The construction of an omniscient endgame database, *Internat. Comput. Chess Assoc. J.* 8 (2) (1985) 66–87.