Parallel Implementation of Combinatorial Algorithms

Norman Matloff

March 13, 2000

1 Overview

Many applications of parallel processing methods involve **combinatorial algorithms**. These involve searching of huge state spaces for an optimal, or approximately optimal, solution to some problem.

1.1 The 8 Queens Problem

A famous example of this is the 8 Queens Problem, in which one wishes to places eight queens on a standard 8x8 chessboard in such a way that no queen is attacking any other. (The generalization, of course, involves n queens on an nxn board.) Suppose our goal is to find all possible solutions.

To start a solution to this problem, we first note in any solution will have the property that no row will contain more than one queen. This suggests building up a solution row by row: Suppose we have successfully placed queens so far in rows 0, 1, ..., k-1 (row 0 being the top row of the board). Where can we place a queen in row k? Well, since we cannot use any column already occupied by the preceding k queens, that means we have a choice of 8-k columns. But even among those k columns, there will be j of them ($0 \leq j \leq 8-k$) that are in the diagonal attack path of some preceding queen. Then we can extend our tentative k-row solution to8-k-j k+1 row solutions.

For example, with k = 2 we may have built up a tentative solution as follows: Define

```
struct TentSoln {
   int RowsSoFar;
   int Cols[8];
   struct TentSoln *Next;
}
```

The array Cols has the interpretation that if Col[I] == J means that we have placed a queen in row I, column J of the board. RowsSoFar is the number of rows in which we have placed queens so far. Next points to the next item in the work pool.

A parallel solution based on this idea would like something like this:

```
while (work pool nonempty or at least one nonidle processor) {
   get a TentSoln struct from the work pool, and point P to it;
```

```
   I = P->RowsSoFar;
   for (J = 0; J < 8; J++) {
      if (a queen at row I, column J would not attack the previous queens) {
         Q = malloc(sizeof(struct TentSoln));
         Q->RowsSoFar+1;
         add the struct pointed to by Q to the work pool;
      }
   }
}
```

There of course would also be code in the case I = 8 to check and see if we have found a solution, and if so, to report it, etc.

Note that any rotation of a solution—interchanging rows and columns—is also a solution. Similarly, any reflection across one of the two main diagonals of the board is also a solution. This information could be used to speed up computation, though at the expense of additionality complexity of the code.

1.2 The 8-Square Puzzle Problem

This game was invented more than 100 years ago. Here is what a typical board position looks like:

0	5	3
1	4	
7	2	6

(The real puzzle has numbering from 1 to 8, but we use 0 to 7.)

Each number is on a little movable square, which can be moved up, down, left and right as long as the spot in the given direction is unoccupied. In the example above, the square 3, for instance, could be moved downward, producing an empty spot at the top right of the puzzle. The object of the game is arrange the squares in ascending numerical order, with square 0 at the upper left of the puzzle (which in this example happens to be the case already).

We again solve this by setting up a work pool, in this case a pool of board positions. Each board position would be implemented in something like this:

```
struct BoardPos   {
   int Row[9];
   int Col[9];
   struct BoardPos *Next;
}
```

Here Row[I] and Col[I] would be the position of the square numbered I. For convenience, we also store the location of the position, in Row[8] and Col[8].

Suppose a processor goes to the work pool and gets the board position depicted above. In the simplest form of the algorithm, the processor would check each of the three possible moves (4 right, 3 down, 6 up) to see if the resulting board position would duplicate one that had already been checked. All moves that lead to

new positions would be added to the work pool. Each processor would loop around, pulling items from the work pool, until some processor somewhere finds a solution to the game (in which case that processor would add termination messages to the work pool, so that the other processors knew to stop). An outline of the algorithm would be as follows:

```
while (work pool nonempty or at least one nonidle processor) {
   get a BoardPos struct from the work pool, and point P to it;
   for (I = 0; I < 8; I++)  {
      for all possible moves of square I do  {
         Q = malloc(sizeof(struct BoardPos));
         fill in *Q according to this move;
         if *Q has not already been checked
            add this board to the work pool;
      }
   }
}
```

Again, code would need to be included for checking to see if a solution has been found, whether we have found that no solution exists, and so on.

Note the operation

```
         if *Q has not already been checked
            add this board to the work pool;
```

Clearly this is needed, to avoid endless cycling. But it is not as inoccuous as it looks. If the set of all previously-checked board positions is to be made available to all processors, this may produce substantial increases in contention for memory and interprocessor interconnects. On the other hand, we could arrange the code such that only certain processors have to know about certain subsets of the set of previously-checked board positions, but this makes the code more complex and may produce load-balancing problems.

A more sophisticated version of the algorithm would use a **branch-and-bound** technique. The idea here is to reduce computation by giving priority in the work pool to those board positions which appear "promising" by some reasonable measure. For example, we could take as our measure the "distance" between a given board position and the goal board position, as defined by the sum of the distances from each numbered square to its place in the winning position. In the example above, for instance, the square numbered 5 is a distance of 2 from its ultimate place (2 meaning, one square to the right, one square down, so 1+1 = 2). The board above is a distance 15 from the winning board.

The idea, then would be that we implement the work pool as an ordered linked list (or other ordered structure), and when a board position is added to the work pool, we insert it according to its distance from the winning board. This way the processors will usually work on the more promising boards, and thus hopefully reach the solution faster.