CS 677 High-performance Computing
Programming Assignment #4

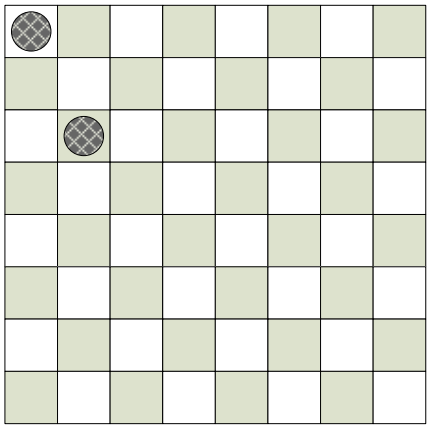
Message-passing: Distributed Solutions to the *k*-Queens Problem

Overview

The assignment is to use MPI to write a distributed program for solving the *k*-Queens problem. The basic idea is to place *k* queens on a *k* x *k* chessboard such that no two queens can attack each other. In other words, no two queens can be in the same row, the same column, or on the same diagonal.

Background

Solving this problem is essentially a search for valid solutions. One way to help reduce the search space is to work across the board from the leftmost columns. For example, consider the traditional chessboard and the classic 8-queens problem (see figure below). The first move is to place a queen in the first row of the first column. Next, place the second queen anywhere in the second column such that it is "safe". Note that it cannot be placed in row 1 or in row 2 (it is on the diagonal), but it can be safely placed in row 3. Continue until all 8 queens have been placed. If at any time a queen cannot be placed in a particular column, then backtrack to the previous column and try placing its queen in a different row (i.e. backtrack up the search tree).

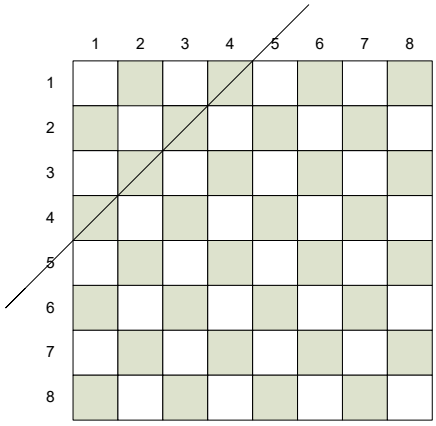


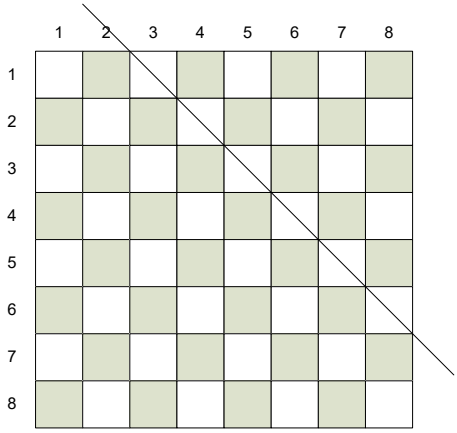
Typically, queen placement is represented by indicating its row. For example, the sequence (7, 5, 3, 1, 6, 8, 2, 4) represents a valid solution, in which the queen in column 1 is located in row 7, the queen in column 2 is located in row 5, etc.

Note: this is a typical search problem. The approach involves generating possible solutions, evaluating each of them, and finally reporting valid results. The "tree" organization of the search space allows for a more directed search. The basic idea of the parallel implementation is to practice message-passing programming: partition the workload, execute in parallel, communicate the results.

Wirth's Algorithm

Many of the solutions to the original Eight Queens problem use backtracking based on an algorithm due to Wirth. Obviously, two queens cannot be on the same row, and that is easy enough to check by maintaining an array of placed queen positions. Now, consider the diagonal as seen in the figure below to the left. Notice that each of the "threatened" squares can be identified as (Row# + Column# = 5). The Right-Hand Side (i.e. the sum) will differ, but the equality relation holds for all 15 possible diagonals of the same orientation. Therefore, similar to the row check, the legality of placing a queen in one of these diagonal positions can be checked by maintaining an array of already placed queens. Finally, consider the diagonal in the figure below to the right. Each of its "threatened" squares can be identified as (Row# – Column# = –2). Again, this relation holds for all diagonals of the same orientation and can be easily checked.





The psuedocode snippet below provides the structure of Wirth's algorithm. Iteration is used to examine all possible locations to place a queen. The conditional checks if a specific position is valid, and recursion is used to implement backtracking.

```
void place_queen (int col)
{
    if (all queens are placed) {
        Solutions++
    }
    else {
        // try all rows
        for (row=1; row <= k; row++) {
            // check if queen can be placed safely
            if (no queens on this row or on these diagonals)
                place queen on this row
                // try rest of problem
                place_queen (col+1);
                // returned from recursion
                // do not place queen on this row (backtrack)
            }
        }
    }
}
```

Work Partitioning

Consider the straightforward approach to dividing the workload. Using *k* processors, each processor *i* could begin by placing its first queen in row *i* of the first column and continue its search from there. Clearly, each processor would be generating different possible solutions (i.e. evaluating a different branch of the search tree).

The straightforward approach is reasonable, but it is not without its issues. Note that because of backtracking, the system may duplicate work in different processes. Also, this approach induces an unbalanced workload, as there are a different number of potential solutions that begin with a queen placed in each of the various rows. Finally, regardless of the size of the cluster, it limits the degree of parallelism to *k*.

Think about your parallel design. There are different scheduling approaches and workload methods you can implement and experiment with.

Requirements / Deliverables:

Implement a program that solves the *k*-Queens problem:

- You must develop a correct, generalized MPI-based solution; the Arch lab is available for use as a virtual cluster.
    - List all 92 solutions for the original 8-queens problem. This can be used to validate the correctness of your parallel implementation.
    - Determine the number of valid solutions for various values of *k*. This can be used to evaluate performance.
- Conduct a timing-based performance evaluation/analysis.
- Submit a hard-copy of your source-code, output, and performance analysis. Be prepared to present your results in class.