Solutions to the n Queens Problem Using Tasking in Ada

by

Russell M. Clapp, Trevor N. Mudge, Richard A. Volz

June 26, 1986

Introduction

This article discusses solutions for the n Queens problem written in Ada. The n Queens problem can be stated as follows: How can n queens be placed on an n by n chess board so that no two queens lie on the same row, column, or diagonal? There is no solution to this problem for $n \leq 3$. For $n \geq 4$, there is always more than one solution. The programs presented here find all solutions for a given n.

We have written two programs using Ada tasking to solve this problem. These solutions were compared to each other and to a third program, Wirth's solution slightly modified and translated from Pascal to Ada. This solution appears in its original form in [1]. The main basis for comparison was execution time, but the number of tasks needed for the first two programs was also examined.

The Sequential Solution

This program is Wirth's solution coded in Ada and modified to provide all possible solutions to the n Queens problem given a particular n. The algorithm proceeds by placing a queen on a square and then checking for a safe square to put a queen in the next column. A safe square is one where a queen may be placed without making the current partial solution invalid. This process begins in the first column, and a solution is found if safe squares are found up to and including the nth column. If no safe square is found in a column, the algorithm backtracks to a previous column where a safe square can be found. The program uses recursion and iteration to explore all possible solutions.

This program did provide some data structures used in the other solutions. Three one dimensional arrays are used to hold bit values indicating the presence of a queen in a row, diagonal, or reverse diagonal. A square can be checked to determine if it lies in the path of a previously placed queen by referencing each of these arrays using the proper combination of the square's row and column numbers. When a queen is placed on the board, these arrays are updated accordingly.

The Task Spawning Solution

This program uses the most concurrency of the three solutions. Each time a queen is placed on a board, a new task is created to find safe squares in the next column to place the next queen. The process starts in the first column, and since the board is initially empty, n tasks are created, each with one queen in a different row in the first column. Each of these tasks then attempts to find safe squares in the next column. Whenever one is found, a new task is created with one more queen placed on the board in that column. Since all safe squares are found, one task may have several child tasks, each with an identical board configuration except for the last occupied column. If a task can find no safe squares in the next column, it does not spawn any child tasks and that solution path is terminated. This program, then, finds all solutions to the problem concurrently.

In a machine with multiple processors, this program may be able to find all solutions quickly if a processor was available to execute each new task created. However, if the machine has multiple processors with a shared memory accessed by a common bus, only one task creation may occur at a time. This is due to the fact that a task creation requires code to be copied to a processor from memory over the bus.

The program can be coded so that all information a task needs is passed to it in a rendezvous, leaving task creation as the only operation that requires use of the shared memory. However, depending on the architecture of the machine, task rendezvous may also require use of the bus. Excessive task creation and message passing then would reduce the parallelism and slow down the computation. This would be the result for even modest values of n, as the number of tasks required appears to increase exponentially in n. A hypercube machine with interconnected processors would speed up the task rendezvous, but would not be well suited to copying code to a new processor for each task creation.

The Square Solution

This program has less concurrent execution than the previous one, but uses significantly fewer tasks. One task is created for each square on the chess board. Each task of this type is identical. Each task in the first column begins at the same time with a queen placed on its square. It proceeds to look for safe squares in the next column. When a safe square is found, a rendezvous occurs with the task of the safe square, passing it the board configuration of the current task with a queen added in the acceptor's square. The accepting task then looks for safe squares in the column to its right, repeating the process. All solutions are found since all possible solutions are explored concurrently. It is possible that a square may be safe for more than one potential solution at the same time. In this case, the calling tasks are queued on the entry point and serviced one at a time. This results in less concurrency than the previous program. Potential solutions that become invalid terminate in a manner similar to that mentioned above.

The advantage of this solution over the previous one is the reduced number of tasks needed. This program requires only n^2 tasks to execute, but uses less concurrency. There is also the possibility of tasks going unused with this solution. In tests involving values of n ranging from 4 to 10, only two tasks went unused in two cases. These were the tasks for the upper right and lower right squares in the $n = 4$ and $n = 6$ cases.

This program could find all solutions quickly if it were run on a multiple processor machine with n^2 processors available. The program can easily be coded so that all information that a task needs is passed to it in a rendezvous. There is no need for shared memory. Thus, this program is well suited to be run on a hypercube machine, with all tasks running in parallel.

Results

The programs were timed using the CLOCK function provided by Ada in the package CALENDAR. The Verdix Version 5.2 compiler was used running on a VAX 11/750 with Unix 4.2 bsd and no other user processes. The timing was done to show the relative efficiency between the programs and includes I/O time. Since the programs were run on a single processor time shared system, the sequential solution was fastest since it involved no tasking overhead. The results are summarized in the following table.

n	Sequential	Spawn	Square
	Execution Time (in seconds)		
4	0.2	0.4	0.5
5	0.4	1.0	1.1
6	0.4	2.2	1.4
7	1.8	8.2	4.0
8	4.9	32.4	11.0

References

[1] - Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

```
with TEXT_IO, CALENDAR;
use TEXT_IO, CALENDAR;
procedure SEQUENTIAL is

        package MY_INT_IO is new INTEGER_IO(INTEGER);
        use MY_INT_IO;
        package TIME_IO is new FIXED_IO(DURATION);
        use TIME_IO;

        NUMBER_OF_QUEENS:constant := 7;
        LOW_DIAG:constant := 1 - NUMBER_OF_QUEENS;

        type CONFIG_ARRAY is array(INTEGER range <>) of BOOLEAN;
        type COL_ARRAY is array(INTEGER range <>) of INTEGER;

        TIME_BEGIN, TIME_END : TIME;
        EXECUTION_TIME : DURATION;

        ROW:    CONFIG_ARRAY(1..NUMBER_OF_QUEENS)          := (others => TRUE);
        R_DIAG: CONFIG_ARRAY(2..(2*NUMBER_OF_QUEENS))      := (others => TRUE);
        DIAG:   CONFIG_ARRAY(LOW_DIAG..NUMBER_OF_QUEENS-1):= (others => TRUE);
        COL:    COL_ARRAY(1..NUMBER_OF_QUEENS)             := (others => 0);

        procedure TRY(I: INTEGER) is
        SAFE: BOOLEAN;
        J: INTEGER := 0;
        begin
                while (J < NUMBER_OF_QUEENS) loop
                        J := J + 1;
                        SAFE := ROW(J) and R_DIAG(I+J) and DIAG(I-J);
                        if SAFE then
                                COL(I) := J;
                                ROW(J)      := FALSE;
                                R_DIAG(I+J) := FALSE;
                                DIAG(I-J)   := FALSE;
                                if  I < NUMBER_OF_QUEENS then
                                        TRY(I+1);
                                        ROW(J)      := TRUE;
                                        R_DIAG(I+J) := TRUE;
                                        DIAG(I-J)   := TRUE;
                                else
                        PUT("Solution follows as row numbers in column order:");
                                    for K in 1..NUMBER_OF_QUEENS loop
                                       PUT(COL(K), 0);
                                       if K < NUMBER_OF_QUEENS then PUT(", ");
                                       end if;
                                    end loop;
                                    NEW_LINE;

                                    ROW(J)      := TRUE;
                                    R_DIAG(I+J) := TRUE;
                                    DIAG(I-J)   := TRUE;

                                end if;
                        end if;
```

```
                end loop;
        end TRY;


begin

        TIME_BEGIN := CLOCK;
        TRY(1);
        TIME_END := CLOCK;
        EXECUTION_TIME := TIME_END - TIME_BEGIN;
        PUT("Execution time for solution ( in seconds) is : ");
        PUT(EXECUTION_TIME, 3, 1);
        NEW_LINE;

end SEQUENTIAL;
```

```
with TEXT_IO, CALENDAR;
use TEXT_IO, CALENDAR;
procedure SPAWN is

--
--     This program uses many tasks to find all solutions for the n-queens
-- problem.  A task type is used for tasks that represent nodes on a path
-- in a solution tree.  There is also a task for controlling I/O requests.
--
--     This program incorporates some basic information about the n-queens
-- problem that simplifies the approach.  First, a valid solution may have
-- only one queen in any row or column.  Second, a one dimensional array may
-- be used to encode information about queens in diagonals.  (A separate array
-- is needed for reverse diagonals).  This reduces the check for a queen in a
-- diagonal to one array reference.  This technique was taken from Wirth's
-- solution involving recursion.
--
--     The program finds all solutions concurently.  One task begins examining
-- squares in the first column.  All squares in this column are safe for
-- potential solutions since the board is initially empty.  So, for each safe
-- square, a queen is placed on the square and a new task is created to check
-- squares in the next column.  Initially, this means that n tasks are created
-- each with a different board configuration, checking for safe squares in the
-- second column.  This task creation then, proceeds from the first column to
-- the last.  If a particular task can find no safe squares in the next column,
-- that potential solution is invalid and that thread of execution terminates.
-- When a safe square is found in the nth column, a solution is found.  A
-- rendezvous with the I/O controlling task occurs before the solution is
-- printed so that separate solutions are not written over each other.
--
--     The program also keeps track of the number of tasks created, and prints
-- this total.  The amount of time needed to execute the program is also printed
-- upon completion.
--

   package INT_IO is new INTEGER_IO(INTEGER);
   use INT_IO;
   package TIME_IO is new FIXED_IO(DURATION);
   use TIME_IO;

   NUMBER_OF_QUEENS:constant := 8;

   COUNT:INTEGER:=0;                    --Count the number of tasks created
   TIME_BEGIN, TIME_END:TIME;
   EXECUTION_TIME:DURATION;

begin

  declare

-- Configuration data structure adapted from Wirth's solution

   LOW_DIAG :constant:= 1 - NUMBER_OF_QUEENS;

   type CONFIG_ARRAY is array (INTEGER range <>) of BOOLEAN;
   type COL_ARRAY is array (INTEGER range <>) of INTEGER;
```

```
type CONFIGURATION is
   record
      ROW:    CONFIG_ARRAY(1..NUMBER_OF_QUEENS)           := (others => TRUE);
      R_DIAG: CONFIG_ARRAY(2..(2*NUMBER_OF_QUEENS))       := (others => TRUE);
      DIAG:   CONFIG_ARRAY(LOW_DIAG..NUMBER_OF_QUEENS-1) := (others => TRUE);
      COL:    COL_ARRAY(1..NUMBER_OF_QUEENS)              := (others => 0);
   end record;

task IO_CONTROL is
   entry START;
   entry STOP;
end IO_CONTROL;

task type NODE is
   entry START(COLUMN:INTEGER; GRID:CONFIGURATION);
end NODE;

type A_OF_T is array(1..1) of NODE;  --Necessary because 'new NODE' is not
                                     -- allowed in the body of NODE

type A_T is access A_OF_T;

FIRST_LEVEL:A_T;
LAY_OUT:CONFIGURATION;

task body NODE is

   BOARD:CONFIGURATION;
   COL:INTEGER;
   ROW:INTEGER := 0;
   NODE_PTR:A_T;
   SAFE:BOOLEAN;

   begin  --The task T

      COUNT := COUNT + 1;       --Count the task after it is created

      accept START(COLUMN:INTEGER; GRID:CONFIGURATION) do
         COL := COLUMN;       --Read configuration from parent task
         BOARD := GRID;
      end START;

      while ROW < NUMBER_OF_QUEENS loop   --For each row in this column

        ROW := ROW + 1;
        SAFE := BOARD.ROW(ROW)
                and BOARD.R_DIAG(COL+ROW)
                and BOARD.DIAG(COL-ROW);      --Find safe squares
        if SAFE then
           BOARD.COL(COL) := ROW;           --Place queen on square
           if COL = NUMBER_OF_QUEENS then  --Found a solution
              IO_CONTROL.START;

              PUT("Solution, row numbers in column order: ");
              for I in 1..NUMBER_OF_QUEENS loop
```

```
                INT_IO.PUT(BOARD.COL(I), 0);
                if I /= NUMBER_OF_QUEENS then PUT(", "); end if;
             end loop;
             NEW_LINE;

             IO_CONTROL.STOP;
          else
             BOARD.ROW(ROW)          := FALSE;   --Place queen on board
             BOARD.R_DIAG(COL+ROW)   := FALSE;
             BOARD.DIAG(COL-ROW)     := FALSE;

             NODE_PTR := new A_OF_T;              --Create child task
             NODE_PTR.all(1).START(COL+1, BOARD); --Pass info to child task

             BOARD.ROW(ROW)          := TRUE;  --Take queen off board for next
             BOARD.R_DIAG(COL+ROW)   := TRUE;  --iteration on this column
             BOARD.DIAG(COL-ROW)     := TRUE;
          end if;
       end if;
    end loop;
  end NODE;

task body IO_CONTROL is

   begin

      loop

       select

         accept START;
         accept STOP;

       or

         terminate;

       end select;

      end loop;

   end IO_CONTROL;

begin    --declare

   TIME_BEGIN := CLOCK;

   FIRST_LEVEL := new A_OF_T;             --Create first task
   FIRST_LEVEL.all(1).START(1, LAY_OUT); --Pass it empty configuration

end; --declare  Do not proceed past this point until all tasks are completed.

TIME_END := CLOCK;

PUT("The total number of tasks created for this solution is: ");
PUT(COUNT, 0);
```

```
    NEW_LINE;
    EXECUTION_TIME := TIME_END - TIME_BEGIN;
    PUT("Execution time for solution (in seconds) is          : ");
    PUT(EXECUTION_TIME, 3, 1);
    NEW_LINE;

end SPAWN;
```

```
with TEXT_IO, CALENDAR;
use TEXT_IO, CALENDAR;
procedure SQUARE is

--
--      This solution uses n-squared tasks to solve the n-queens problem.
-- There is one task for each square on the chess board.  There is also
-- a task for controlling I/O requests.  This program finds all possible
-- solutions to the n-queens problem, for a particular n.
--
--      The program incorporates some basic information about the n-queens
-- problem that simplifies the approach.  First, a valid solution may have
-- only one queen in any row or column.  Second, a one dimensional array
-- may be used to encode information about queens in diagonals.  This reduces
-- the check for a queen in a diagonal to one array reference.  This technique
-- was taken from Wirth's solution involving recursion.
--
--      The program finds all solutions concurrently.  First, each task in the
-- first column begins, assuming it is the only square in the solution thus far.
-- This means that n different threads of execution occur at the start.  Each
-- task then checks the next column for 'safe' squares, i.e. squares where a
-- queen may be placed in proceeding with a potential solution from the current
-- square.  Then, the task rendezvous with each safe square in the next column
-- passing it the current board configuration.  This has the potential of
-- splitting that particular thread of execution into several paths.  If a task
-- can find no safe squares in the next column, that thread of execution
-- terminates.  If a safe square is found in the nth column, a solution is
-- found.  A rendezvous with the I/O controlling task occurs before the solution
-- is printed so that separate solutions are not written over each other.
--
--      The program also keeps track of task usage, and prints a message if any
-- tasks go unused.  The amount of time needed to execute the program is also
-- printed upon completion.

   package INT_IO is new INTEGER_IO(INTEGER);
   use INT_IO;
   package TIME_IO is new FIXED_IO(DURATION);
   use TIME_IO;

   NUMBER_OF_QUEENS:constant := 8;

-- For keeping track of task usage
   type USAGE_ARRAY is array(INTEGER range <>, INTEGER range <>) of BOOLEAN;
   STATS:USAGE_ARRAY(1..NUMBER_OF_QUEENS, 1..NUMBER_OF_QUEENS) :=
                                              (others => (others => FALSE));

   TIME_BEGIN, TIME_END:TIME;
   EXECUTION_TIME:DURATION;

begin    --Main

   declare

-- Configuration data structure adapted from Wirth's solution

   LOW_DIAG :constant:= 1 - NUMBER_OF_QUEENS;
```

```
type CONFIG_ARRAY is array (INTEGER range <>) of BOOLEAN;
type COL_ARRAY is array (INTEGER range <>) of INTEGER;

type CONFIGURATION is
   record
      ROW:    CONFIG_ARRAY(1..NUMBER_OF_QUEENS)           := (others => TRUE);
      R_DIAG: CONFIG_ARRAY(2..(2*NUMBER_OF_QUEENS))       := (others => TRUE);
      DIAG:   CONFIG_ARRAY(LOW_DIAG..NUMBER_OF_QUEENS-1)  := (others => TRUE);
      COL:    COL_ARRAY(1..NUMBER_OF_QUEENS)              := (others => 0);
   end record;

type ORD_PAIR is
   record
      ROW:INTEGER;
      COL:INTEGER;
   end record;

task IO_CONTROL is
   entry START;
   entry STOP;
end IO_CONTROL;

task type SQUARE is
   entry BRANCH(BOARD:CONFIGURATION; POSITION:ORD_PAIR);
end SQUARE;

type A_SQUARE is access SQUARE;

type TASK_ARRAY is array(1..NUMBER_OF_QUEENS, 1..NUMBER_OF_QUEENS)
                of A_SQUARE;

SQUARE_ARRAY:TASK_ARRAY;
EMPTY_CONFIG:CONFIGURATION;
FIRST_CONFIG:CONFIGURATION;
FIRST_POSITION:ORD_PAIR;

task body SQUARE is   -- n squared of these tasks referenced by TASK_ARRAY

   LAY_OUT:CONFIGURATION;
   POS:ORD_PAIR;
   NEW_BOARD:CONFIGURATION;
   NEW_POSITION:ORD_PAIR;

   begin  --The task SQUARE

    loop

     select

      accept BRANCH(BOARD:CONFIGURATION; POSITION:ORD_PAIR) do
         LAY_OUT := BOARD;    -- Read data from task in previous column
         POS := POSITION;
      end BRANCH;

      STATS(POS.ROW, POS.COL) := TRUE; --Indicate that the task has been used
```

```
      if POS.COL = NUMBER_OF_QUEENS then     -- A solution
         IO_CONTROL.START;                   -- Lock out writers

         PUT("Solution, row numbers in column order: ");
         for J in 1..NUMBER_OF_QUEENS loop
            INT_IO.PUT(LAY_OUT.COL(J), 0);
            if J /= NUMBER_OF_QUEENS then PUT(", "); end if;
         end loop;
         NEW_LINE;

         IO_CONTROL.STOP;
      else

         for I in 1..NUMBER_OF_QUEENS loop     --Check each row in next column
            NEW_BOARD := LAY_OUT;      --Initialize board to empty next column
            if NEW_BOARD.ROW(I)                --Check for safe square
               and NEW_BOARD.R_DIAG(I+POS.COL+1)
               and NEW_BOARD.DIAG(I-(POS.COL+1))   then   -- Pass it on

                  -- Place queen on square for next task
               NEW_BOARD.COL(POS.COL + 1)     := I;
               NEW_BOARD.ROW(I)               := FALSE;
               NEW_BOARD.R_DIAG(I+POS.COL+1)  := FALSE;
               NEW_BOARD.DIAG(I-(POS.COL+1))  := FALSE;
               NEW_POSITION.ROW := I;
               NEW_POSITION.COL := POS.COL + 1;

                 --Pass on info to safe square in next column
               SQUARE_ARRAY(NEW_POSITION.ROW, NEW_POSITION.COL).BRANCH(
                                                  NEW_BOARD, NEW_POSITION);
            end if;
         end loop;

       end if;

      or

       terminate;

      end select;

     end loop;

   end SQUARE;


task body IO_CONTROL is

   begin

      loop

       select

         accept START;
```

```
            accept STOP;

          or

            terminate;

          end select;

        end loop;

      end IO_CONTROL;

   begin   --declare

     TIME_BEGIN := CLOCK;

     for I in 1..NUMBER_OF_QUEENS loop    --Create tasks for each square
        for J in 1..NUMBER_OF_QUEENS loop
           SQUARE_ARRAY(I, J) := new SQUARE;
        end loop;
     end loop;

     for I in 1..NUMBER_OF_QUEENS loop    --For each row in the first colum
        FIRST_CONFIG              := EMPTY_CONFIG;   --Re-initialize to empty
        FIRST_CONFIG.COL(1)       := I;
        FIRST_CONFIG.ROW(I)       := FALSE;     --Mark occupied square
        FIRST_CONFIG.R_DIAG(I+1)  := FALSE;
        FIRST_CONFIG.DIAG(I-1)    := FALSE;
        FIRST_POSITION.ROW := I;
        FIRST_POSITION.COL := 1;

            --Pass on data to first column task
        SQUARE_ARRAY(I, 1).BRANCH(FIRST_CONFIG, FIRST_POSITION);
     end loop;

end;    --declare     Do not proceed until all tasks completed.

     TIME_END := CLOCK;

     for I in 1..NUMBER_OF_QUEENS loop    --Print out unused task info
        for J in 1..NUMBER_OF_QUEENS loop
           if not STATS(I, J) then
              PUT("Task for board square (row, col) ");
              PUT(I, 0); PUT(", "); PUT(J, 0);
              PUT_LINE(" was not used.");
           end if;
        end loop;
     end loop;

     EXECUTION_TIME := TIME_END - TIME_BEGIN;
     PUT("Execution time for solution (in seconds) is :");
     PUT(EXECUTION_TIME, 3, 1);
     NEW_LINE;

end SQUARE;
```

(9 pages of output not printed -Ed.