# A Specimen MPI Application: N-Queens in Parallel

**Timothy J Rolfe**
Department of Computer Science
Eastern Washington University
Cheney, Washington 99004-2493 USA
Timothy.Rolfe@mail.ewu.edu

*Abstract*: The generalized problem of placing *n* queens on an *n*-by-*n* board provides an "embarrassingly parallel" problem for parallel solution. This paper expands on the discussion presented in the May 2005 issue of *Dr. Dobb's Journal* [1], specifically taking the parallel execution through Java threads and bringing it into an application in C taking advantage of MPI.

*General Terms*: Algorithms, Performance
*Categories and Subject Descriptors*: F.2.2 Nonnumerical Algorithms and Problems — Computations on discrete structures; G.2.1 [Discrete Mathematics] Combinatorics (permutations and combinations); G.2.3 [Discrete Mathematics] Applications
*Keywords*: Backtracking, Parallel Processing, MPI

## 1. INTRODUCTION

In an article in the May 2005 issue of *Dr. Dobb's Journal,* I discussed optimizations possible in solving the generalized *n*-queens problem (positioning *n* queens on an *n*-by-*n* board such that none of them are under attack), giving a more accessible account of the problem that I discussed in a 1995 SCCS presentation [2] and also expanding on it (omitting parallel execution through PVM but adding parallel execution through Java threads).

I have received electronic mail from two students at the University of Sannio in Italy asking for an MPI version of the *n*-queens solution presented in PVM in the SCCS paper. That motivated me to develop an MPI implementation, but developed from the 2005 *Dr. Dobb's Journal* discussion rather than the 1995 SCCS discussion.

In the course of that development, I discovered that sometimes one has to teach an old dog *old* tricks: I have just finished my year being referenced in a Beatles' song ("When I'm Sixty-Four"), yet I found myself doing exactly the same wrong-headed thing against which I warn my students — sitting down and writing the whole huge application before starting to test and debug it.

## 2. REVIEW OF THE *N*-QUEENS SOLUTION

Each solution to the *n*-queens problem can be represented by a vector giving the position of the queen in each row, since horizontal movement by the queens guarantees that there can only be one to a row. Further, because vertical movement guarantees that there is only one queen in each column, that solution vector will be a permutation vector. Consequently one can expand on the recursive generation of permutations [3].

```
void permute(int x[], int idx, int n)
{  int k;
   if (idx == n-1) // Complete permutation
      process(x, n);
   else
   {  int hold;
      for (k = idx; k < n; k++)
      {  swap (x, idx, k);
         if (feasible(x, idx))
            permute(x, idx+1, n);
      }
      // Above right-shifts by one
      hold = x[idx]; // Belongs at end.
      for (k = idx+1; k < n; k++)
         x[k-1] = x[k];
      x[n-1] = hold;
   }
}
```

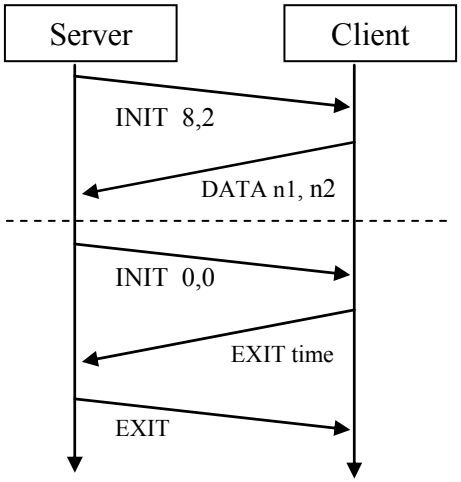
For the queens' problem, the feasibility check is for attack from a lower-indexed queen on the diagonal (where row–column is constant) or on the antidiagonal (where row+column is constant). Niklaus Wirth has shown that these checks can be done in constant time if there are two Boolean vectors of size 2*n*–1 flagging whether a particular diagonal or antidiagonal is occupied [4]. Indeed, since the recursive step is taken only when a partial permutation is feasible, only the newly added row needs to be verified against the lower-indexed queens.

## 3. MESSAGE-PASSING STRUCTURE

The LAM/MPI implementation of the Message Passing Interface is being used, but in a fashion that would be consistent with the MPICH implementation (that is, developing the program as a single-program/multiple-data implementation). Consequently we need to determine who will be communicating with whom and what the messages will be. Within the MPI communicator, processes are identified by their rank within the communicator. The rank zero process will be the task server, providing work to the compute engine clients. Those clients will have ranks greater than zero. Under the MPI implementation in use the number of processes is determined at start-up. Thus if the program is run specifying five process (–np 5), there will be four compute engine clients and the one task server.

The over-all problem is split into subproblems based on the position of the queen in row zero. Consequently the information required by the compute engine client is simply the size of the board and the position of the queen. (This same message can constitute a termination message — specifying a size-zero board.) The client can then generate the initial *n*-entry permutation vector with the appropriate number at the front and process it beginning with row one.

The version of Nqueens being used isolates both the number of unique solutions (retaining only one case of configurations that are equivalent under the symmetry operations of rotation and reflection) and the number of total solutions. When the client has completed the computation, it needs to pass this information back to the task server (which accumulates the total) and receive back another subproblem to process.




Once all of the computations are completed, the program will accumulate information about the total processing time used by the compute engine clients, so these numbers need to be transmitted to the server. The final hand-shaking will be the server sending messages to the clients for them to cease executing. The following figure shows the communications between the server and a specimen client. In addition, it shows the message tags associated with those messages (INIT from server to client, DATA from client to server, and EXIT flowing in both directions).

## 4. INCREMENTAL DEVELOPMENT: PART 1

The first step is to isolate the message passing from the problem solution. The initial program simply sends the messages.

The server program determines the size of the board, and then sends problem specifications to the available clients for queens in positions 0 through (size+1)/2 — this latter value because anything found after that will be a (vertical) mirror image of a solution already processed. There are three phases to this: (1) sending the first board specifications to the clients; (2) while there are column positions yet to be explored, receiving solutions from clients and sending them new specifications; and (3) receiving the final solutions from clients and sending them the size-zero termination message. The final transaction is receiving the processing time from all of the clients (with tag EXIT) and then finishing the hand shaking by sending to the client a message with tag EXIT for them to terminate.

The client programs do all the work. The initial stub version, however, can just receive problem specifications and return dummy values in the DATA message back.

Under the MPI implementation used (LAM/MPI) [5], output from `stdout` all goes to the rank-0 computer, which acts as the task server. Consequently the message traffic can easily be seen by simply inserting tracing `printf` messages to report points in the communication.

Code is available on the web [6]. The communications verification is in the program `msg_tst.c`. The required timing functions are in `cpuTimes.c`.

## 5. INCREMENTAL DEVELOPMENT: PART 2

The solution sketched requires separating the row zero positioning from the solution of the problem beginning with row one. This can be modeled in a purely sequential program, one in which one procedure deals out problems to another procedure that starts them, calling the backtracking procedure for row one.

```
// x is the data type:  swap items y and z.
#define swap(x, y, z) {x tmp=y;y=z;z=tmp;}

void Start(int k, int Size)
{  static int *Board = NULL, *Trial = NULL;
   if (Board == NULL)

// [Dynamic memory allocation
// details omitted]

   swap (int, Board[0], Board[k]);
// CRITICAL:  mark [0] as in-use
   Mark (0, Board[0], Size, Diag, AntiD, 1);
```

```
   Nqueens (Board, Trial, Size, 1);
   Mark (0, Board[0], Size, Diag, AntiD, 0);
   swap (int, Board[0], Board[k]);
}

void Deal (int Size)
{  int k, lim = (Size+1) / 2;
   for ( k = 0; k < lim; k++ )
      Start(k, Size);
}
```

Code is available on the web [6]. The splitting verification is in the program `split.c`.

## 6. PARALLEL PROGRAM CONSTRUCTION

The final parallel program is constructed simply by migrating the logic from the `Start` procedure from `split.c` into its proper location in `msg_tst.c`. Since each piece was verified separately, the resulting program works — provided the logic from `Start` is correctly moved into the compute engine client program. Code is available on the web [6]. The final program is in the file `MpiQueen.c`.

## 7. SPECIMEN EXECUTION RESULTS

The final program was run on a cluster of five unloaded quad-processor Xeon computers. Due to hyperthreading, these appear to Linux to have eight 1.5 GHz processors. LAM/MPI was started in a fashion that started processes such that the task server ran by itself on a computer, with "cpu=1" in the MPI configuration file. The compute engine servers ran three to a computer in the distribution, based on "cpu=3". For each pair of tables, the first table shows the clock time required and the speed-up ratios; the second table shows the processing time required on the individual compute engines.

| No. of Queens | No. of Compute Engines | MPI Clock Time | Speed-Up: Seq.time / Parallel time |
|---|---|---|---|
| 14 | 1 | 1.42 | 1.01 |
| 14 | 2 | 0.80 | 1.79 |
| 14 | 3 | 0.60 | 2.38 |
| 14 | 4 | 0.42 | 3.40 |
| 14 | 5 | 0.41 | 3.49 |
| 14 | 6 | 0.38 | 3.76 |
| 14 | 7 | 0.29 | 4.93 |
| 1.43 | Time for sequential program | | |

| Individual Compute Engine Processing Times | | | | | | |
|---|---|---|---|---|---|---|
| 1.41 | | | | | | |
| 0.80 | 0.61 | | | | | |
| 0.59 | 0.40 | 0.43 | | | | |
| 0.38 | 0.40 | 0.42 | 0.21 | | | |
| 0.39 | 0.40 | 0.21 | 0.21 | 0.21 | | |
| 0.38 | 0.19 | 0.21 | 0.21 | 0.21 | 0.21 | |
| 0.17 | 0.28 | 0.20 | 0.21 | 0.21 | 0.21 | 0.29 |

One can clearly see one consequence of parallel processing: the over-all time required is determined by the *slowest* participating process.

Because of the mirror symmetry correction one needs only compute the left seven columns of the fourteen. Because of the coarse granularity of the problem, sometimes one or more processes compute more instances than other processes. For instance, for cases with 4, 5, or 6 compute engines, one or more has to compute two configurations, so that they have about the same speed-up ratios.

For boards of size 15 and size 16, one needs to compute the left eight columns.

| No. of Queens | No. of Compute Engines | MPI Clock Time | Speed-Up: Seq.time / Parallel time |
|---|---|---|---|
| 15 | 1 | 9.59 | 1.01 |
| 15 | 2 | 4.89 | 1.98 |
| 15 | 3 | 3.63 | 2.66 |
| 15 | 4 | 2.50 | 3.86 |
| 15 | 5 | 2.48 | 3.90 |
| 15 | 6 | 2.48 | 3.90 |
| 15 | 7 | 2.27 | 4.26 |
| 15 | 8 | 1.77 | 5.46 |
| 9.66 | Time for sequential program | | |

| Individual Compute Engine Processing Times | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9.58 | | | | | | | |
| 4.72 | 4.89 | | | | | | |
| 3.50 | 3.63 | 2.47 | | | | | |
| 2.34 | 2.38 | 2.48 | 2.49 | | | | |
| 2.28 | 2.39 | 2.48 | 1.24 | 1.26 | | | |
| 1.41 | 2.39 | 2.48 | 1.24 | 1.66 | 1.26 | | |
| 2.27 | 1.13 | 1.75 | 1.23 | 1.25 | 1.26 | 1.80 | |
| 1.01 | 1.60 | 1.22 | 1.76 | 1.25 | 1.73 | 1.25 | 1.76 |

| No. of Queens | No. of Compute Engines | MPI Clock Time | Speed-Up: Seq.time / Parallel time |
|---|---|---|---|
| 16 | 1 | 60.40 | 1.01 |
| 16 | 2 | 30.68 | 1.99 |
| 16 | 3 | 22.91 | 2.67 |
| 16 | 4 | 15.77 | 3.88 |
| 16 | 5 | 16.15 | 3.78 |
| 16 | 6 | 15.79 | 3.87 |
| 16 | 7 | 15.13 | 4.04 |
| 16 | 8 | 11.17 | 5.47 |
| 61.12 | Time for sequential program | | |

| Individual Compute Engine Processing Times | | | | | | | |
|---|---|---|---|---|---|---|---|
| 60.4 | | | | | | | |
| 29.7 | 30.7 | | | | | | |
| 22.0 | 22.9 | 15.6 | | | | | |
| 14.1 | 14.9 | 15.6 | 15.8 | | | | |
| 8.8 | 15.0 | 16.1 | 16.2 | 10.5 | | | |
| 8.8 | 9.8 | 15.8 | 15.8 | 10.4 | 10.8 | | |
| 8.8 | 15.1 | 10.8 | 7.8 | 10.5 | 8.0 | 11.2 | |
| 8.9 | ***7.0*** | 10.8 | 11.0 | 10.5 | 8.0 | ***11.2*** | 11.1 |

The effect of the granularity is quite obvious above: for the 16x16 board, positioning the queen in column 1 leads to lots of decision-tree pruning, while positioning in column 6 has significantly less pruning — a time ratio of 1.6. To counterbalance that, the task server process could itself position queens in row 1 as well as row 0, distributing the tasks from row 2 to the compute engine clients.

## 8. SUMMARY

Incremental development works very well even in developing parallel programs. In such a circumstance, develop the communications of the participating processes separately from the program modification to allow distribution of subproblems.

## 9. WEB RESOURCE

http://penguin.ewu.edu/~trolfe/MpiQueen/index.html. This page provides access to this paper, and also to the programs discussed above and an Excel workbook giving the results of the numerical experiment that gave the above tables.

## 10. ACKNOWLEDGEMENTS

These results were obtained using equipment within the Computer Science Department at Eastern Washington University.

## REFERENCES

[1] Timothy Rolfe, "Optimal Queens", *Dr. Dobb's Journal*, Vol. 30, No. 5 (May 2005), pp. 32-37. http://www.ddj.com /dept/architect/184406068 provides access to the text of the article, and ftp://66.77.27.238/sourcecode/ddj/2005/0505.zip provides access to the source code.

[2] Timothy Rolfe, "Queens on a Chessboard: Making the Best of a Bad Situation", presented in the Technical Paper Sessions at the Small College Computing Symposium at Augustana College (Sioux Falls, SD) 21-22 Apr 1995. Published in *SCCS: Proceedings of the 28th Annual Small College Computing Symposium (1995)*, pp. 201-10. The paper and associated programs are available through http://penguin.ewu.edu/~trolfe/SCCS-95/index.html

[3] Timothy Rolfe, "Backtracking Algorithms", *Dr. Dobb's Journal*, Vol. 29, No. 5 (May 2004), pp. 48, 50-51. Text of the article is available through http://www.ddj.com/dept/architect/184405652. ftp://66.77.27.238/sourcecode/ddj/ 2004/0405.zip provides access to the source code.

[4] Niklaus Wirth, "Program development by stepwise refinement", *Communications of the ACM*, Vol. 14, No. 4 (April 1971), pp. 221-227. It also appears in his *Algorithms and Data Structures* (Prentice-Hall, 1986), pp. 153-157.

[5] Information on MPI itself is available through http://www-unix.mcs.anl.gov/mpi/. Information on LAM/MPI is available through www.lam-mpi.org.

[6] http://penguin.ewu.edu/~trolfe/MpiQueen/