UMEÅ UNIVERSITY
Dept. of Computing Science
Obligatory assignment

The N-Queens Problem

John-John Markstedt
c14jmt@cs.umu.se
April 4, 2018

CONTENTS

I. Problem

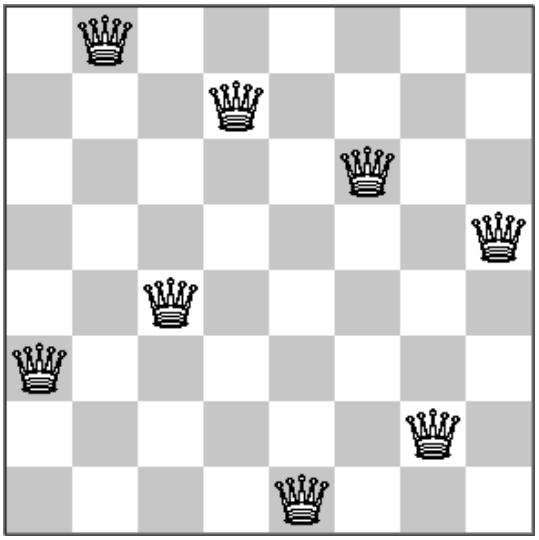

Figure 1: *a solution for eight queens[1]*

The n-queen problem is the problem of placing n queens on an n x n chessboard such that no two queens are threatening each other. It's the more general variant of the eight queens problem which is, of course, the same problem but with eight queens on an 8x8 chessboard. Only n=2 and n=3 have no solution and the solutions grow quickly with n. The full sequence calculated to date is available in the On-Line Encyclopedia of Integer Sequences and the currently the last known answer is for n=27 which have 29,363,495,934,315,694 solutions[2].

II. Sequential algorithm

Since each solution have similar sibling solutions, one could rotate or reflect a solution to find another, a lot of queen placements are therefor not necessary to verify to find all solutions solution[3]. However the fundamental algorithm is the basically the same if rotation and reflection is not used.

At the heart of n-queens problem is a short recursive beautiful function. It requires a global integer to be incremented upon finding a new solution, this could also be implemented by passing the solutions along and recursively adding them together on the merging returns.

```
bool recursiveQueenPlacement
   (int board[N][N], int col)
{
    // Counts each solution
    if (col >= N)
    {
      numSol++;
      return true;
    }
    //Place queen for each row in column
    for (int i = 0; i < N; i++)
    {
        // if legit placement
        if ( isSafe(board, i, col) )
        {
            //place queen
            board[i][col] = 1;

            // recur to place queen next column
            recursiveQueenPlacement(board, col+1)

            board[i][col] = 0;
        }
    }
    return false;
}
```

Figure 2: *The recursive algorithm at the heart of n-queens*

The algorithm uses a two dimensional board with zeros representing empty squares and ones representing queens. To initiate a board full of zeros may be passed into the function along with the first column. The algorithm places a queen in each square on that column and branches of, calling the same function again with the updated board and the next column. The algorithm forms a tree continuing down each branch until; either 1) a queen cannot be placed due to threat from another already placed queen(see fig 3), this cores is commonly referred to as pruning; or 2) the column reaches the end of the column. This algorithm is a backtracking algorithm, since it tries a combination until it threatened or done and then backtracks to previous column and tries the next square.

To check if a placement is valid, a help function is used (see isSafe in fig 3). It takes the position where to place the queen, and checks the squares on the same row and the two diagonals, if no queen is found the square is safe.

```
bool isSafe
(int board[N][N], int row, int col)
{
    int i, j;
    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    /*Check upper diagonal on left side*/
    for (i=row, j=col;i>=0&&j>=0;i--,j--)
        if (board[i][j])
            return false;

    /*Check lower diagonal on left side*/
    for (i=row, j=col;j>=0&&i<N;i++,j--)
        if (board[i][j])
            return false;

    return true;
}
```

Figure 3: *Function to validate if a placement is valid or not.*

In this particular implementation only half of the board placements is checked, on the first column the queens are placed up the square n/2. Since each solution flipped over is also a solution. This finds exactly half the solutions, so the solution total can be doubled. If n is odd the solutions with the queen on the middle row in the leftmost column are only counted ones[4].

III. Parallel algorithm

The sequential algorithm can easily be used in parallel. Since the algorithm takes in a previous board, which can essentially be a branch, one could pre-calculate branches and then divide the branches among the cores. The cores could then in parallel find the solution for it's branches and then reduce the solutions back to one core.

The branches however, have tremendous deviations in run time. Some branches quickly gets pruned off while others run for relatively long time. This creates a need for some sort of shared work pool and raises a question of how long that pool should be. The longer the work pool the more overhead but also less deviation in idling between cores in the end.

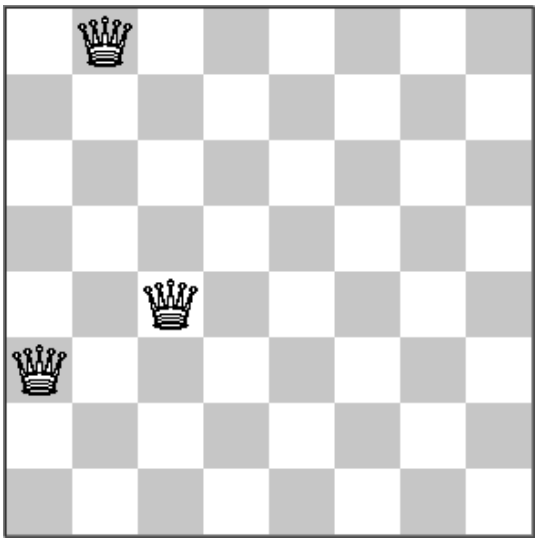

Figure 4: *A pre-calculated branch*

A. Load balancing

The question then becomes how many branches is sufficient? and how should it be generalized to work with different cores and different board sizes?

It's obvious that optimal formula, it there even is one, is dependent on both board size and cores. However to make it easy this implementation only takes cores into account, getting this formula:

$$cores * w <= ((boardSize/2) + (boardSize * (i-1)))$$

Where w is the amount of problem proportional to cores and the smallest i to satisfy the equation is subsequently 'sufficient' depth to pre-calculate to get at least w problems per core. In this implementation w is set to five.

If one wish to optimize the problem amount(pa) for a specific board size and core amount one could easily go about it be measuring overhead in time as a function of pa, O(pa) and measuring the largest deviation between cores finishing time as a function of pa, D(pa). A equation to find the with following equation.

$$T(pa) = O(pa) + D(pa)$$

$$T'(pa) \approx 0$$

Note that the functions are discrete and only defined for integers.

B. MPI Communication

MPI does not have shared memory and the cores need to synchronize somehow so a core is allocated to dispense work. Pre-calculations is done before the core split(MPI_INIT) and each core has the full queue of branches. Communication is initiated by the working cores via a MPI_SEND to the zero core. The zero core receives and respond to the same core with an integer corresponding to the index in the work queue, the zero core increment the responding index and waits for next working core to ask for work.

When the queue is empty the result are gathered with a MPI_REDUCE(fig 6) with the summation function. The total solution reach the zero core and the final solution is discovered.

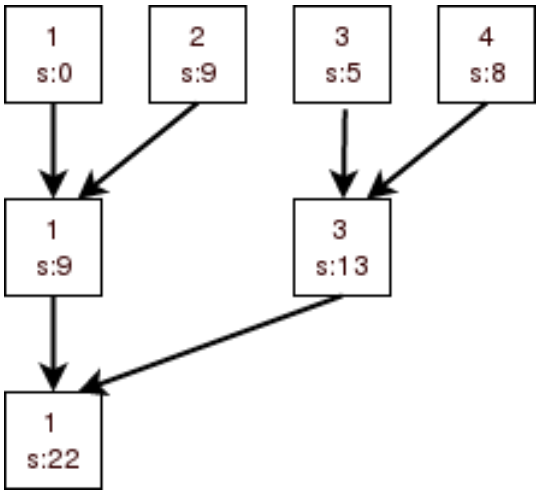

Figure 5: *An illustration of MPI_REDUCE*

IV. Experimental setup

The experiment is performed on the supercomputer Kebnekaise at HPC2N. It ran on a single node with a base frequency of 2.6GHz[5]. Further specifications can be found on the HPC2N website[5]. The problem is inherently branch heavy so GPU's cannot efficiently be used.

The experiment is partitioned into two parts. Firstly the program is executed with a fixed board size and increasing cores and secondly the program is executed with a increased board size and fixed core amount. The sequential algorithm is also executed for each board size so speedup and efficiency metrics can be calculated.

Each board size and cores pair are executed five times so deviations can be avoided by using the median.

V. Results

A. 16x16 Chessboard, increasing cores.

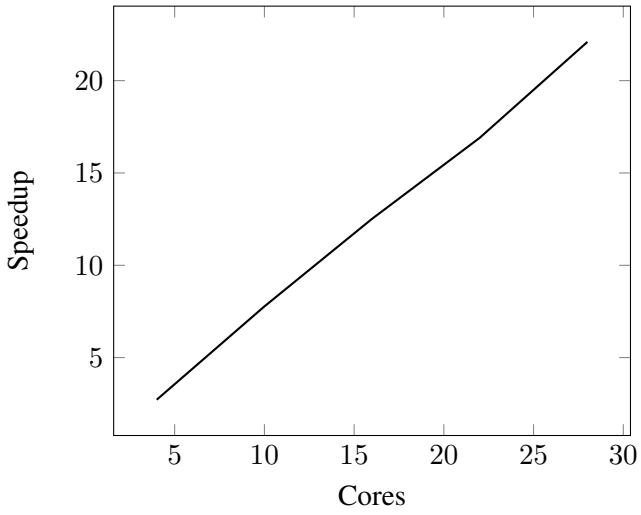

Figure 6: Speedup fixed problem size.

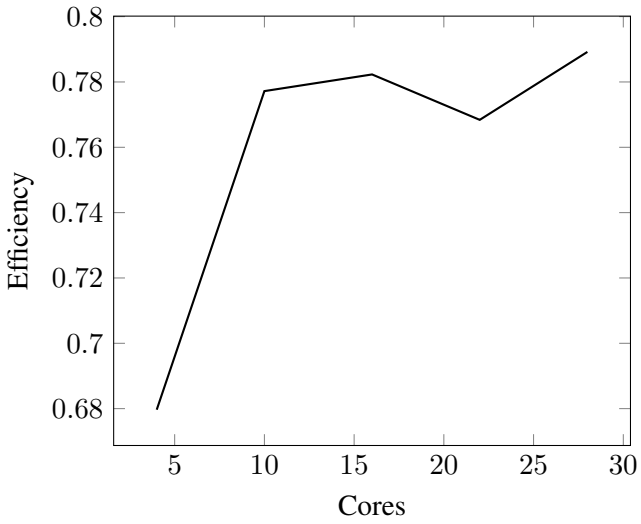

Figure 7: Efficiency fixed problem size.

B. 24 cores, increasing chessboard size

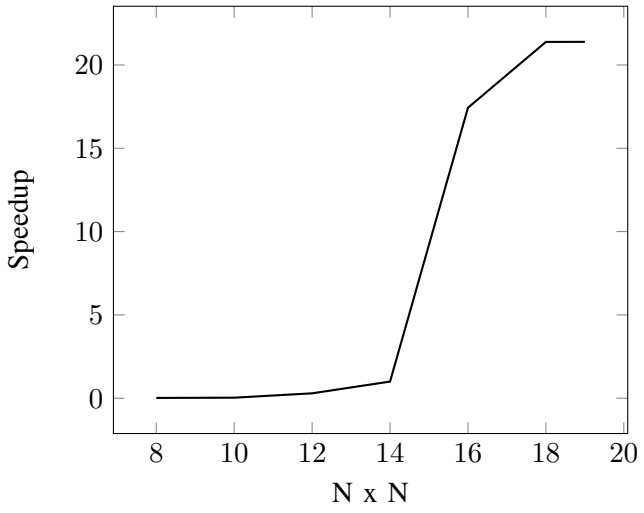

Figure 8: Speedup fixed cores.

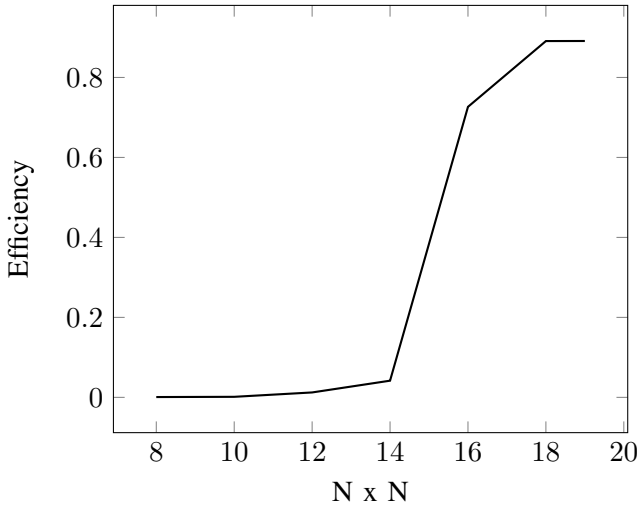

Figure 9: Efficiency fixed cores.

VI. Summary and Conclusions

There is a few obvious points that can be pointed out. Firstly since one core is used solely to delegate work it will affect the efficiency significantly for small amounts of cores. Secondly the sequential algorithms for n < 15 is so fast(6ms) in relation to initiation of the parallel environment such that speedup and efficiency for those n is non existent.

For larger n, the efficiency continues to increase as impact of the cost of core zero continues to decrease in relation. The variation between cores(as seen by the dent p = 20 in fig 7), that the efficiency/speedup is not a straight or curved line is most likely caused by the calculation of problem size(as seen in section III).

All in all the problem is quite scalable for n > 15 where the otherwise trivial setup time become significant. The efficiency holds until 28 cores and the tests are insufficient to say much more.

A. Improvements

There are two obvious factors that can be improved in this solution.
1) Using Rotation and Reflection to eliminate branches from the get go.
2) Implementing the board with bits instead of integers, making the isSafe and reallocation of the board on each recursion much faster.

REFERENCES

[1] https://letstalkdata.com/2013/12/n-queens-part-1-steepest-hill-climbing/

[2] The On-Line Encyclopedia of integer sequences
N.J.A Sloane
1973 with later additions
https://oeis.org/A002562

[3] Wiesstein Eric. W.
Queens Problem
MathWorld - A Wolfram Web Resource
http://mathworld.wolfram.com/QueensProblem.html

[4] Sequential n-queens
Source code
https://github.com/Johnstedt/N_Queens_Puzzle/blob/master/n-queens.c

[5] Kebnekaise HPC2N, Umeå University
https://www.hpc2n.umu.se/resources/hardware/kebnekaise

VII. APPENDIX

board	cores	time(s)
8	1	0.039515
8	1	0.005887
8	1	0.005820
8	1	0.006063
8	1	0.006524
10	1	0.012205
10	1	0.010780
10	1	0.010518
10	1	0.011487
10	1	0.011324
12	1	0.126832
12	1	0.137974
12	1	0.131725
12	1	0.143180
12	1	0.137553
14	1	2.701961
14	1	2.705915
14	1	2.706824
14	1	2.714577
14	1	2.726048
16	1	116.905184
16	1	116.858324
16	1	116.800595
16	1	117.288180
16	1	116.916415
16	1	117.318764
16	1	116.892444
16	1	117.249263
16	1	117.229931
16	1	117.211958
18	1	7484.554829
18	1	7592.921977
18	1	7487.480657
18	1	7466.366401
18	1	7465.868947
19	1	65324.593308
19	1	65326.335311
19	1	64483.570383
19	1	64467.921925
19	1	66092.728741

Figure 9: Showing run for sequential data

board size	cores	median	mean	std deviation
8	1	0.006063	0.012762	0.013379
10	1	0.011324	0.011263	0.000588
12	1	0.137553	0.135453	0.005634
14	1	2.706824	2.711065	0.008535
16	1	116.905184	116.953740	0.172139
18	1	7484.5548	-	-
19	1	65324.593	-	-

Figure 9: Showing run for sequential data aggregated

board	cores	time(s)
8	24	0.338399
8	24	0.337922
8	24	0.323769
8	24	0.353904
8	24	0.315210
10	24	0.377513
10	24	0.378943
10	24	0.299654
10	24	0.322961
10	24	0.355944
12	24	0.340829
12	24	0.326173
12	24	0.356659
12	24	0.350382
12	24	0.316492
14	24	0.501949
14	24	0.467634
14	24	0.512713
14	24	0.466575
14	24	0.458439
16	24	6.726658
16	24	6.684882
16	24	6.705324
16	24	6.757147
16	24	6.661344
18	24	350.172703
18	24	349.746206
18	24	349.789673
18	24	350.098679
18	24	350.016534
19	24	3055.663145
19	24	3057.710257
19	24	3054.283825
19	24	3054.117081
19	24	3052.825643

Figure 9: Showing run for static cores and increased problem size

board	cores	time(s)
16	4	42.991166
16	4	42.999632
16	4	43.001902
16	4	42.991309
16	4	43.026596
16	10	15.058815
16	10	14.985173
16	10	15.066871
16	10	15.040822
16	10	14.990488
16	16	9.369219
16	16	9.339596
16	16	9.248873
16	16	9.226057
16	16	9.345293
16	22	6.953044
16	22	6.902602
16	22	6.910734
16	22	6.979525
16	22	6.914859
16	28	5.290084
16	28	5.270778
16	28	5.325644
16	28	5.334707
16	28	5.253322

Figure 9: Showing run for increased cores and static problem size

board size	cores	median
8	24	0.337922
10	24	0.355944
12	24	0.467634
14	24	2.706824
16	24	6.705324
18	24	350.0166
19	24	3054.284

Figure 9: Showing run for static cores and increasedproblem size aggregated data

board size	cores	median
16	4	42.999632
16	10	15.040822
16	16	9.339596
16	22	6.914859
16	28	5.290084

Figure 9: Showing run for increased cores and static problem size a