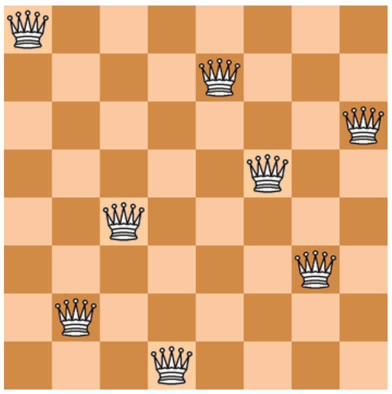
Project Report: N-Queens solver in Parallel as a CSP

I learned about Constraint Satisfaction Programming earlier in the semester, and was very interested to see how it would perform if given more processors in parallel. In particular, I was wondering whether I could achieve a linear, sub-linear, or super-linear speedup. And at what point adding more processors would not significantly help. I used the popular N-Queens puzzle to benchmark my approach, while attempting to avoid adding unnecessary domain-specific information. Thus, this implementation treats the solution as part of the NP-complete class of problems and ignores some of the reductions that can be done -- including the O(1) algorithmic solution to this puzzle, as well as some of the probabilistic methods that at the cutting edge of current research approaches. I found that by distributing the search effort across multiple processors, I could achieve a speed up to the first discovered-solution time of a couple orders of magnitude in some cases.

The N-Queens puzzle is an extension of the 8-Queens puzzle proposed by Max Bezzel in 1848. In the 8-Queens puzzle, the question is how to place 8 queens on a standard 8x8 chess board such that no two queens are currently in a position to attack another queen, given the standard chess rules allowing horizontal, vertical, or diagonal movement by any number of squares. The N-Queens puzzle generalizes that question to that of placing N queens on an NxN chessboard such that none of them can attack another queen. The explicit algorithmic solution involves placing the queens in a specific stair-stepping pattern. This is a pattern typically seen in solutions found by an exhaustive solver. For the 8-Queens problem, there are 92 distinct solution or, if symmetrical solutions are counted together, 12 unique solutions. The first solution found using the backtracking method is shown pictorially below (image courtesy of Wikipedia):



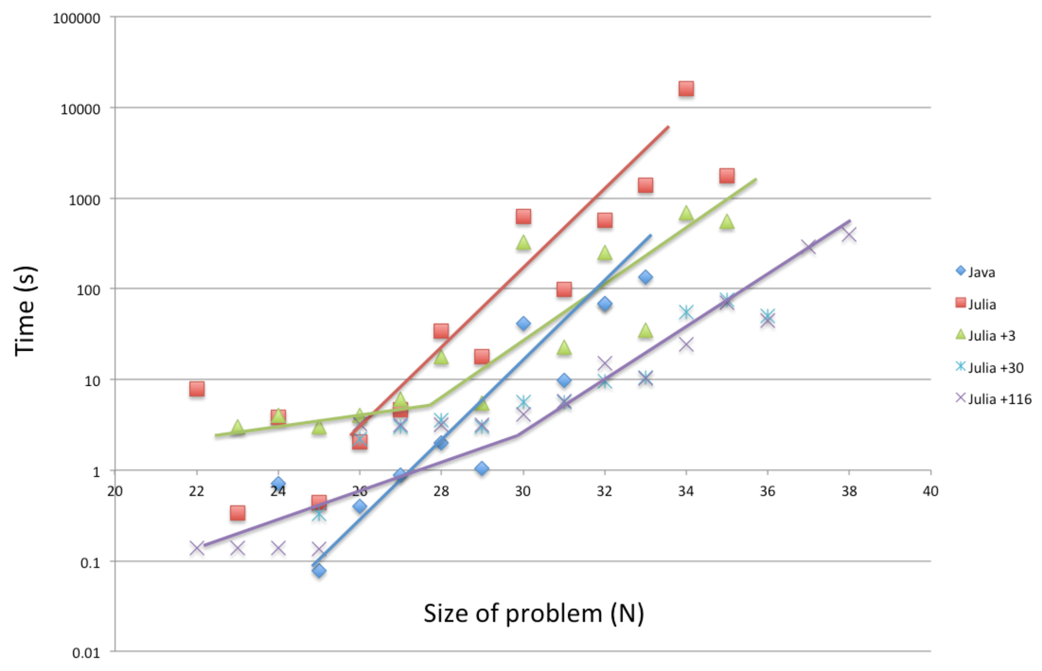
The primary brute-force method of finding a single solution involves a backtracking search. This is essentially a search of a depth-first tree. We use the simple fact that only one queen can be place on a row to limit the domain to that of finding a solution that places 1 queen in any column in each of the N rows such that no queens are attacking. For every assignment at row n, we try every assignment at row n+1. We keep placing queens at row n+1 until either n=N and a solution is found – so we are done – or there is no possible assignment at that row n that does not conflict with a queen placed previously – so we backtrack and try the next assignment at row n. While this is exhaustive, it is also inefficient, exhibiting roughly O(N!) growth. This means that while N=28 may be quickly solvable, N=32 can take a while, and N=34 is severely impractical. One possible optimization that I looked at was checking for possible assignment consistency ahead of time instead of when placing the queen. This method is called backtracking with forward checking and it can be used limit the number of times an invalid position is looked at, as the algorithm recursively searches up and down the tree, at the cost of creating and maintaining this list.

I felt this problem lent itself to the fairly obvious form of parallelism by breaking up the entire problem tree and searching solutions on separate processors. However, I also recognized that there are a fair number of solutions and symmetrical placements. Therefore, simply assigning each processor a branch of the search tree on the same level would be more likely to find the same solution many times than to find a single unique solution more quickly. Thus I developed a simple algorithm that divided the search space unevenly and quasi-randomly across all processors. However, to ensure that all processors would eventually do about the same amount of work, I only distribute a small portion of the problem at a time and then select a new chunk to send to the next processor until some processor finds a solution in its assigned search block. To keep the scheduler simple, I left all consistency checking up to the processor assigned the work. To avoid giving one processor too small of a workload relative to the other processors, I gave each processor linearly more nodes at its base level as I moved down the search tree. So for 4 processors, I would assign, at each increasing level, 1 then 4 then 8 then 16 (i.e. all) branches to search. Below you can see the effect of this algorithm on the order in which the processors returned from searching. The format gives the processor id number then the search domain given to the processor. For this size problem, some of the ordering was due to communication delays rather than actual work differences, and three processes found the solution at the end before the information that a solution was already found was propagated.

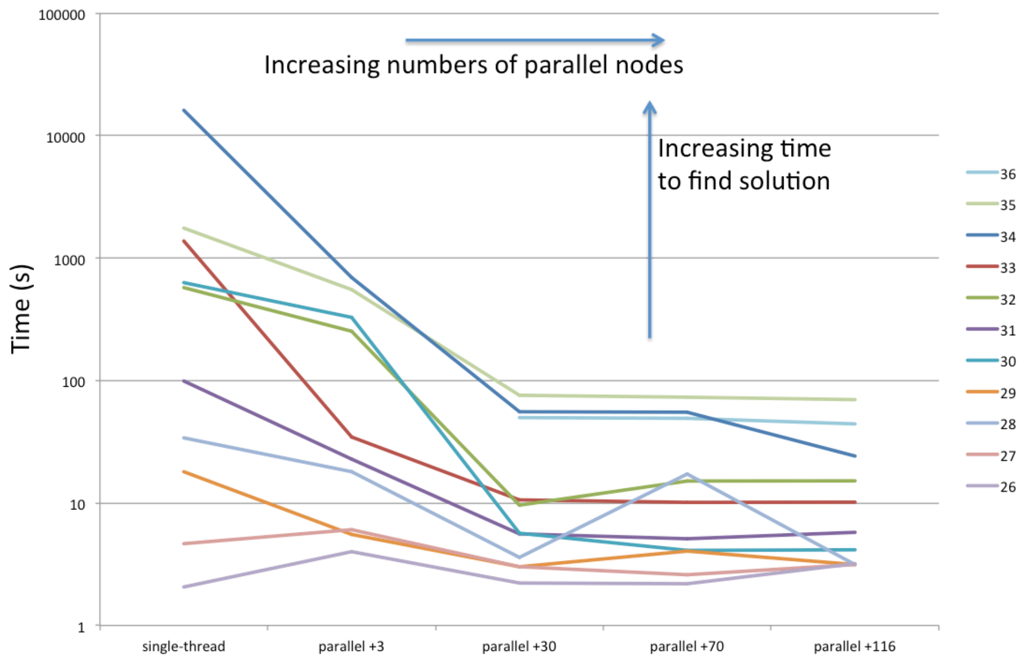
```
julia> searchSolutionParallel(N:=16, forwardChecking:=false)
3: Domains(2 : 1..2 : 1..16 : ...)
4: Domains(2 : 3 : 1..4 : 1..16 : ...)
```

```
        5: Domains(2 : 3 : 5 : 1..8 : 1..16 : …)
        3: Domains(2 : 3 : 5 : 9 : 1..16 : …)
        4: Domains(2 : 3 : 5 : 10..16 : 1..16 : …)
        5: Domains(2 : 3 : 6..9 : 1..16 : …)
        3: Domains(2 : 3 : 10 : 1..8 : 1..16 : …)
        4: Domains(2 : 3 : 10 : 9 : 1..16 : …)
        3: Domains(2 : 3 : 11..14 : 1..16 : …)
        4: Domains(2 : 3 : 15 : 1..8 : 1..16 : …)
        3: Domains(2 : 3 : 15 : 9 : 1..16 : …)
        4: Domains(2 : 3 : 15 : 10..16 : 1..16 : …)
        3: Domains(2 : 3 : 16 : 1..16 : …)
        2: Domains(1 : 1..16 : …) found solution
        3: Domains(2 : 6 : 1..4 : 1..16 : …) found solution
        5: Domains(2 : 3 : 10 : 10..16 : 1..16 : …)
        4: Domains(2 : 4..5 : 1..16 : …) found solution
```

The results exceeded my expectations, with a super-linear speed up for sufficiently large problems, and sometime over an order of magnitude drop in search time. The effect was much more pronounced for increasing values of N, since slightly larger problems went from essentially unreachable to rapidly solvable with only a modest number of additional processors. I believe that setup and communications time are dominating for N<28, inhibiting noticeable results at those problem sizes. The parallelism of the backtracking method was able to take advantage of the multiple processors searching in parallel to achieve a speed up on the overall problem. The following plots provide two views on the same data showing the improvements achieved as more processors are added. In the first graph, the fit lines are drawn by hand to guide the eye in clustering related data points. It is useful to note that the parallelism appears to have decreased the slope of the log-linear line, implying that the speed-up will be increasingly large for higher values of N.



In the second graph, we observe the decreasing benefit from adding more processors, for small values of N.



Adding more processors appeared to have little effect on the speed of solution, but did affect the largest problem that could be solved in a reasonable amount of time, as can be seen in the following tabular form of the data (times given in seconds):

	single-thread	parallel +3	parallel +30	parallel +70	parallel +116
N=38	infeasible	infeasible	infeasible	infeasible	401
37	infeasible	infeasible	infeasible	infeasible	292
36	infeasible	infeasible	50	49	44
35	1758	553	76	73	70
34	16099	694	56	55	24
33	1378	35	11	10	10
32	574	253	10	15	15
31	99	23	6	5	6
30	631	328	6	4	4

I suspect the limiting element is the efficiency with which I am dividing up the work, but if I could know this in advance, I would presumably also have found a way of making p=np!

As I wanted to make my implementation of forward-checking integrate seamlessly with the backtracking and domain assignment, it does not store the history information as efficiently as possible. The additional cost of performing the forward-checking and maintaining the domain history appears to outweigh the benefits of performing the early domain pruning. This is true when executed serially on a single processor and was also true when searching in parallel. Part of the additional disadvantage for the parallel case could be the fact that I didn't prune the domains on the head node. Thus all child nodes had to perform the same initial work, some of which immediately invalidate the entire search domain, requiring additional communication, which could have been avoided if the master node had done more work initially. However, I had trouble handling the algorithmic complexity of ensuring that all nodes are eventually searched combined with the requirement of eliminating all future combinations that are not valid. However, I also decided that the loss would likely be slight – given that each processor could be performing this additional elimination step in parallel instead of upfront in serial.

Using a simple approach to dividing the problem into an absurdly parallel search problem yielded sizeable benefits for reducing search time and increasing the maximum locatable solution size. While we did not overcome the exponential growth in time complexity, it does appear that the solution difficulty was greatly reduced. Unfortunately, however, this does means it is still is far from being competitive with the more advanced probabilistic solvers that are currently on the cutting edge of research in this area. For the exhaustive approach for guaranteeing a solution, the implemented parallelism resulted in some large reductions in search time, without requiring the addition of domain-specific information, such as the explicit solution or even that the domain of each row is shared (since only one queen per column).

The code for this project can be obtained at:

https://bitbucket.org/vtjnash/jqueens/downloads