N-Queens

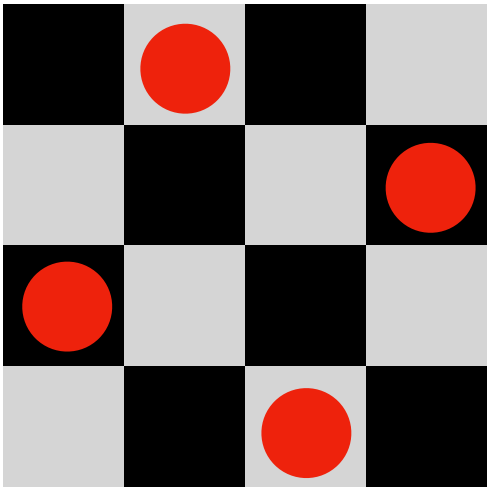
Leveraging Parallel Functional Programming to calculate the number of solutions the classic N-Queens Problem

Alexandra Holguin
anh2150
COMS 4995, Fall 2021

Introduction

Inspired by the game of Chess, N-Queens is a classic problem in computer science. Given a natural number N, the problem is to find the number of ways N queens can be arranged on an NxN chessboard such that no queen threatens another. In other words, there must be one and only one queen in every row, column and diagonal of a square chess board of a given size.

For values of N greater than 10, both the amount of calculation and the number of solutions grows at an exponential rate, making this a very good problem that can be parallelized.



Example of a solution for N = 4

Approach

A naive approach of checking every possible configuration of N queens on an NxN board would be a colossal task. Consider N = 8. Checking every possible configuration would mean checking 178,462,987,637,760 configurations.

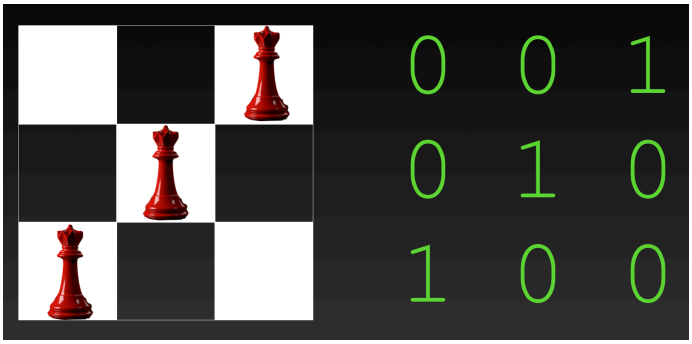
So, we can use some heuristics and intuitive rules of thumb to cut down on the total amount of work needed to be done. If we start with all the queens arranged on a major diagonal of the board, we have already ensured that there is only a single queen per row and column. Now we can generate permutations of the rows to generate all such configurations. For N = 8, this cuts down the total number of permutations to consider to just 8!, or 40,320. And all that needs be done is to check for diagonal conflicts in the generated permutations.

Data Modeling

The most naive approach would be to use a list of lists of ints and directly map each block on a chess board to an Int in a two-dimensional data structure.

However, given the large number of permutations involved, using the least amount of RAM possible becomes important. So a much better approach is to encode the configuration of each row as a number and use a list of integers instead.

In my solution, I used a Word32 to encode the configuration of each row. This Word32 is comprised exclusively of 0-bits, except the specific bit that represents where the queen is present. For example, if the queen is in the last column, the value 0000...0001 is used to represent it. If the queen is in the second-to-last column, the value 000... 0010 is used to represent it and so on.



As a result, a board configuration is represented as [Word32]. An Array instead of a list would likely save me some additional RAM usage, but the laziness of Haskell lists, helps keeps the total working memory in check.

An alternate approach would be to use the integer value of the column that the queen is in. Doing this would also change the operations required to check for diagonals, which we will talk about later.

Preparation

Before the actual computation can begin, I generate a list of all possible configurations that need to checked for diagonal conflicts. This part was very simple by using the standard library function `permutations`.

```
diagonalOfQueens :: Int -> [Word32]
diagonalOfQueens n = [shiftL 1 i | i <- [0 .. (n -1)]]

-- generating all board configurations
diagonal = diagonalOfQueens n
boards = permutations diagonal
```

Checking For Diagonals

To check if there is a diagonal conflict, I take the first row of the board and shift all the bits left by one and compare it against the next row. A simple bitwise-And is enough to ensure that no two queens are in the same diagonal. This process is repeated for the rest of the rows, by shifting the bits for every step down the board. The same process is also repeated for the rest of the rows of the board. Finally the whole process is also repeated while shifting the rows to the right.

```
hasConflictWithFirstRow :: ShiftFn -> Word32 -> [Word32] -> Int -> Bool
hasConflictWithFirstRow _ _ [] _ = False
hasConflictWithFirstRow shiftFn queen (x : xs) offset =
  ((queen `shiftFn` offset) .&. x) /= 0 || hasConflictWithFirstRow shiftFn
queen xs (offset + 1)


hasAnyDiagonalConflict :: [Word32] -> Bool
hasAnyDiagonalConflict board =
  isDiagonalConflict shiftL board || isDiagonalConflict shiftR board
  where
    isDiagonalConflict _ [] = False
    isDiagonalConflict dir (x : xs) =
      hasConflictWithFirstRow dir x xs 1 || isDiagonalConflict dir xs
```

This whole process is computationally efficient as it relies on fast bit shifting and bitwise AND operations.

If we had used the column index as the number instead, we would need to add or subtract 1 as we went down the rows, and check for integer equality instead. While the difference between the two approaches is minimal, the bitwise operators were marginally faster in my testing.

Parallelization

Naive Initial Approach

Checking a long list of board configurations is an inherently parallelizable task. The first approach was to map over the entire list in parallel. This approach worked for smaller values of N, but quickly broke down due to a large number of spark overflowing. There was far too much GC pressure too and the cost of creating Sparks was as bad or worse than the computational speedup.

Better Approach

Next, I tried to control the number of sparks being created by generating the the list of permutations and chunking it up into 10 roughly equal lists of permutations. From there, I was

able to run the 10 chunks in parallel which was able to utilize the 10 cores available on my CPU.

This approach was quite effective, and was able to get a speedup of around 2.5x. However, the memory usage was still quite high, and after further testing, I realized that it wasn't even able to calculate the solutions for N = 12 or greater as it would use up too much memory and get killed by the OS.

Other Exploration

I looked into reducing the memory usage by not generating the permutations up-front. I was able to generate a permutation by index. This meant that I would be able to simply generate a list of N! elements, and the parallel tasks would be able to generate the relevant permutations in parallel.

```
nthPermutation :: [Word32] -> Int -> [Word32]
nthPermutation [] _ = []
nthPermutation diagonal n =
  let
    len = length diagonal
    (position, subN) = n `divMod` factorial (len - 1)
    first = diagonal !! position
    rest = diagonal \\ [first]
  in
    (diagonal !! position) : nthPermutation rest subN
```

When testing this approach, I saw a very minimal decrease in GC-time, but at a significant cost to MUT time. So, after a few tests, I abandoned this approach.

Final Solution

After iterating on the previous approaches, I landed on a good balance between the previous two approaches.

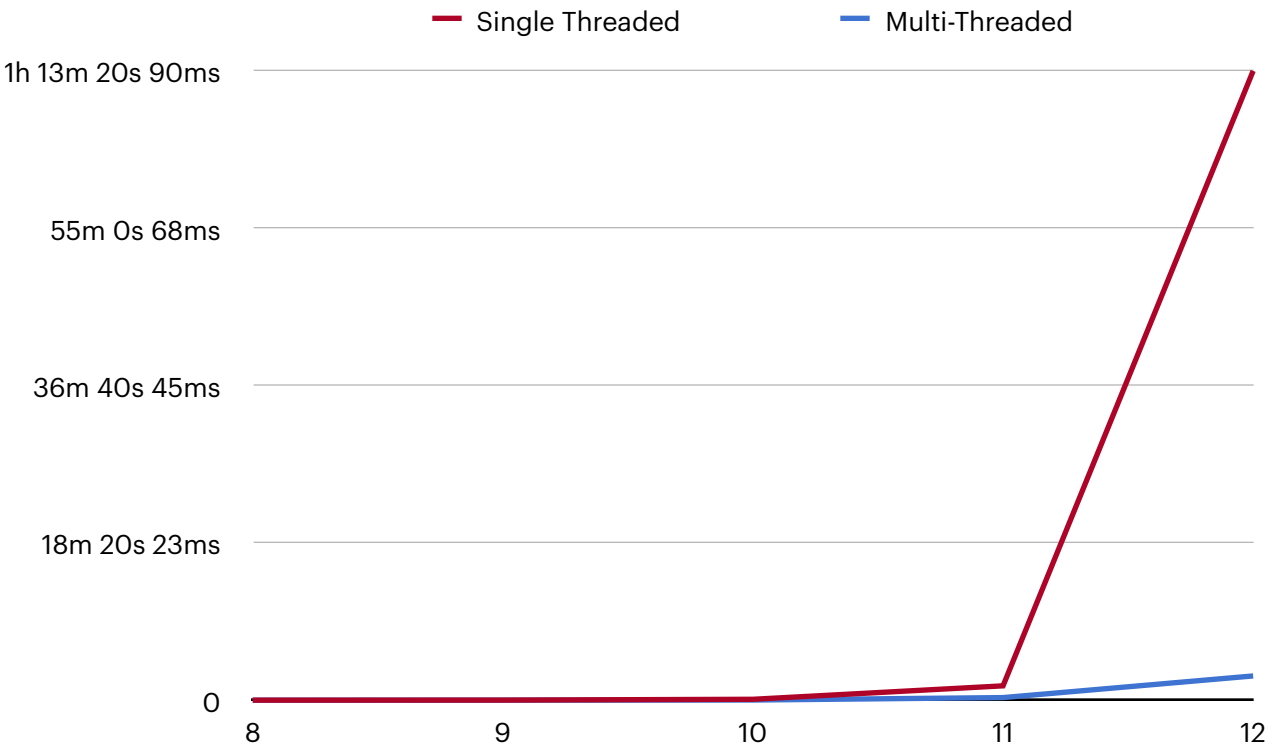
In the final solution, instead of generating a set number of sparks, I'm generating N sparks. To start, I just create a list from 0 to n. This list is then computed in parallel. Each computation involves taking this Int as the index of the first row of all permutations within. The permutations for the rest of the rows are generated within each spark.

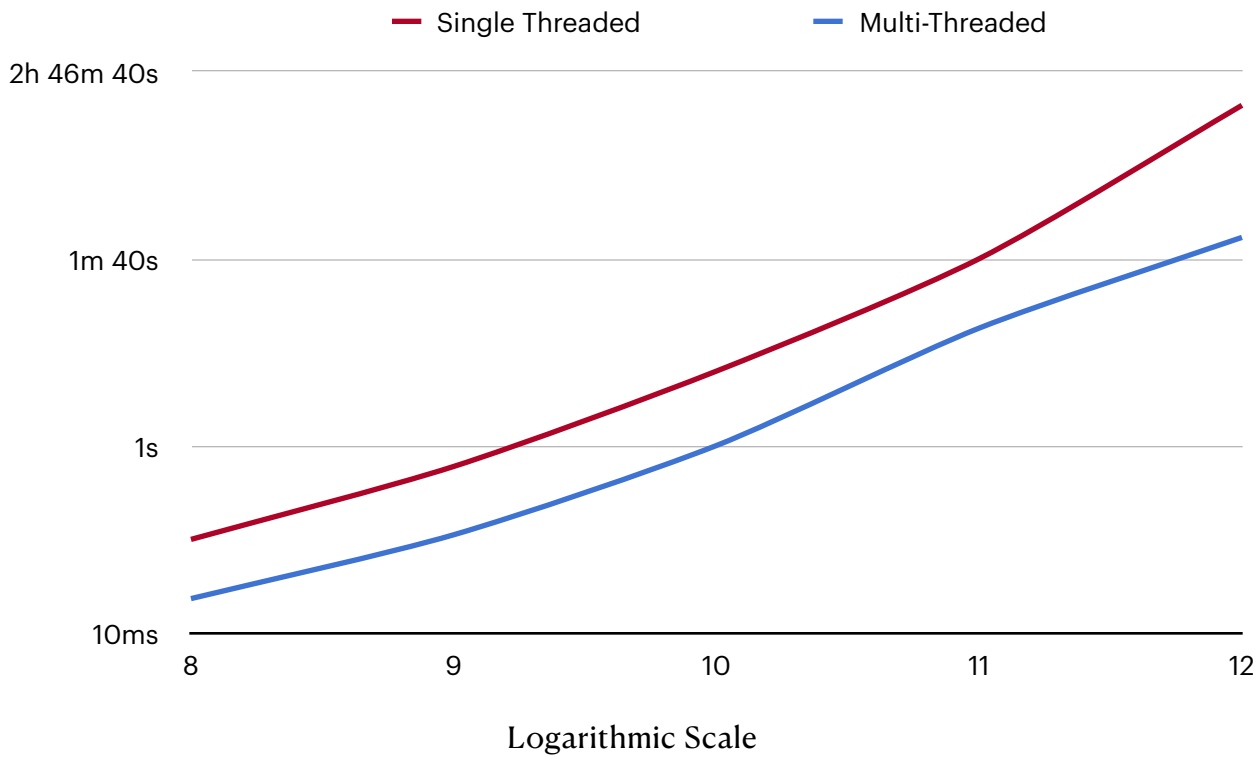
As a result, the work that needs to be sequentially is reduced to generating a list N elements long, and then adding N Ints towards to end to count the total number of solutions for a given N. Meanwhile, each spark generates and checks (N-1)! permutations. Overall, this is also means that other than the sparks themselves, this approach has no other memory overhead.

Results

While testing different values of N, I was able to see a 6X improvement in elapsed time comfortably, while often getting close to a 7X speedup. The GC performance is also quite reasonable. I'm seeing productivity rates of around 70%.

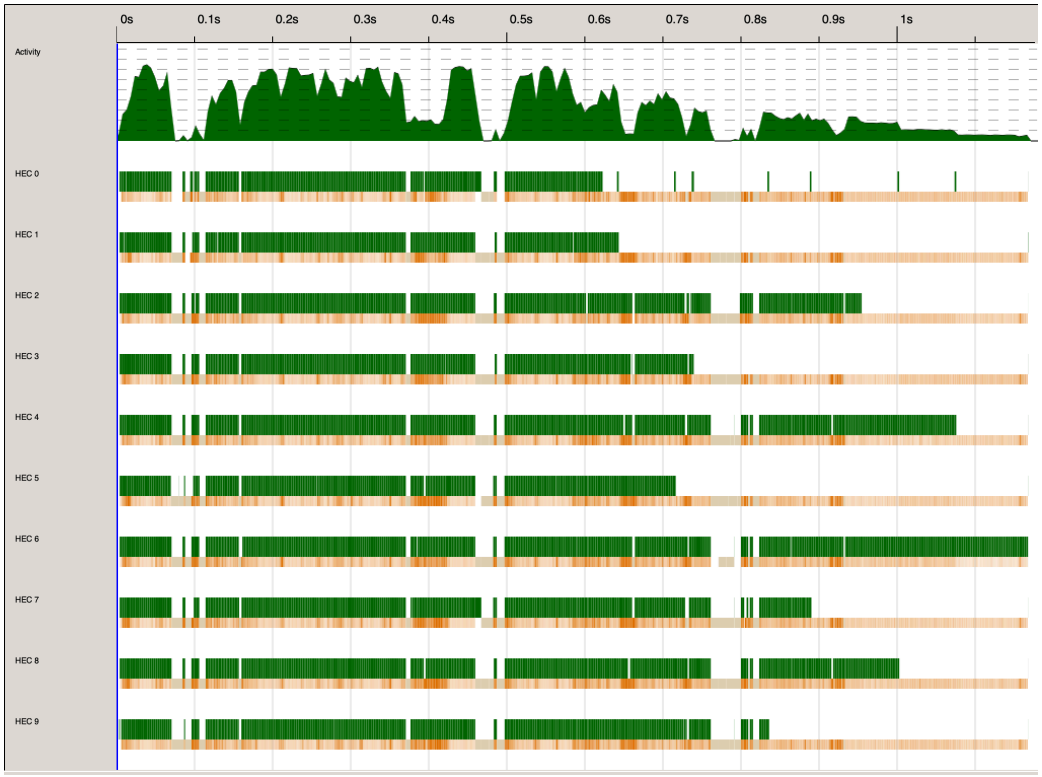
N	Single Threaded	Multi-Threaded	Time Delta
8	0.099s	0.023s	31.5%
9	0.598s	0.111s	18.5%
10	6.276s	1.002s	16%
11	100.235s	18.118s	18%
12	4400.081s	169.454s	3.8%





Threadscope

I'm seeing extremely good usage across threads. The productivity rate is close to the single threaded version.



Conclusion

As described, it is easy to see the significant speedups available by using parallel computing to calculate the number of solutions for N-queens.

The final solution I landed on works extremely well and is able to get much faster results with little overhead. I'm testing on an M1 Max ARM processor which has 8 performance cores and 2 efficiency cores. However, clock speeds are somewhat slower for parallel tasks. As such, it is fair to assume that a 6.5x speed up is close to ideal, and this implementation has little to no overhead.