ParQ: Parallel N-Queens

COMS W4995 - Parallel Functional Programming - Fall 2022
Prof. Stephen Edwards

Martin Ristovski, Anastasija Tortevska
December 22, 2022

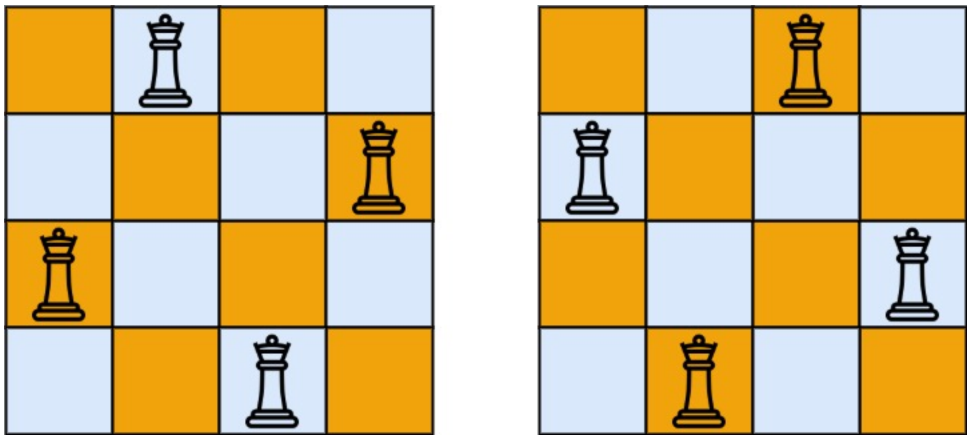
Contents

1 Introduction

The N-queens problem is a classic problem in computer science and mathematics that involves placing N chess queens on an $N \times N$ chessboard such that no two queens are able to attack each other. In other words, no two queens should be placed on the same row, column, or diagonal. This problem can be generalized to other chess pieces, such as rooks or bishops, but it is most commonly phrased in terms of queens.

The problem becomes more difficult as the value of N increases. For example, it is relatively easy to solve the problem for $N = 4$ with solution 2 or $N = 5$ with solution 10, but finding solutions for larger values of N can be quite challenging.

The N-queens problem has many practical applications, including in the field of artificial intelligence, where it is used as a test case for algorithms that search for solutions to complex problems. It is also of theoretical interest, as it can be used to demonstrate the capabilities and limitations of various algorithmic approaches.



2 Approach

We first begin by defining a sequential algorithm for solving the N-queens problem. We give an intuitive outline as follows:

1. Create a function that takes in the dimensions of the chessboard (N) and a list of queens that have already been placed on the board.

2. Check if the list of queens already placed on the board is a valid solution. This can be done by checking if any two queens share the same row, column, or diagonal. If the list of queens is not a valid solution, move on.

3. If the given list is a valid solution, check if it is a complete solution (i.e. if the number of queens on the board is equal to N). Check if it is a complete solution.

4. For each possible position, place a queen on the position and call the function recursively with the updated list of queens.

3 Implementation

3.1 Chess Board

We simulate a chessboard using a 01-matrix, where the 1 values correspond to spots where we are unable to place a queen. If the board is empty we would have an entire $N \times N$ matrix all filled with 0 entries. From here we can place a queen, on any space, which will convert any entry reachable by her using legal chess moves into 1.

3.2 Backtracking & DFS

In the N-queens problem, DFS (depth-first search) is a search algorithm that is used to explore the search space and find a solution by traversing the tree of possible states (configurations of the chessboard).

In each step of the DFS algorithm, we select a node (a state) from the search tree and explore all of its children (its possible successor states) before moving on to the next node. This is done by recursively calling the DFS function on each child node. If the function finds a valid solution (a state where all N queens are placed on the chessboard such that no two queens attack each other), it returns True, otherwise, it returns False.

We use DFS and backtracking to efficiently search for a solution by starting with an empty chessboard and trying to place queens one by one, backtracking whenever we reach a state that is not a valid solution. The algorithm will search the search tree in a depth-first manner, exploring all possible configurations of the chessboard until it finds all possible solutions.

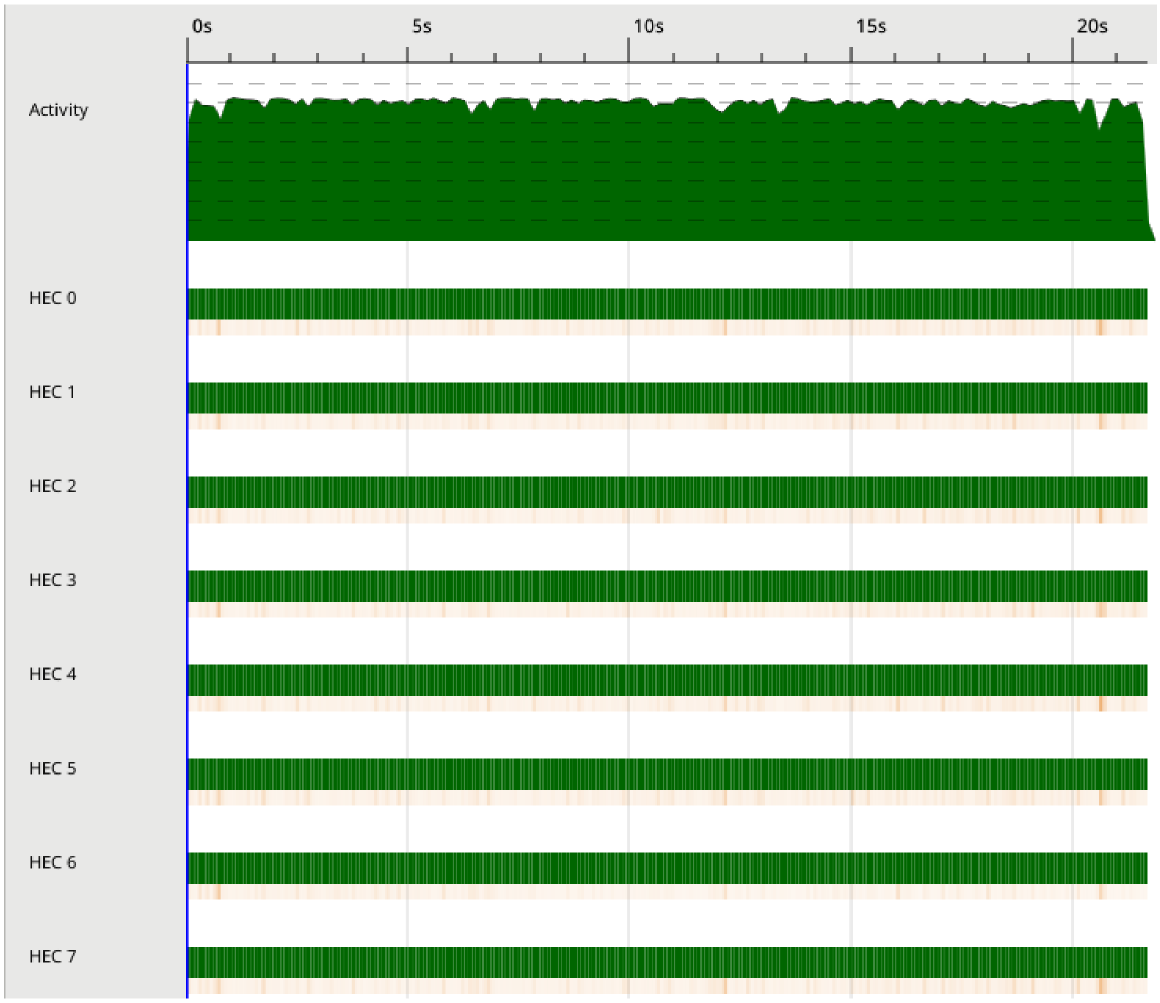
3.3 Pruning

In our algorithm, we prune away subtrees of the search tree if we see that there are more remaining queens than remaining safe positions. This is because such states cannot evolve into valid solution states by adding queens to the board.

3.4 Parallelization

The parallelization of ParQ is achieved by incorporating the Eval monad. We track the moment when all queens have been placed on the chessboard in a certain configuration, we get 1 as an answer, however, if this is not the case then we continue searching the rest of the rows while summing the result. During this process, `parList` and `rseq` are used in order to achieve the parallelization. We also limit the number of sparks we generate through the help of a user-provided spark depth parameter. We found that having a finite spark depth significantly reduces the amount of time spent on garbage collection.

Overall, our parallelization strategy does a pretty good job when it comes to evenly assigning work to all threads, as can be seen in the example threadscope output below:



4 Performance Testing and Results

We ran a number of tests to compare the performance of our parallel solution to that of our sequential solution. We show the following three plots from our testing:

1. a comparison of the time taken for Sequential N-Queens, and (using thread count 1 through 8) Parallel N-Queens with spark depth $d = 4$, and Parallel N-Queens with spark depth $d = 8$ for $n = 13$, i.e. a 13×13 board with 13 queens,

2. a comparison of the time taken for Sequential N-Queens, and (using thread count 1 through 8) Parallel N-Queens with spark depth $d = 4$, and Parallel N-Queens with spark depth $d = 8$ for $n = 15$, i.e. a 15×15 board with 15 queens,

3. a comparison of the time taken for Sequential N-Queens and Parallel N-Queens with 5 threads and spark depth $d = 4$ for values of n from 1 to 15.

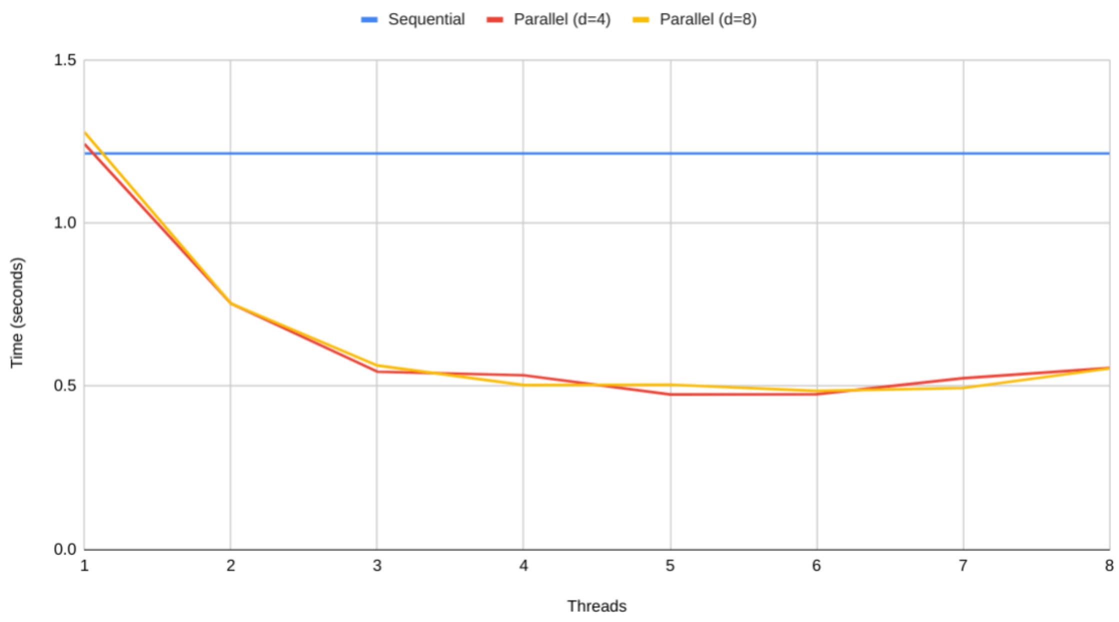
Our findings include:

- The particular value of the spark depth did not have a significant impact on the time performance of our parallel algorithm. That being said, removing the spark depth parameter from our parallelization strategy altogether (i.e. just using parList rseq) had a negative impact on performance as we ended up with a lot of fizzled and GC'ed sparks.

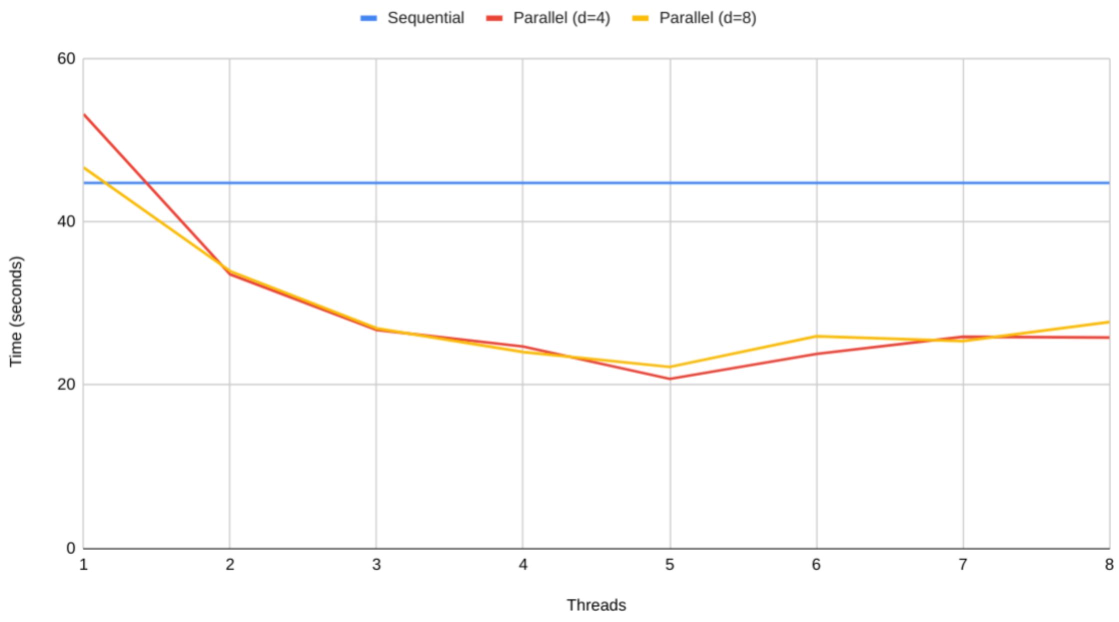
- Generally, the optimal value of threads (at least for our parameter space of $n \leq 15$) is 4-6. Going beyond that increases the parallelization overhead and going below that still leaves performance on the table. This is why we picked 5 threads in the third plot.

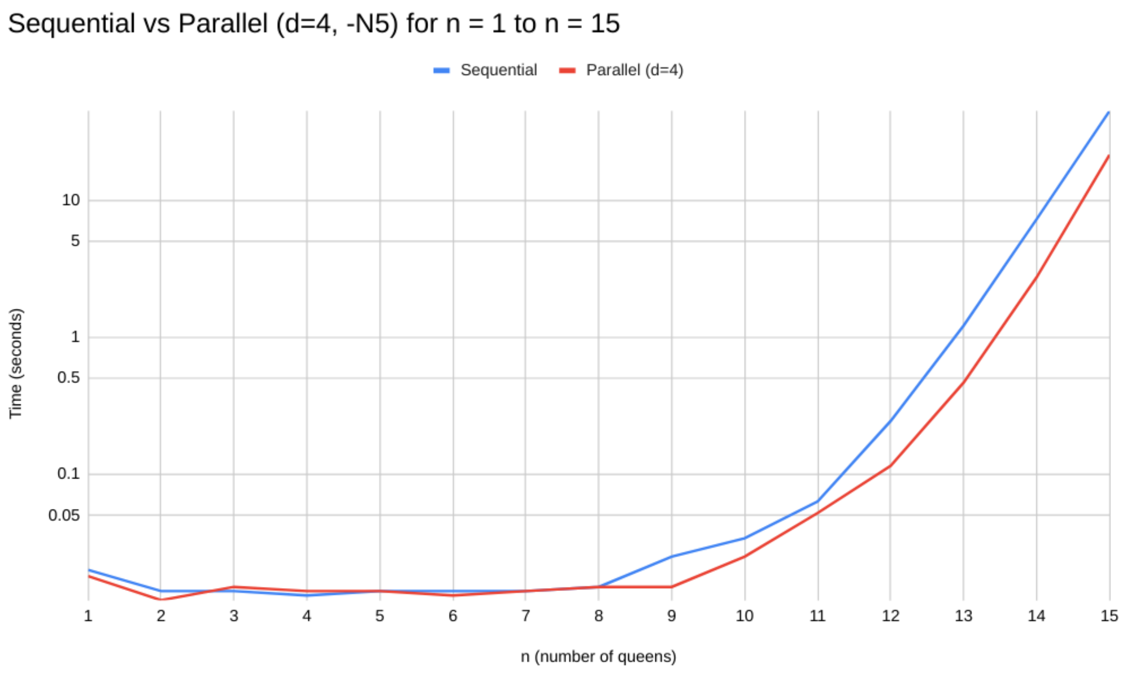
- For $n \leq 8$, there are no performance benefits to using a parallel approach. As n increases, the factor of improvement stabilizes, and for $n \geq 12$, we end up with the parallel version being 2 to 3 times faster than the sequential one.

13 x 13 board, 13 queens



15 x 15 board, 15 queens





Sequential vs Parallel (d=4, -N5) for n = 1 to n = 15
Sequential
Parallel (d=4)
Time (seconds)
10
5
1
0.5
0.1
0.05
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
n (number of queens)

5 Code

NB: Long lines have been split into multiple lines here in ways which may not be accepted by GHC. Please refer to the accompanying .tar.gz for the source files.

5.1 Main.hs

```haskell
module Main (main) where

import NQueens (nQueensParallel, nQueensSequential)
import System.Environment (getArgs, getProgName)
import System.Exit (die)

main :: IO ()
main = do
    args <- getArgs
    progName <- getProgName

    case args of
        ["par", depth, number] -> putStrLn $ show $ nQueensParallel
                                                    (read depth) (read number)
        ["seq", number] -> putStrLn $ show $ nQueensSequential (read number)
        _ -> do
            die $ "Invalid arguments. Please use the following format: \n"
                     ++ progName ++ " par <spark depth> <number of queens> \n
                              or \n" ++ progName ++ " seq <number of queens>"
```

5.2 NQueens.hs

```
module NQueens ( nQueensParallel, nQueensSequential ) where

import Data.Bits
import qualified Data.Map as Map
import Control.Parallel.Strategies(parList, rseq, r0, Strategy, runEval)

-- | The strategy for parallelizing the n-queens problem.
strategy :: Int -> Int -> [Word] -> Strategy [Int]
strategy depth number rows
    | depth < number - length rows = r0
    | otherwise = parList rseq

-- | A list of the positions that a queen can occupy on a given row.
occupiable :: Word -> Word -> Word -> [Word]
occupiable l c r = (l' .|. c .|. r') : occupiable l' c r'
    where l' = shift l (-1)
          r' = shift r 1

-- | A map of the positions that a queen can occupy on a given row.
unsafeBoard :: Int -> Map.Map Int [Word]
unsafeBoard n = Map.fromAscList [ (x, occupiable bx bx bx) |
                                                x <- [0..n-1], let bx = bit x ]

-- | Prune the search tree if the number of queens on the board is greater than
--   the number of remaining rows.
prune :: Int -> [Word] -> Bool
prune n b = colsPruned && rowsPruned
    where
        colsPruned = n >= length b + (popCount $ foldr (.&.)
                                                    ((bit n) - 1 :: Word) b)
        rowsPruned = notElem ((bit n) - 1 :: Word) $ map
                                                 (.&. ((bit n) - 1 :: Word)) b

-- | Add a queen to the board.
addQueen :: Int -> [Word] -> Int -> [Word]
addQueen n rows x = zipWith (.|.) rows $ (unsafeBoard n) Map.! x

-- | Count the number of solutions to the n-queens problem.
countMap :: Int -> [Word] -> ([Word] -> Int) -> [Int]
countMap _ [] _ = []
countMap n (row:rows) countDFS = map countDFS $ filter (prune n)
                  (map (addQueen n rows) $ filter (not . testBit row) [0..n-1])

nQueensSequential :: Int -> Int
nQueensSequential n = countDFS $ replicate n 0
  where
    countDFS [] = 1
    countDFS (row:rows) = sum $ (countMap n (row:rows) countDFS)

nQueensParallel :: Int -> Int -> Int
nQueensParallel d n = countDFS $ replicate n 0
  where
    countDFS [] = 1
    countDFS (row:rows) = sum (runEval ((strategy d n rows)
                                         (countMap n (row:rows) countDFS)))
```