K-Queens

Phillip Yan and Viktor Basharkevich

December 2024

1 Introduction

Our project, K-queens, is a play on the classic N-Queens problem, which itself is derived from the well known **Eight Queens Puzzle**, where one must place eight queens on a chessboard such that no two queens share the same row, column, or diagonal (i.e. no two queens "attack" one another". The goal is to return how many such configurations exist for a given n-dimensional board.

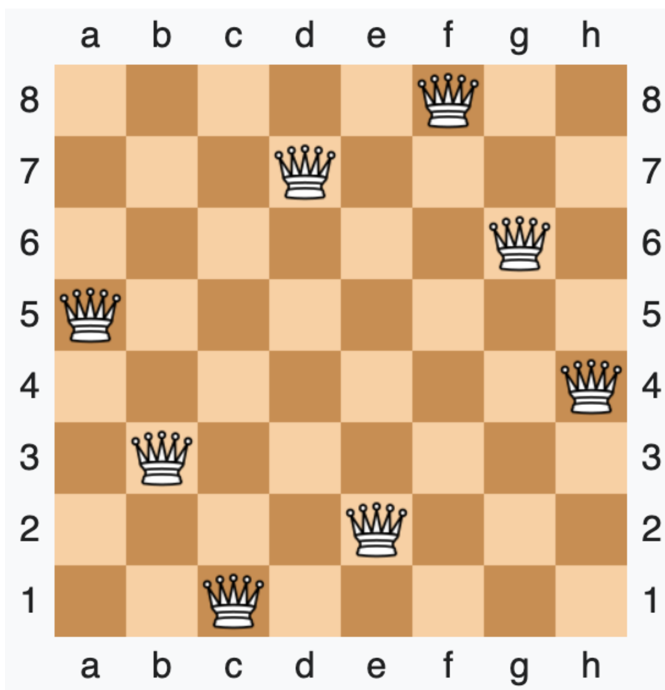

Figure 1: An example of a solution for an 8x8 board.

Initially published in 1848, the problem is known for its potential computational intensity, thus opening it up to potential optimizations via parallelizations which is why we chose this problem.

2 Sequential Solution

For our initial sequential solution, we took a brute force method by generating all possible configurations of placing N queens. This involved a recursive function which branched into two paths for each square, one where a queen is placed on that square and a second where a queen is not placed. This however, means that the branching factor is dependent on the number of squares in the grid which itself is the square of the dimension of a square board. Thus, the overall runtime is a rather abysmal $2^{(n*n)}$ where n is the input dimension of the board.

Despite the existence of more optimized algorithms, we deliberately chose this brute force approach because it offered a clear path to explore potential speedup opportunities via parallelization. The simplicity of the method made it easier to identify areas where parallel execution could reduce runtime significantly. Below is our initial code for the sequential implementation, alongside the main file which handles IO.

Listing 1: app/main.hs

```haskell
module Main (main) where

import Lib
import System.Environment(getArgs)
import Text.Read (readMaybe)
import System.Exit (exitFailure)


-- getUsageError :: String
-- getUsageError = "Usage: stack run <number>"

-- Number must be between 1 and 100 to be valid
validateNum :: Int -> IO ()
validateNum num
    | num < 1 || num > 100 = do
        putStrLn "Error: The integer must be between 1
            and 100."
        exitFailure
    | otherwise = return ()


-- We need to ensure [potentailNum] is valid, and then
    call someFunc with it as the argument
main :: IO ()
main = do
    args <- getArgs
```

```haskell
    case args of
        [potentialNum] -> case readMaybe potentialNum ::
           Maybe Int of
            Just _  -> do
                let num = read potentialNum :: Int
                validateNum num
                putStrLn (someFunc num)
            Nothing -> do
                putStrLn "Error: Argument must be an
                    integer between 1 and 100."
                exitFailure
        _ -> do
            putStrLn "Error: Please provide exactly one
                argument (integer between 1 to 100)."
            exitFailure
```

Listing 2: src/Lib.hs

```haskell
module Lib
    ( someFunc
    ) where

import qualified Data.Map as Map


-- Given a number n, makes a nxn matrix (2D) with all
    elements set to 0, using a map where the key is (row,
    col) and the value is 0
generateMatrix :: Int -> Map.Map (Int, Int) Int
generateMatrix n = Map.fromList [((i, j), 0) | i <- [0..n
    -1], j <- [0..n-1]]

getCoordsFromIndex :: Int -> Int -> (Int, Int)
getCoordsFromIndex index n = (index `div` n, index `mod`
    n)

-- Given a matrix board, place a queen at the given index
    , return the new matrix
placeQueen :: Int -> Map.Map (Int, Int) Int -> Int -> Map
    .Map (Int, Int) Int
placeQueen index board n = Map.insert (getCoordsFromIndex
     index n) 1 board

-- Validate all rows: Every row must sum to 1
validateRows :: Map.Map (Int, Int) Int -> Int -> Bool
validateRows board n = all (\r -> sum [board Map.! (r, c)
     | c <- [0..n-1]] == 1) [0..n-1]
```

```haskell
-- Validate all columns: Every column must sum to 1
validateCols :: Map.Map (Int, Int) Int -> Int -> Bool
validateCols board n = all (\c -> sum [board Map.! (r, c)
    | r <- [0..n-1]] == 1) [0..n-1]

-- Validate all diagonals: Every diagonal must sum to 1
   or less
-- validateDiagonals :: Map.Map (Int, Int) Int -> Int ->
   Bool
-- validateDiagonals board n = all (\d -> sum [board Map
   .! (r, c) | (r, c) <- [(r, d - r) | r <- [0..n-1], d -
    r >= 0], r < n, d - r < n] == 1) [0..2*n-2]

validateDiagonals :: Map.Map (Int, Int) Int -> Int ->
   Bool
validateDiagonals board n =
    let mainDiagonal = sum [Map.findWithDefault 0 (i, i)
        board | i <- [1..n]]
         antiDiagonal = sum [Map.findWithDefault 0 (i, n -
             i + 1) board | i <- [1..n]]
    in mainDiagonal == 1 && antiDiagonal == 1

-- Validate that there are n queens on the board in total
validateNumQueens :: Map.Map (Int, Int) Int -> Int ->
   Bool
validateNumQueens board n = sum (Map.elems board) == n

-- Given a matrix board represented as a map (Int, Int)
   Int, validate it (return 1 if valid, 0 otherwise)
-- The sum of every row must be 1
-- The sum of every column must be 1
-- The sum of every diagonal must be 1 (and every anti-
   diagonal must be 1)
validateBoard :: Map.Map (Int, Int) Int -> Int -> Int
validateBoard board n
    | not (validateRows board n) = 0
    | not (validateCols board n) = 0
    | not (validateDiagonals board n) = 0
    | not (validateNumQueens board n) = 0
    | otherwise = 1

-- Given a current index, a number of the remaining
   queens, and the matrix board as a hashmap,
-- return the number of ways you can place the remaining
   queens
```

```
solveNQueens :: Int -> Int -> Map.Map (Int, Int) Int ->
    Int
solveNQueens n index board
     | index == (Map.size board) = validateBoard board n
     | otherwise = solveNQueens n (index + 1) board +
        solveNQueens n (index + 1) (placeQueen index board
         n)

-- someFunc must take in an Int and return a String (that
     really just says how many
-- different ways you can solve the n-queens problem for
    a given n, the input)
someFunc :: Int -> String
someFunc n = "Answer: " ++ show (solveNQueens n 0 (
    generateMatrix n))
```

At n=5, this algorithm takes about 6.5 seconds to return the correct answer of 10. Let's see if we can do better.

3 Naive Parallelization

Given the recursive binary branching nature of our sequential algorithm, an immediate potential optimization is simply naively utilizing par for parallelization for each branch. The only change we need to do is in the main solveNQueens function in Lib.hs:

```
solveNQueens :: Int -> Int -> Map.Map (Int, Int) Int ->
    Int
solveNQueens n index board
     | index == (Map.size board) = validateBoard board n
     | otherwise = solution1 `par` solution2 `par` (
        solution1 + solution2)
     where
          solution1 = solveNQueens n (index + 1) board
          solution2 = solveNQueens n (index + 1) (
              placeQueen index board n)
```

However, although using the same n=5 (and with 6 cores) we see an immediate speedup to 2 seconds, we see some concerning issues:

- 75 million sparks created
- Only 9,000 converted
- 48 million overflowing
- Tens of millions GC'd or fizzled

We suspect two main issues are occurring here

1. **Spark Pool Overflow:** Each recursive call creates two new sparks without forcing evaluation, quickly exceeding the spark pool capacity, since creating sparks is a quick evaluation which probably outpaces evaluation.

2. **Unevaluated Thunks:** The build-up of unevaluated computations also potentially leads to increased memory pressure from stored thunks and redundant computation attempts (shown by high fizzled count) resulting in wasted parallelization effort (high GC'd count) as many sparks never getting evaluated before becoming garbage

An initial quick fix would be to utilize pseq to force evaluation, pacing the creation of sparks better with that of the actual evaluation as to hopefully not overflow the spark pool:

```
solveNQueens n index board
| index == (Map.size board) = validateBoard board n
| otherwise = solution1 `par` (solution2 `pseq` (
    solution1 + solution2))
where
     solution1 = solveNQueens n (index + 1) board
     solution2 = solveNQueens n (index + 1) (
         placeQueen index board n)
```

Indeed, this fixes the overflow problem (0 overflow) and drastically reduces the fizzle to tens of thousands. However, there remains a problem where the GC'd count is still in the tens of millions implying a lot of unnecessary branches of computation.

4 Optimization By Limiting Par Depth

As title suggests, a method we saw in class is to deal with potentially unnecessary branches of computation, namely to limit the depth of the parallelization. Note that there are n^2 layers of potential parallelization (one for each element in grid)

For this specific problem, we believe that there are three potential approaches when considering a board dimension n:

1. Using a fixed depth of par (i.e. only spark 5 times regardless if input n), then switching to sequential for the rest.

2. Using a fixed depth of sequential, (i.e. sparking until you reach the last 5 elements, which you will deal with sequentially)

3. Using a dynamic depth of par, perhaps as a function of n.

4.1 Fixed Par Depth

Let's first modify the code to do par but then switch to sequential. To do so, we modify the main input file to take in a second parameter which is par depth which we additionally add a verifier which ensures depth $< n^2$.

We also modify the Lib.hs file with both a parallel and sequential solver, and code in the parallel to ensure a transition to the sequential once the specified depth is reached.

```
-- Parallel version: handles parallelism up to a fixed
    depth
solveNQueens :: Int -> Int -> Map.Map (Int, Int) Int ->
    Int -> Int
solveNQueens 0 n board index = solveNQueensSequential n
    index board
solveNQueens _ n board index | index == (n * n) =
    validateBoard board n
solveNQueens depth n board index =
     solution1 `par` solution2 `pseq` (solution1 +
         solution2)
   where
     solution1 = solveNQueens (depth - 1) n board (index +
          1)
     solution2 = solveNQueens (depth - 1) n (placeQueen
         index board n) (index + 1)


-- Sequential Version
solveNQueensSequential :: Int -> Int -> Map.Map (Int, Int
    ) Int -> Int
solveNQueensSequential n index board
     | index == (Map.size board) = validateBoard board n
     | otherwise = solveNQueensSequential n (index + 1)
         board + solveNQueensSequential n (index + 1) (
         placeQueen index board n)

-- Updated someFunc to take depth as input
someFunc :: Int -> Int -> String
someFunc n depth = "Answer: " ++ show (solveNQueens depth
     n (generateMatrix n) 0)
```

Using n=5 (and 6 cores), we tested different depths and obtained the following data. Additionally, we also tested n=5 from 1 to 6 cores as well to see if a plateau occurs.

This results in a speedup graph of:

		Sparks				Time		
Board N Size	Depth (1 to n^2)	Sparks Total	Converted	GCed	Fizzled	Time Total	Elapsed	Speedup from sequential
5	1	1	1	0	0	6.5	3.47	1.873198847
5	2	3	3	0	0	6.244	1.93	3.367875648
5	3	7	6	0	1	6.4	1.83	3.551912568
5	4	15	12	0	3	6.51	1.49	4.362416107
5	5	31	14	0	17	6.51	1.39	4.676258993
5	6	63	22	0	41	6.727	1.735	3.746397695
5	7	127	30	0	97	6.645	1.535	4.234527687
5	8	255	38	0	217	7.176	2.167	2.999538533
5	9	512	42	1	469	6.684	1.517	4.284772577
5	10	1023	71	0	952	6.812	1.782	3.647586981
5	11	2050	47	3	2000	7.074	1.963	3.311258278
5	12	4098	63	3	4032	6.742	1.598	4.067584481
5	13	8203	114	539	7550	6.846	1.582	4.108723135
5	14	16407	93	4428	11886	6.884	1.75	3.714285714
5	15	32816	129	16570	16117	6.862	1.58	4.113924051
5	16	65633	170	45200	20263	6.782	1.498	4.339118825
5	17	131263	210	106423	24630	6.887	1.544	4.20984456
5	18	262541	243	233031	29267	6.842	1.473	4.412763069
5	19	525131	240	488991	35900	6.88	2.075	3.13253012
5	20	1054080	688	999281	54111	7.178	1.898	3.424657534
5	21	2103267	599	2045935	56733	7.091	1.669	3.894547633
5	22	4208578	717	4117332	90529	7.291	1.665	3.903903904
5	23	8408301	446	8283775	124080	7.427	1.783	3.645541223
5	24	16844393	562	16622464	221367	8.232	2.103	3.090822634

Figure 2: Data after running varying par depths at n=5.

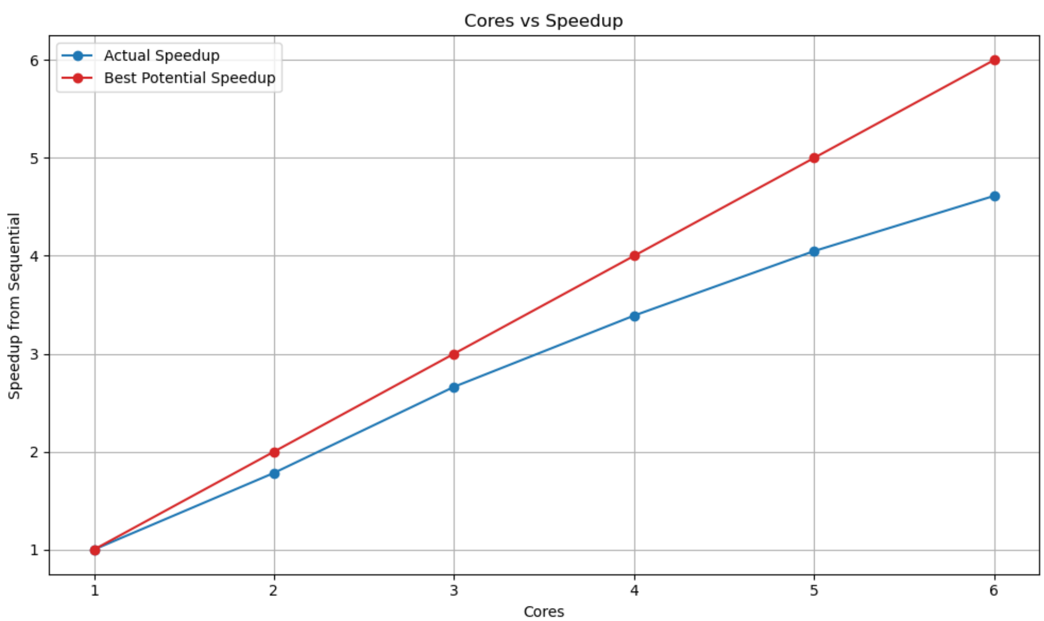

Figure 3: Cores used vs speedup at n=5 depth=5

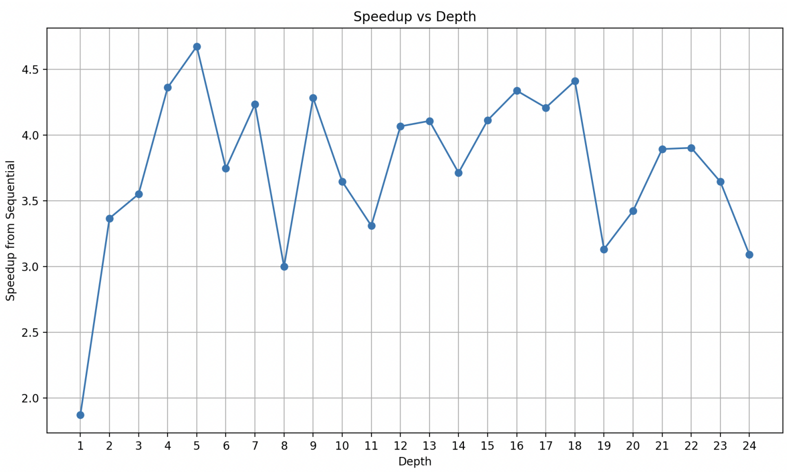

Figure 4: Speedup. Note the rough plateau after depth of 5.

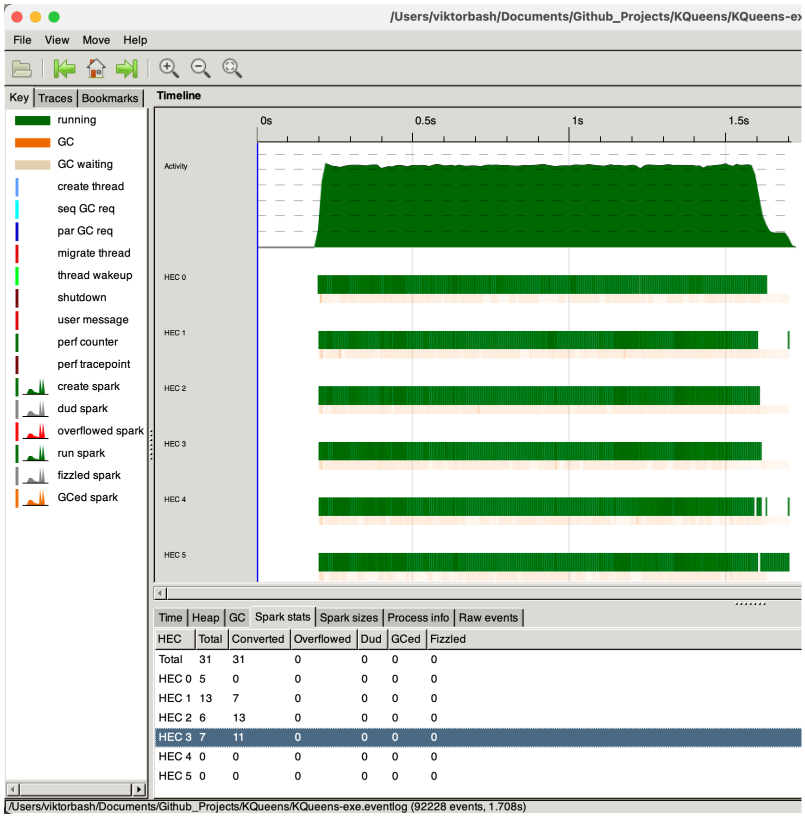

Figure 5: ThreadScope of n=5 with depth=5

4.2 Fixed Sequential Depth

Intuitively, it might not make sense to have a set depth for all n, since n can vary. Perhaps it might be better instead to fix the point at which the algorithm becomes sequential.

This is perhaps best seen where the speedup from parallelization no longer matters; see the table below for data on n = 3 and n = 4:

		Sparks				Time		
Board N Size	Depth (1 to n^2)	Sparks Total	Converted	GCed	Fizzled	Time Total	Elapsed	Speedup from sequential
3								
3	1	1	1	0	0	0.005	0.011	1.363636364
3	2	4	1	0	3	0.005	0.011	1.363636364
3	3	12	8	0	4	0.005	0.01	1.5
3	4	15	0	0	15	0.005	0.013	1.153846154
3	5	73	20	0	53	0.005	0.012	1.25
3	6	185	33	0	152	0.005	0.013	1.153846154
3	7	138	69	0	69	0.007	0.012	1.25
3	8	633	6	0	627	0.004	0.011	1.363636364
4								
4	1	1	1	0	0	0.018	0.014	2.428571429
4	2	3	3	0	0	0.017	0.013	2.615384615
4	3	7	7	0	0	0.018	0.012	2.833333333
4	4	16	10	1	5	0.018	0.013	2.615384615
4	5	35	17	8	10	0.021	0.013	2.615384615
4	6	72	19	20	33	0.022	0.012	2.833333333
4	7	145	22	65	58	0.02	0.011	3.090909091
4	8	333	60	118	155	0.023	0.012	2.833333333
4	9	598	58	331	209	0.02	0.013	2.615384615
4	10	1170	63	540	567	0.021	0.012	2.833333333
4	11	2651	59	1202	1390	0.023	0.013	2.615384615
4	12	4693	273	2861	1559	0.021	0.011	3.090909091
4	13	10158	138	5232	4788	0.024	0.011	3.090909091
4	14	17602	57	10157	7388	0.02	0.013	2.615384615
4	15	36997	69	16242	20686	0.023	0.013	2.615384615

Figure 6: Speedup. Note the limited growth in efficiency when looking at n=3

This may imply that given n=3, the 9 layers of recursive depth may not be worth parallelizing, as the overhead in creating the sparks might make it comparable in runtime to simply running the 9 layers sequentially.

Thus, although a crude analysis which would require further more extensive fine-tuned testing to find the actual fixed sequential depth, we think that the data shows a proof of concept in this potentially slightly more optimal approach over fixed par depth.

4.3 Dynamic Par Depth

A final alternative is to dynamically generate par depth based on n value. Intuitively, although a fixed sequential depth has the potential benefits we listed above, by definition fixing sequential means we cannot control the actual par depth. For example, consider n=100; given fixed sequential when there are 5 layers, there's definitely going to be spark overflow issues and inefficiencies which we described which derive from that in the naive parallelization of part 3 because of the other 9,995 layers.

Thus, to ensure balance, a more dynamic approach may be needed, which may express the optimal depth perhaps as a linear or exponential relation towards n, again more thorough testing would be needed for n¿5.

5 Other methods/considerations

5.1 Partition

We briefly considered utilizing a static partitioning for parallelization in the form of breaking a grid into smaller ones (i.e. considering a 8x8 grid as four 4x4 grids). Recursively applying this partition could potentially drastically increase speedup. However, the key problem occurs when trying to recombine the grids; the current problem format asks for the number of methods of placement rather than the different boards. However, if this problem were instead asking for board layout, this might be a viable approach

5.2 parBuffer

Given that the par depth strategy in part 4 revolved largely around controlling the number of sparks over a period of time, another potential method which allows for more granular control is parBuffer. Instead of crudely limiting sparks via limiting parallelization up to a certain depth, this allows for a sliding window. However, a key challenge is the recursive nature of the problem which makes it difficult to use parBuffer as it is applied on lists.

One way we could get this to work though would be to utilize par depth and parBuffer in conjunction; at the end of each sequential thread in the par depth method, instead of checking each board one by one, we could potentially add boards of each thread into a list, which we then apply parBuffer to, resulting in parallelization within each of the individual threads. (i.e. given n=5 and parallelization depth of say, 5, that still means $2^{25-5} = 2^{20}$ board configurations per thread. These can be aggregated and checked for verification via parBuffer).

Note for our parallelization efforts, we would end up using the parMap function (since parBuffer/parList is not necessarily as relevant for an algorithm consisting of exploring a tree of function calls generated by brute force DFS).