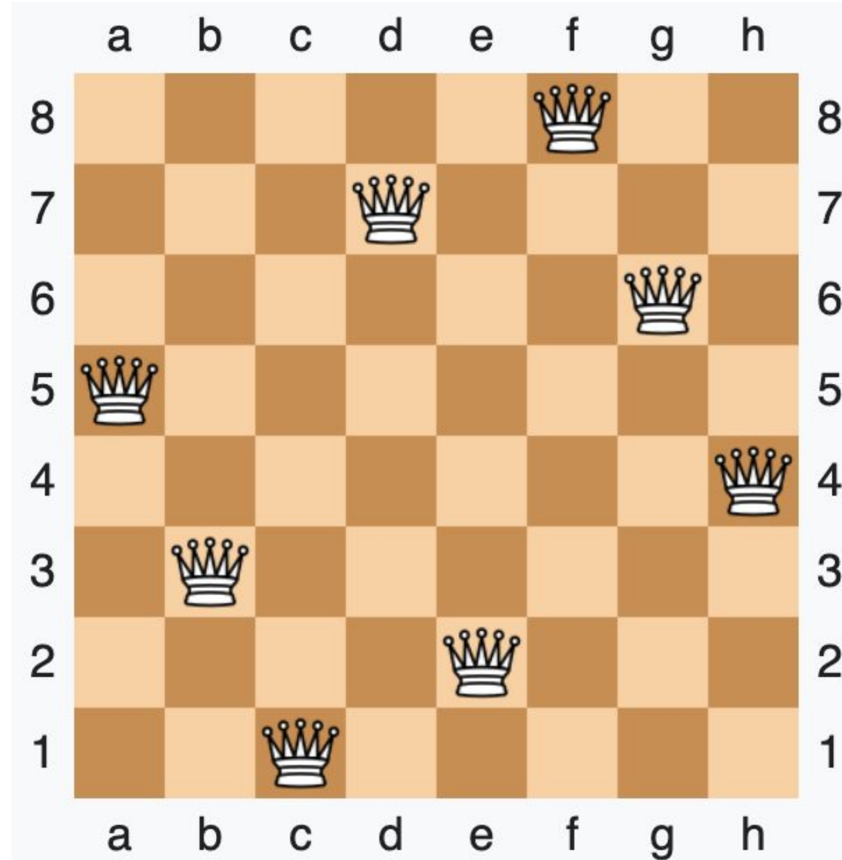
KQueens

Viktor Basharkevich
& Phillip Yan

Intro: The N-Queens Problem



Sequential Approach

```
Answer: 10
  24,200,551,320 bytes allocated in the heap
      16,152,552 bytes copied during GC
         124,808 bytes maximum residency (2 sample(s))
          73,816 bytes maximum slop
              32 MiB total memory in use (0 MiB lost due to fragmentati

                                     Tot time (elapsed)  Avg pause  Max
  Gen  0      5812 colls,  5812 par    0.126s   0.271s     0.0000s    0
  Gen  1         2 colls,     1 par    0.001s   0.001s     0.0007s    0

  Parallel GC work balance: 0.82% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.010s  (  0.010s elapsed)
  MUT     time    6.435s  (  5.863s elapsed)
  GC      time    0.127s  (  0.273s elapsed)
  EXIT    time    0.001s  (  0.012s elapsed)
  Total   time    6.572s  (  6.158s elapsed)

  Alloc rate    3,760,982,762 bytes per MUT second

  Productivity  97.9% of total user, 95.2% of total elapsed
```

```haskell
-- Validate both number and depth
validateNum :: Int -> IO ()
validateNum num
    | num < 1 || num > 100 = do
        putStrLn "Error: The integer must be between 1 and 100."
        exitFailure
    | otherwise = return ()

validateDepth :: Int -> Int -> IO ()
validateDepth depth n
    | depth < 0 || depth >= n * n = do
        putStrLn $ "Error: Depth must be between 0 and " ++ show (n * n - 1) ++ "."
        exitFailure
    | otherwise = return ()

-- Main function updated to accept depth
main :: IO ()
main = do
    args <- getArgs
    case args of
        [potentialNum, potentialDepth] -> do
            case (readMaybe potentialNum :: Maybe Int, readMaybe potentialDepth :: Maybe Int) of
                (Just num, Just depth) -> do
                    validateNum num
                    validateDepth depth num
                    putStrLn (someFunc num depth)
                _ -> do
                    putStrLn "Error: Both arguments must be integers. Number must be between 1 and 100.
Depth must be valid."
                    exitFailure
        _ -> do
            putStrLn "Error: Please provide exactly two arguments (number and depth)."
            exitFailure
```

Naive Parallel Approach

```
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)
```

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

Pseq For Forced Evaluation

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.

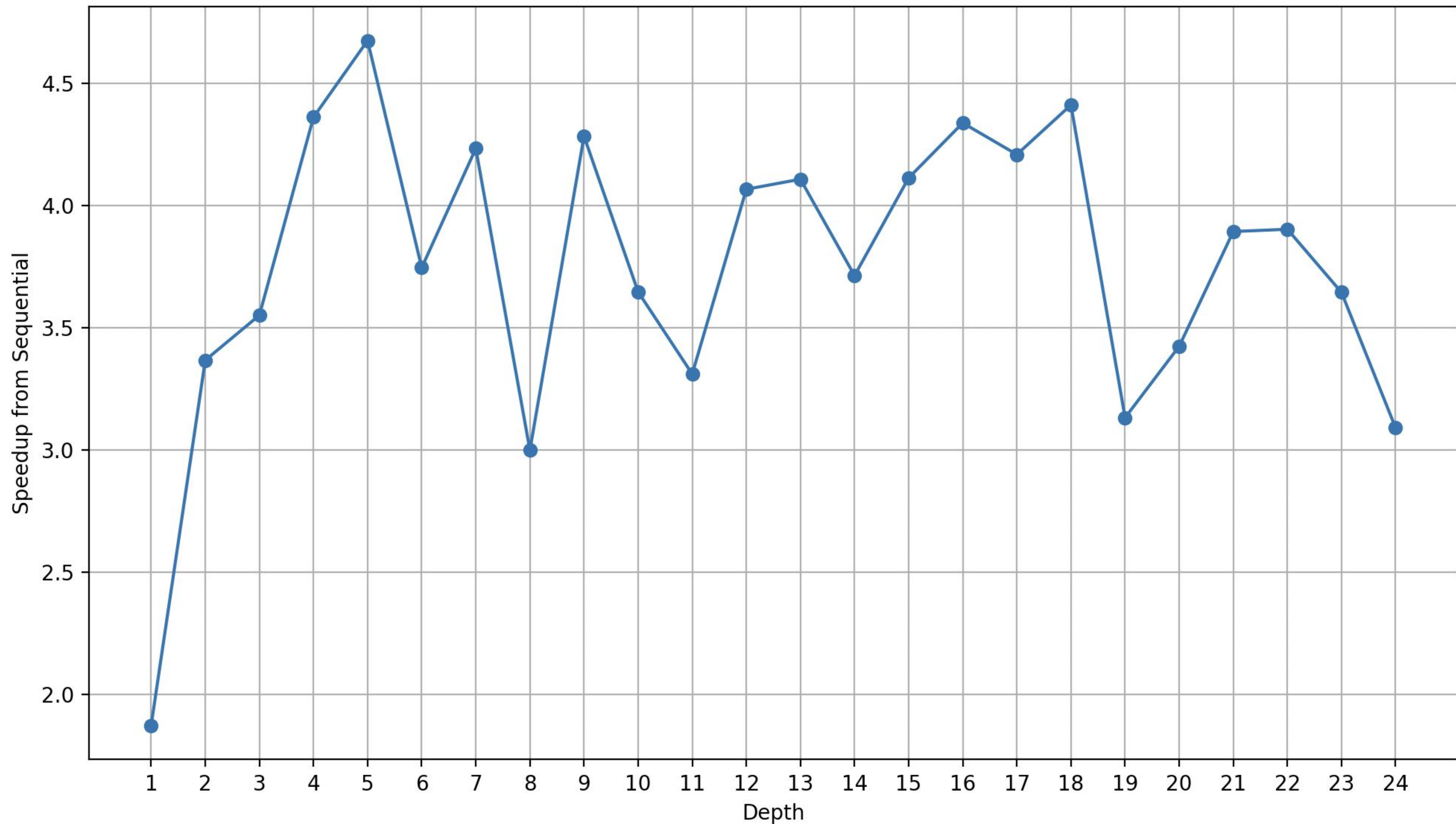
2. **Unevaluated Thunks:** The build-up of unevaluated computations also potentially leads to increased memory pressure from stored thunks and redundant computation attempts (shown by high fizzled count) resulting in wasted parallelization effort (high GC'd count) as many sparks never getting evaluated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

```
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)
```

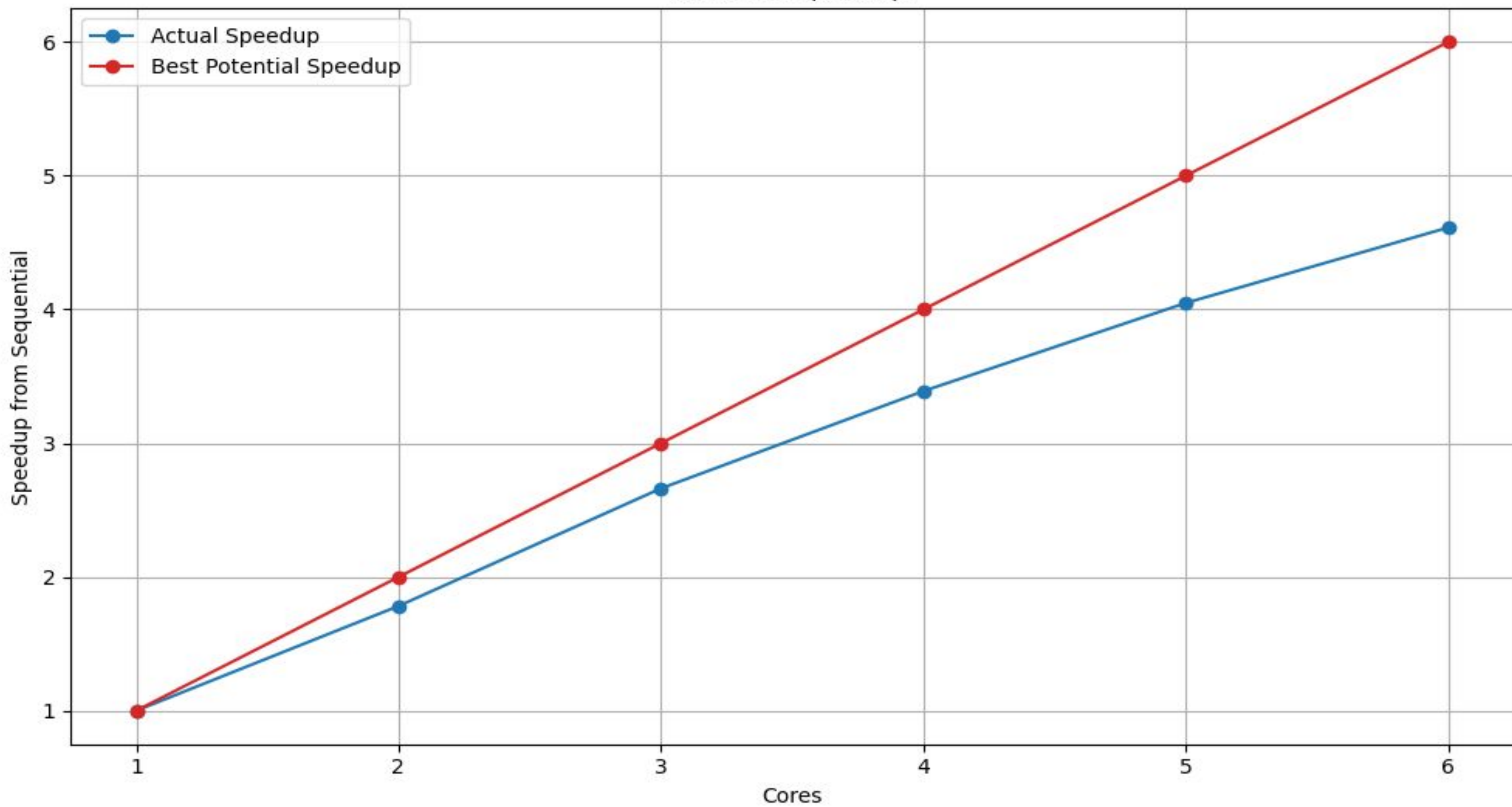
Depth Par Approach, n=5 is optimal

```haskell
-- 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)
```

		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



Speedup vs Depth
Speedup from Sequential
Depth
2.0
2.5
3.0
3.5
4.0
4.5
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24


Cores vs Speedup
Actual Speedup
Best Potential Speedup
Speedup from Sequential
Cores
1
2
3
4
5
6
1
2
3
4
5
6

/Users/viktorbash/Documents/Github_Projects/KQueens/KQueens-exe.eventlog - ThreadScope

File View Move Help

Key | Traces | Bookmarks

Timeline

- running
- GC
- GC waiting
- create thread
- seq GC req
- par GC req
- migrate thread
- thread wakeup
- shutdown
- user message
- perf counter
- perf tracepoint
- create spark
- dud spark
- overflowed spark
- run spark
- fizzled spark
- GCed spark

0s 0.5s 1s 1.5s

Activity

HEC 0

HEC 1

HEC 2

HEC 3

HEC 4

HEC 5

Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	31	31	0	0	0	0
HEC 0	5	0	0	0	0	0
HEC 1	13	7	0	0	0	0
HEC 2	6	13	0	0	0	0
HEC 3	7	11	0	0	0	0
HEC 4	0	0	0	0	0	0
HEC 5	0	0	0	0	0	0

/Users/viktorbash/Documents/Github_Projects/KQueens/KQueens-exe.eventlog (92228 events, 1.708s)

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.

Other Ideas/Approaches

```
parBuffer :: Int -> Strategy a -> Strategy [a]
```

Like `evalBuffer` but evaluates the list elements in parallel when pushing them into the buffer.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
```

A combination of `parList` and `map`, encapsulating a common pattern:

```
parMap strat f = withStrategy (parList strat) . map f
```