PFP Final Project - NQueens

Qiao Huang

December 22, 2021

Abstract

NQueens is a classic problem whose solutions grow exponentially with the input size. In this project, I implement the traditional backtracking DFS algorithm with pruning and parallelize it with the Eval monad. The results show that my parallelization strategy works well and around 6x speedup is achieved under the 8-way parallelization experiments. Some observations are discussed in the last section.

1 Introduction

NQueens is a well-known problem, that is, how many different ways can we place n queens on an $n \times n$ chessboard such that they can't attack each other. As we can see in figure 1, there are 2 solutions when $n = 4$. This project only interested in the number of solutions, and is not intended to output the detailed placement plan.

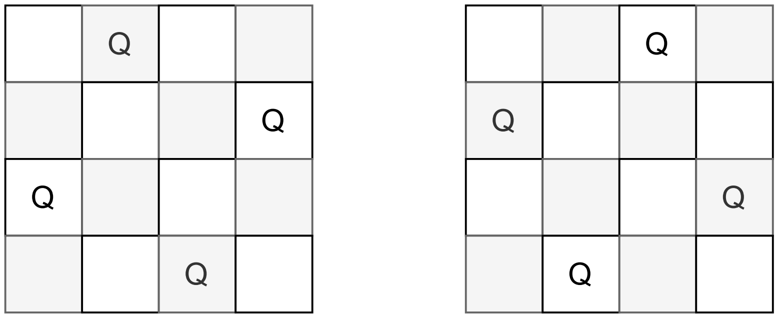

Figure 1: 2 solutions when $n = 4$

2 Methods

2.1 Sequential algorithm

A classic backtracking DFS algorithm with pruning is applied.

2.1.1 Board representation

The chessboard status is represented by a zero-one matrix, where value one means we can't place a queen on this square. If we search the solution in order such that the k-th queen we placed is on the k-th row of the board, then the top k rows are guaranteed to be filled with queens, so we only need to track the status of the bottom $n - k$ rows (Figure 2). To speed up the computation, this matrix is represented as a list of `Word` in the Haskell program, each matrix entry corresponds to a bit.

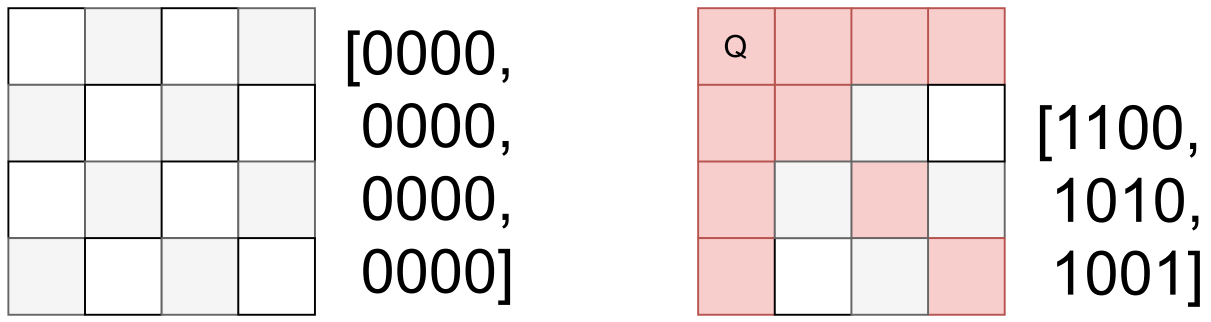

Figure 2: Represent board as a 0-1 matrix. Red squares mean that we can't place a queen there. When the board is empty, we can place a queen on any square, so it corresponds to an all-zero 4*4 matrix. If we have placed a queen in the first square, the first row of the matrix is discarded, so the representation becomes a 3*4 matrix.

2.1.2 DFS step

On the k-th step I try to place a queen on the k-th row. Since only the bottom part of the board is tracked, the k-th row is corresponding to the first row of our representation matrix, i.e. the first `Word` in the list.

Let `row` be the first element and `rows` be the remaining list. On any value 0 position in `row` we can place the k-th queen here, and then the board is updated accordingly (Figure 3). Firstly a free column in `row` is chosen to place this queen, then I compute the influence of this queen over `rows` and update it. As we can see in figure 4, there are only n different patterns of influence when placing a queen, so it's pre-computed and stored in a map to speed up the search.

2.1.3 Pruning

Two pruning strategies are used during the DFS. For example, in the case shown in figure 5, there are two more queens to be placed on the chessboard, but firstly, the third row is fully occupied, and secondly, there is only one possible column to place a queen, and other columns are fully occupied, so we shouldn't continue searching.

2.2 Parallelization

The Eval monad is used to implement parallel computation. Similar to the Fibonacci example in the class, I run parallel to a certain depth to balance between spark overhead and load balancing. When all queens are placed, i.e. the board status is an empty list, the answer is 1; Otherwise, search the following rows and sum the results up. After the first few queens are placed, I just use `r0` and don't spark threads anymore; Otherwise, I apply `parList rseq` strategy to distribute workloads. For the detailed implementation, please refer to the appendix.

3 Experiments

The first experiment aims to find the best parallelization depth, and the results are summerized in table 1. Load balancing is almost perfect when the depth is greater than two as the speed up is

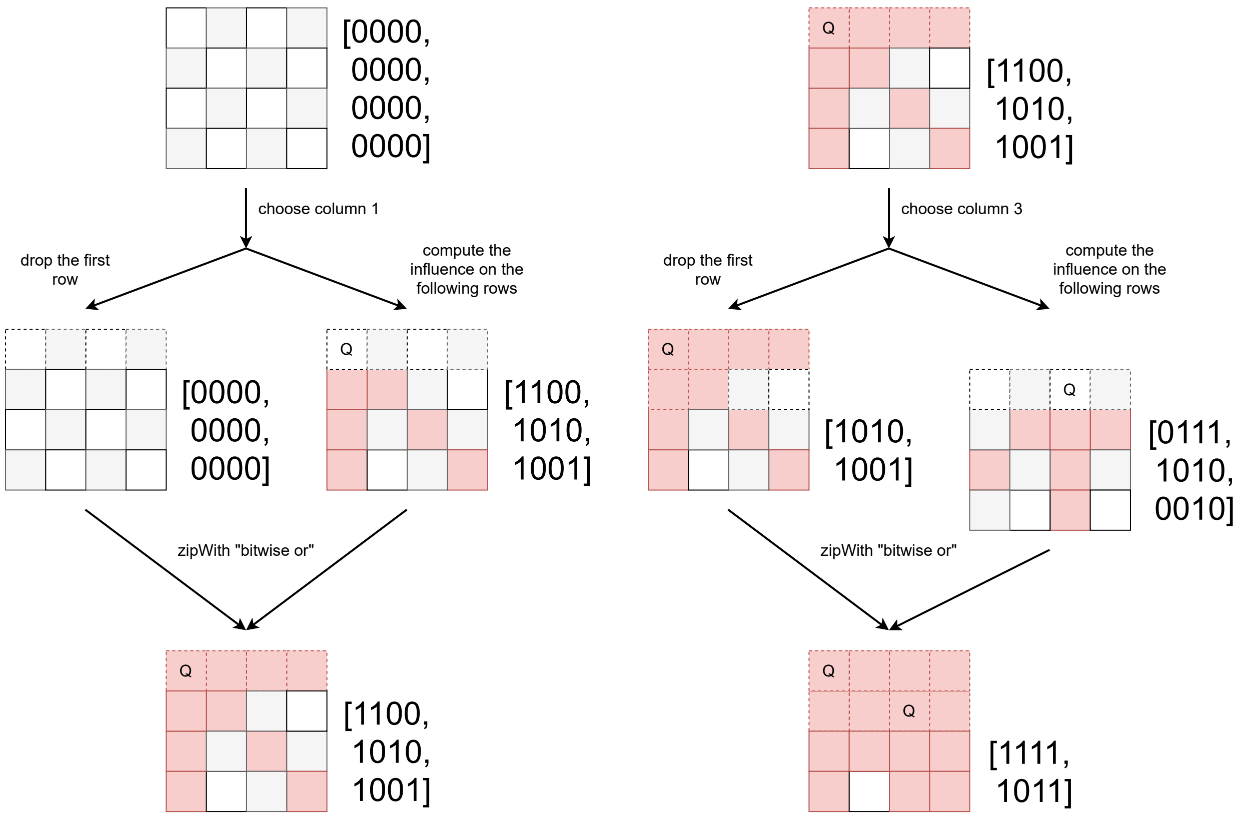

Figure 3: On the left and right are two DFS steps. The remaining board and the influence on it doesn't nessecarily have the same length. `zipWith` will automatically align them and compute the correct next board states.

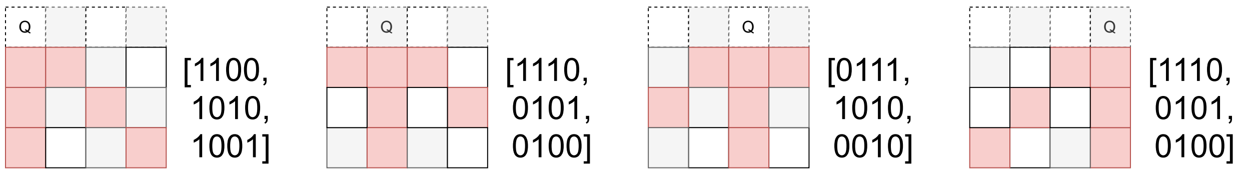

Figure 4: 4 different influence patterns when $n = 4$

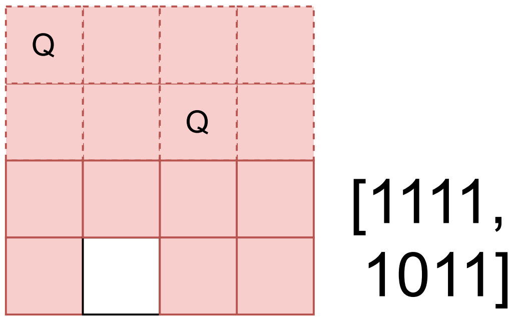

Figure 5: A example pruning case

very close to 8, and parallel overhead grows steadily along with depth. So I choose depth equal to 3 in the following experiments.

Depth	Sparks				Time (s)		Speedup
	total	converted	GC'ed	fizzled	total	elapsed	
1	15	14	0	1	40.718	5.592	7.281474
2	197	41	0	156	41.177	5.232	7.870222
3	1972	603	9	1360	42.276	5.302	7.973595
4	15989	584	36	15369	42.284	5.302	7.975104
5	105508	1168	3046	101294	43.182	5.412	7.978936
6	570734	4020	338888	227826	44.347	5.562	7.973211
7	2485313	6393	2124954	353966	44.656	5.603	7.970016
8	8307226	3544	7836934	466748	43.726	5.482	7.976286
9	20703677	4649	20056548	642480	45.030	5.642	7.981212
10	36301682	4988	35528726	767968	46.260	5.802	7.973113
11	47505513	3145	46741416	760952	47.085	5.902	7.977804
12	54862349	10462	53425107	1426780	49.586	6.212	7.982292
13	56392867	4596	55492976	895295	48.531	6.082	7.979448
14	58611134	2157	57868722	740255	48.646	6.092	7.985227
15	60931926	3550	60078427	849949	49.793	6.242	7.977091

Table 1: Parallelization overhead and running time with $n = 15$ and 8 cores

Figure 6 shows the running time when $n = 15$ under different numbers of cores. The speedup is satisfactory as the actual running time is very close to ideal.

Threadscope analysis is performed and the results are in figure 7. As you can see, about 7 cores are kept busy, and the workload is perfectly balanced. The wasted computation is mainly garbage collection.

4 Discussion

There are several observations during my development. First, I noticed that DFS is significantly faster than BFS, I believe it's because a lot more live data is kept during the BFS, which causes slower garbage collection. Second, by eliminating repeated computations, that is to pre-compute n different influence patterns and simply fetch them during DFS, I can achieve 20 percent speedup on top of the 8-way parallelization. Finally, there is around 100 gigabytes heap allocation in total, resulting in significant garbage collection overhead, maybe there are some further optimization opportunities around that.

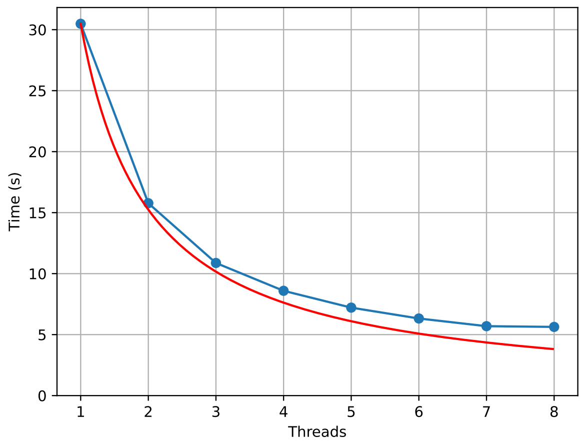

Figure 6: Running time with different numbers of cores. The red line is the ideal running time, and the blue dots are the actual times.

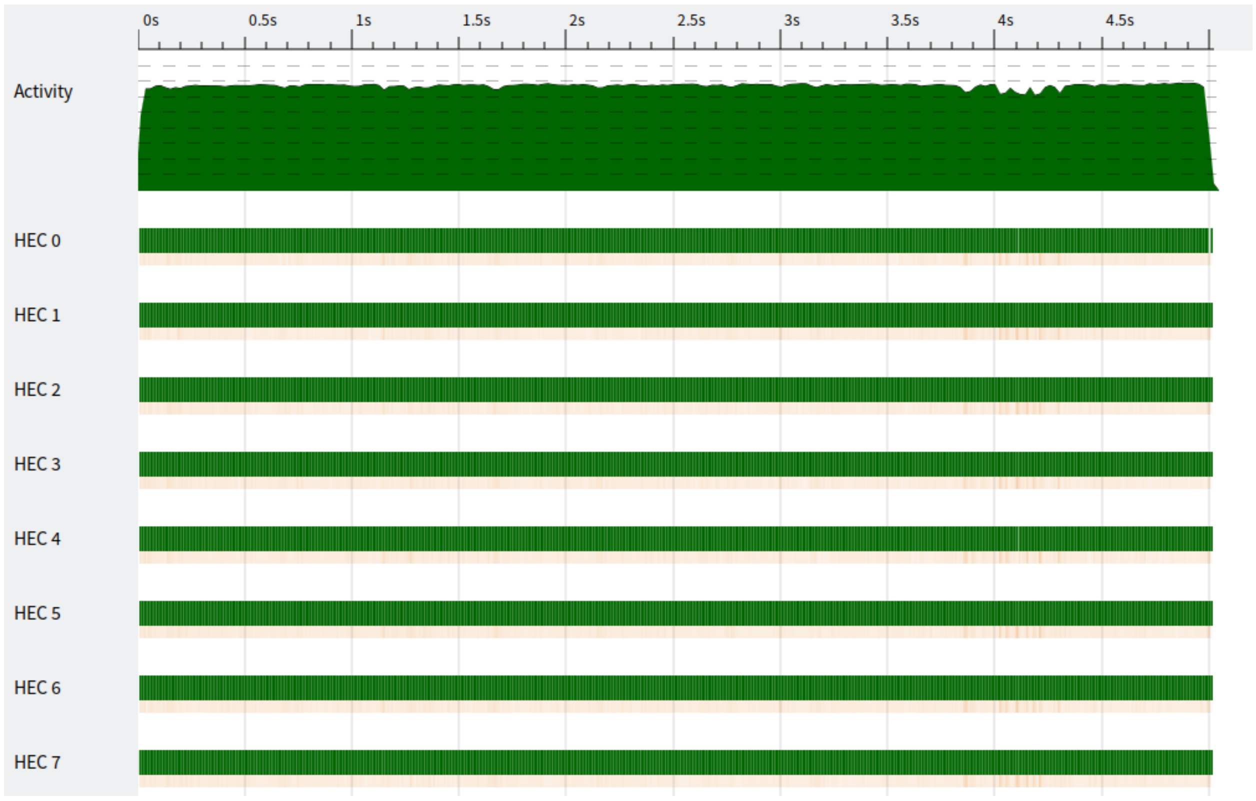

Figure 7: Threadscope analysis when $n = 15$ with 8 cores

Appendices

```haskell
type Board = [Word]

nQueens :: Int -> Int -> Int
nQueens n dep = count $ replicate n 0
  where
    mask = (bit n) - 1 :: Word

    occupyBoard = Map.fromAscList [ (x, occupy bitx bitx bitx)
                                  | x <- [0..n-1], let bitx = bit x ]
      where
        occupy c l r = (c .|. l' .|. r') : occupy c l' r'
          where l' = shiftL l 1
                r' = shiftR r 1

    count [] = 1
    count (row:rows) = sum (nums `using` strat)
      where
        nums = map count $ filter feasible boards
        strat
          | n - dep > length rows = r0
          | otherwise             = parList rseq

        boards = map placeQueenAt $ filter (not . testBit row) [0..n-1]
        feasible b = rowsFeasible && columnsFeasible
          where
            rowsFeasible = notElem mask $ map (.&. mask) b
            columnsFeasible = n >= length b + (popCount $ foldr (.&.) mask b)

        placeQueenAt x = zipWith (.|.) rows $ occupyBoard Map.! x
```

Source Code 1: My Haskell code, where `count` is the DFS function returning the total number of ways to place queens on the given chessboard, `boards` are a list of board status after placing an additional queen, and `feasible` is the pruning function.