2021 Fall Parallel Functional Programming

Parallel 15 Puzzle Problem

Alex Kuan-yao Huang kh3120@columbia.edu
Aditya Sidharta aks2266@columbia.edu

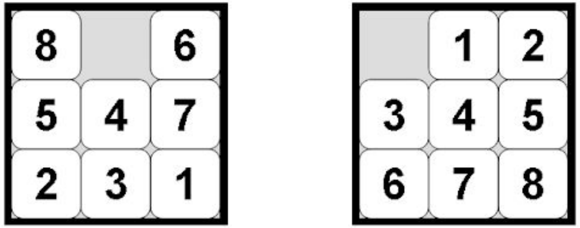
Problem Description

Input: K puzzles with size 2x2 to 4x4

Output: The step used to get the target configuration

Rules:

1. Can only swap the empty cell with the neighbors (up, down, right, left)
2. Not all puzzle is solvable, if not solvable, return -1
3. No duplicated digits appears in the input



Algorithms: this is an **NP hard** problem!

A* Algorithm: heuristic "cost function", a more balanced algorithm between path length and search space complexity

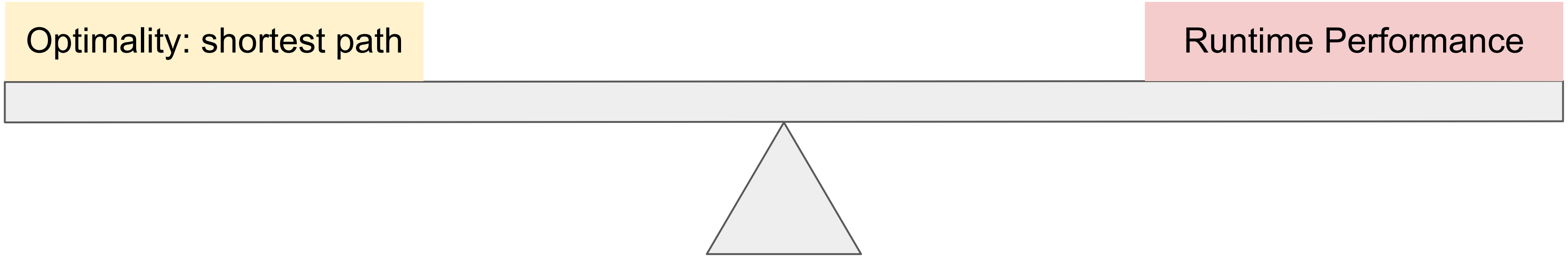
1. Manhattan Distance
2. Hamming Distance

Breadth-first search: The search space is too large

1. 15 puzzle: 16!/2 = 20922789888000
2. 24 puzzle: 24!/2 = 7.76*10^24

Greedy Algorithm (layer by layer):

Usually not optimal



Algorithms: this is an **NP hard** problem!

A* Algorithm: heuristic "cost function", a more balanced algorithm between path length and search space complexity

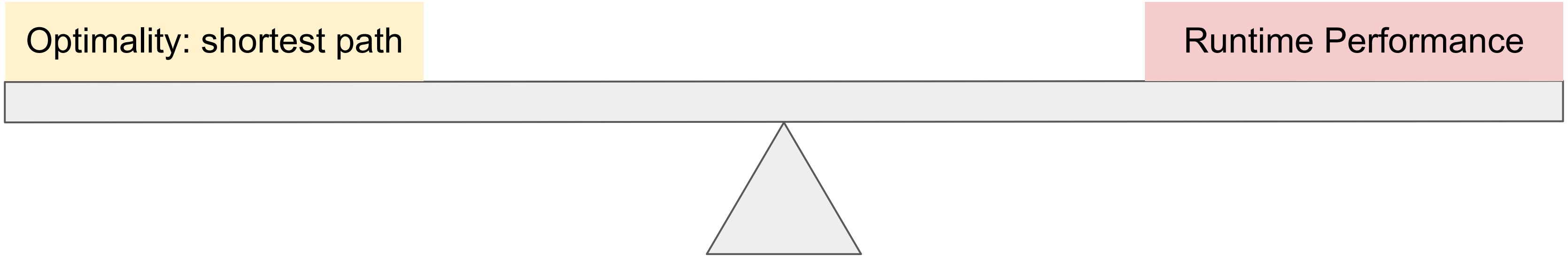
1. Manhattan Distance
2. Hamming Distance

Breadth-first search: The search space is too large

1. 15 puzzle: 16!/2 = 20922789888000
2. 24 puzzle: 24!/2 = 7.76*10^24

Greedy Algorithm (layer by layer):

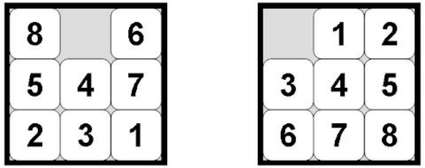
Usually not optimal



Algorithms: this is an **NP hard** problem!

A* Algorithm: heuristic “cost function”, a more balanced algorithm between path length and search space complexity

1. Manhattan Distance
2. Hamming Distance

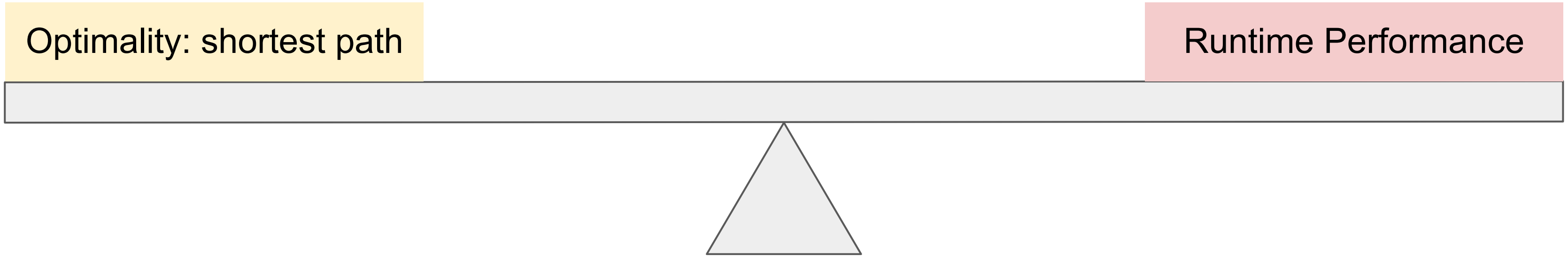


Breadth-first search: The search space is too large

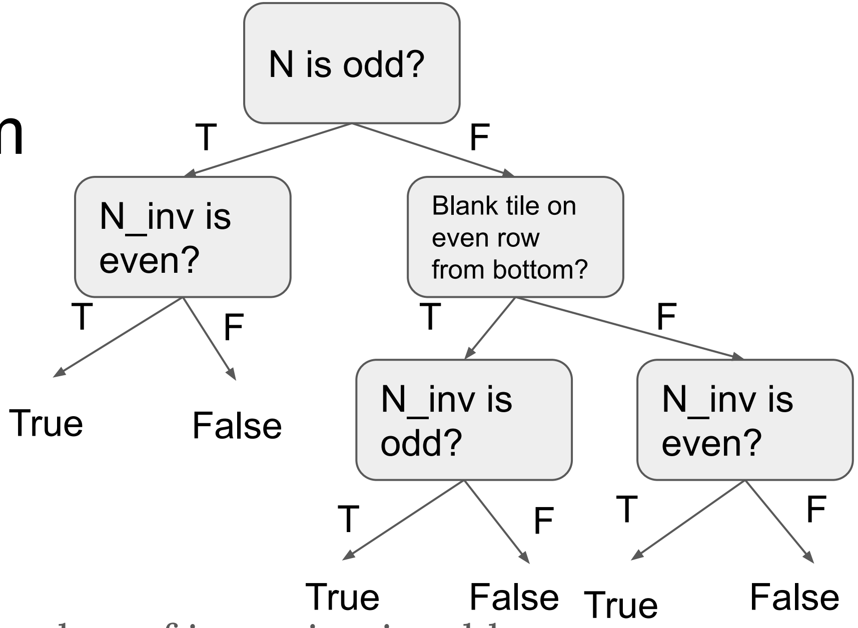
1. 15 puzzle: 16!/2 = 20922789888000
2. 24 puzzle: 24!/2 = 7.76*10^24

Greedy Algorithm (layer by layer):

Usually not optimal



Solvability of 15 puzzle problem



Input : nxn puzzle
Output: solvable or not (bool)
Algorithm Solvability :: state -> bool

```
If n is odd then
    If number of inversion is even then
        Return true
    Else
        Return false
Else if blank tile is on the even row and number of inversion is odd
    Return true
Else if blank tile is on the odd row and number of inversion is even
    Return true
Return false
```

A* Algorithm

Input : Initial State x K
Output: path length
Algorithm A* :: initialstate -> endstate -> Int

```
HashMap mp                                              \\ storing visited states
PriorityQueue pq(initialstate, cost, length=0) \\ candidates
While ( !pq.empty( ) ) do
        If pq.top().state == endstate:
                Return pq.top().length
        Let neighbors <- getNeighbors pq.top()
        Let validNeighbors <- filter neighbors by mp
        pq.pop()
        For neighbor in validNeighbors do
                Add (neighbor, cost of neighbor, length + 1) to pq
                Add neighbor state to HashMap
Return -1
```

A* Algorithm

Input : Initial State x K

1. Parallelize solving each puzzle using parList rseq

Output: path length

```
Algorithm A* :: initialstate -> endstate -> Int
    HashMap mp                                      \\ storing visited states
    PriorityQueue pq(initialstate, cost, length=0) \\ candidates
    While ( !pq.empty( ) ) do
        If pq.top().state == endstate:
            Return pq.top().length
        Let neighbors <- getNeighbors pq.top()
        Let validNeighbors <- filter neighbors by mp
        pq.pop()
        For neighbor in validNeighbors do
            Add (neighbor, cost of neighbor, length + 1) to pq
            Add neighbor state to HashMap
    Return -1
```

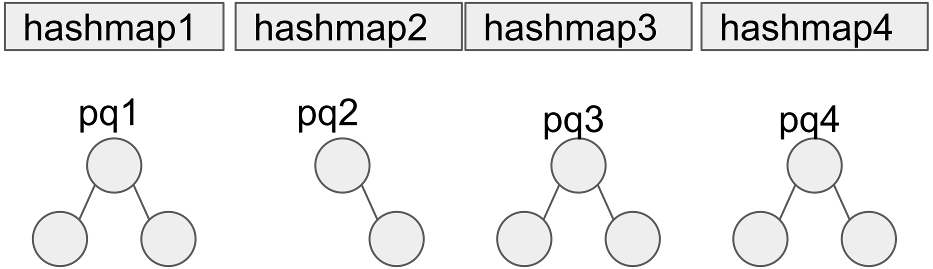
A* Algorithm

Input : Initial State x K
Output: path length
Algorithm A* :: initialstate -> endstate -> Int

```
    HashMap mp                                     \\ storing visited states
    PriorityQueue pq(initialstate, cost, length=0) \\ candidates
    While ( !pq.empty( ) ) do
        If pq.top().state == endstate:
            Return pq.top().length
        Let neighbors <- getNeighbors pq.top()
        Let validNeighbors <- filter neighbors by mp
        pq.pop()
        For neighbor in validNeighbors do
            Add (neighbor, cost of neighbor, length + 1) to pq
            Add neighbor state to HashMap
    Return -1
```

2. Parallelize getting four neighbors (will create 4 new states)

A* Algorithm



Input : Initial State x K
Output: path length
Algorithm A* :: initialstate -> endstate -> Int

```
    HashMap mp                                         \\ storing visited states
    PriorityQueue pq(initialstate, cost, length=0) \\ candidates
    While ( !pq.empty( ) ) do
        If pq.top().state == endstate:
            Return pq.top().length
        Let neighbors <- getNeighbors pq.top()
        Let validNeighbors <- filter neighbors by mp
        pq.pop()
        For neighbor in validNeighbors do
            Add (neighbor, cost of neighbor, length + 1) to pq
            Add neighbor state to HashMap
    Return -1
```

3. Creating k priority queues and finding their neighbors in Parallel. Then collect their neighbors' state to hashmap.

Intuition: the optimal route may not be heuristically the best.

Strategy 1: parallelism between test cases

- Similar to Sudoku solution explained in class, it is intuitive to parallelize the solver over different puzzles.

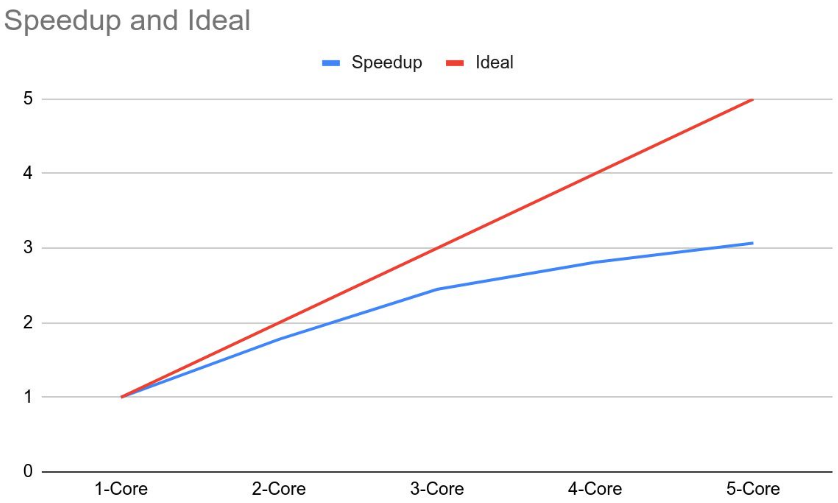
- parBuffer was used in order to regulate the number of sparks created to avoid sparks overflow

Strategy 1: Parallelism Between Test Cases

We use 100 4x4 solvable puzzles to test our performance on Linux Machine, 6 CPU, 32GB memory

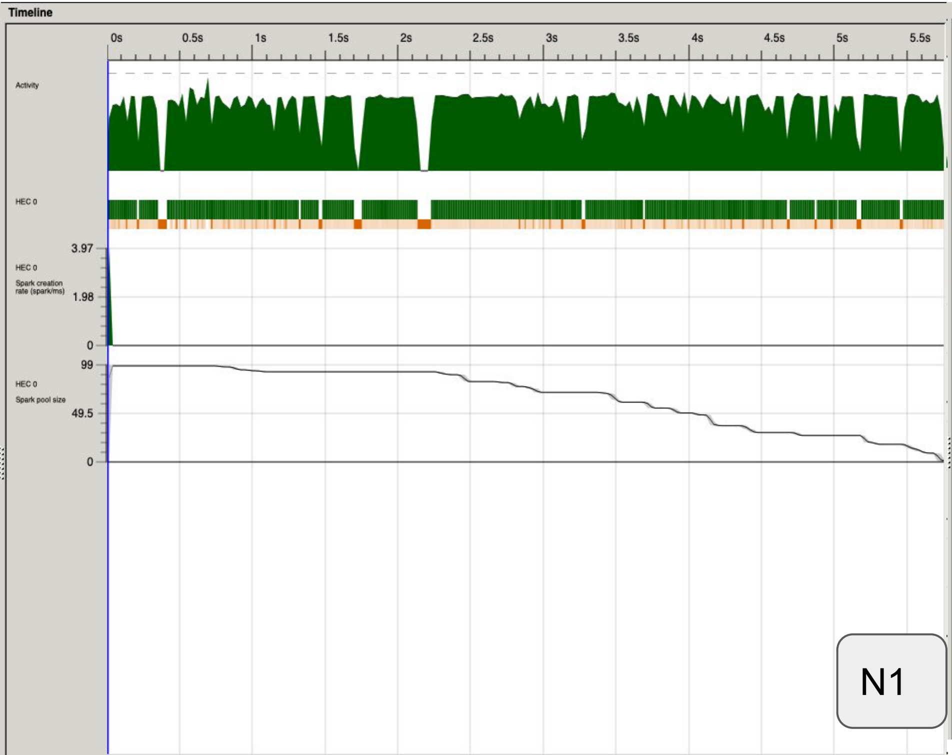
Using parList rseq

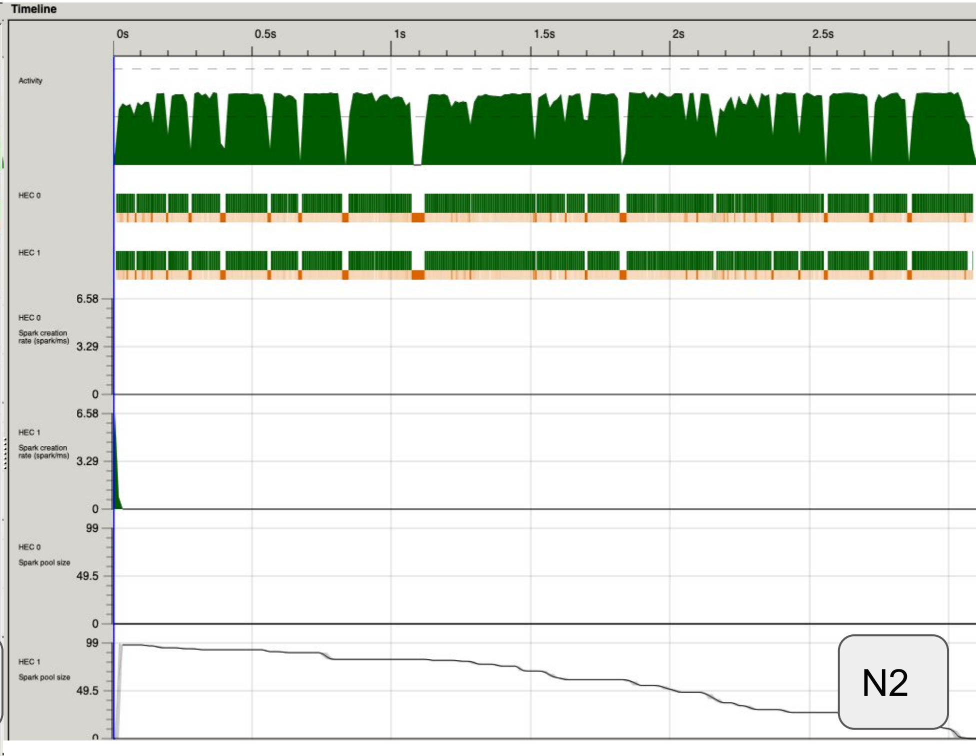
	ParallelPuzzle	Speedup
1-Core	12.73	1.00
2-Core	7.16	1.78
3-Core	5.2	2.45
4-Core	4.53	2.81
5-Core	4.15	3.07



Strategy 1: Parallelism Between Test Cases

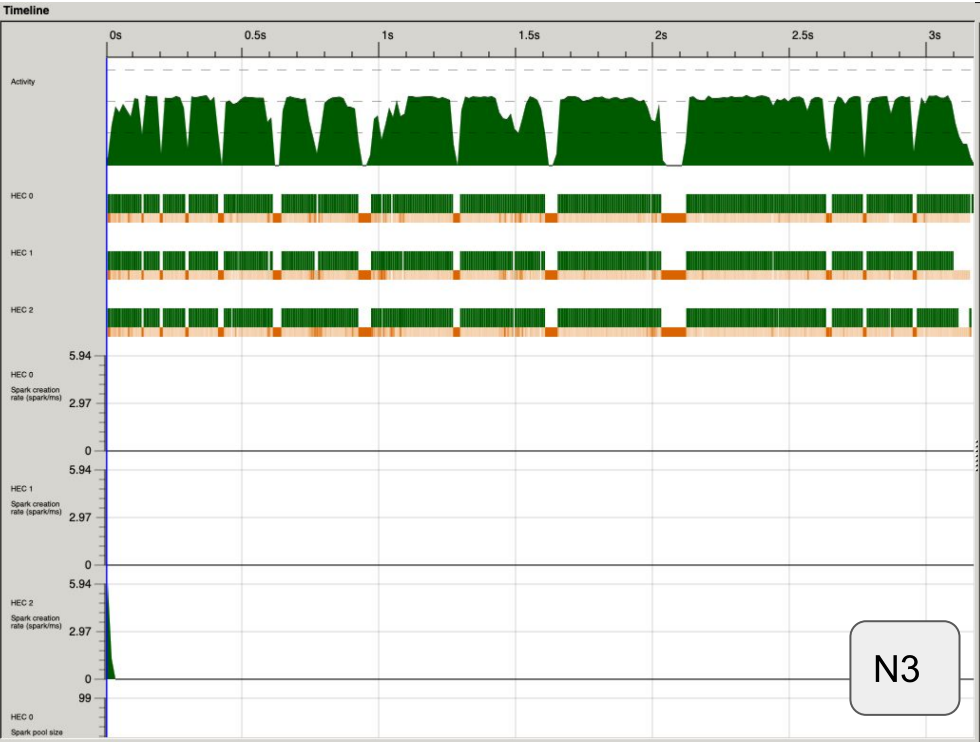
The workload is nearly evenly distributed. → Good !

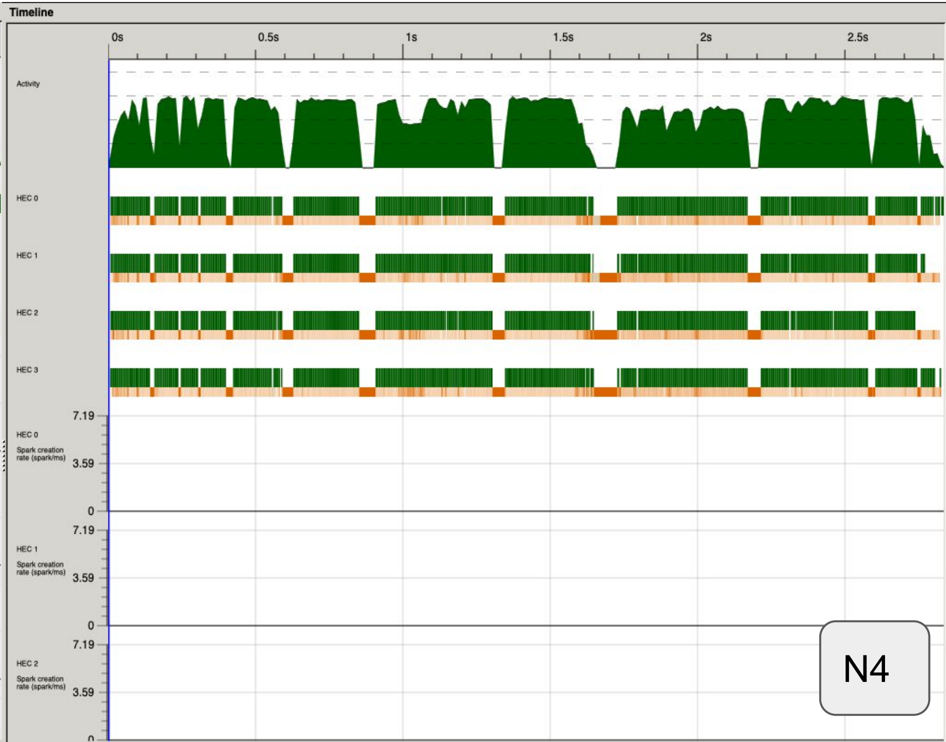




Strategy 1: Parallelism Between Test Cases

The garbage collection time is growing when number of thread is increasing.





Strategy 2: Parallelizing Neighbor Generation

- Most of the steps in A* algorithm depends on previous step
- The only obvious parallelization is the calculation of possible neighbour in each steps

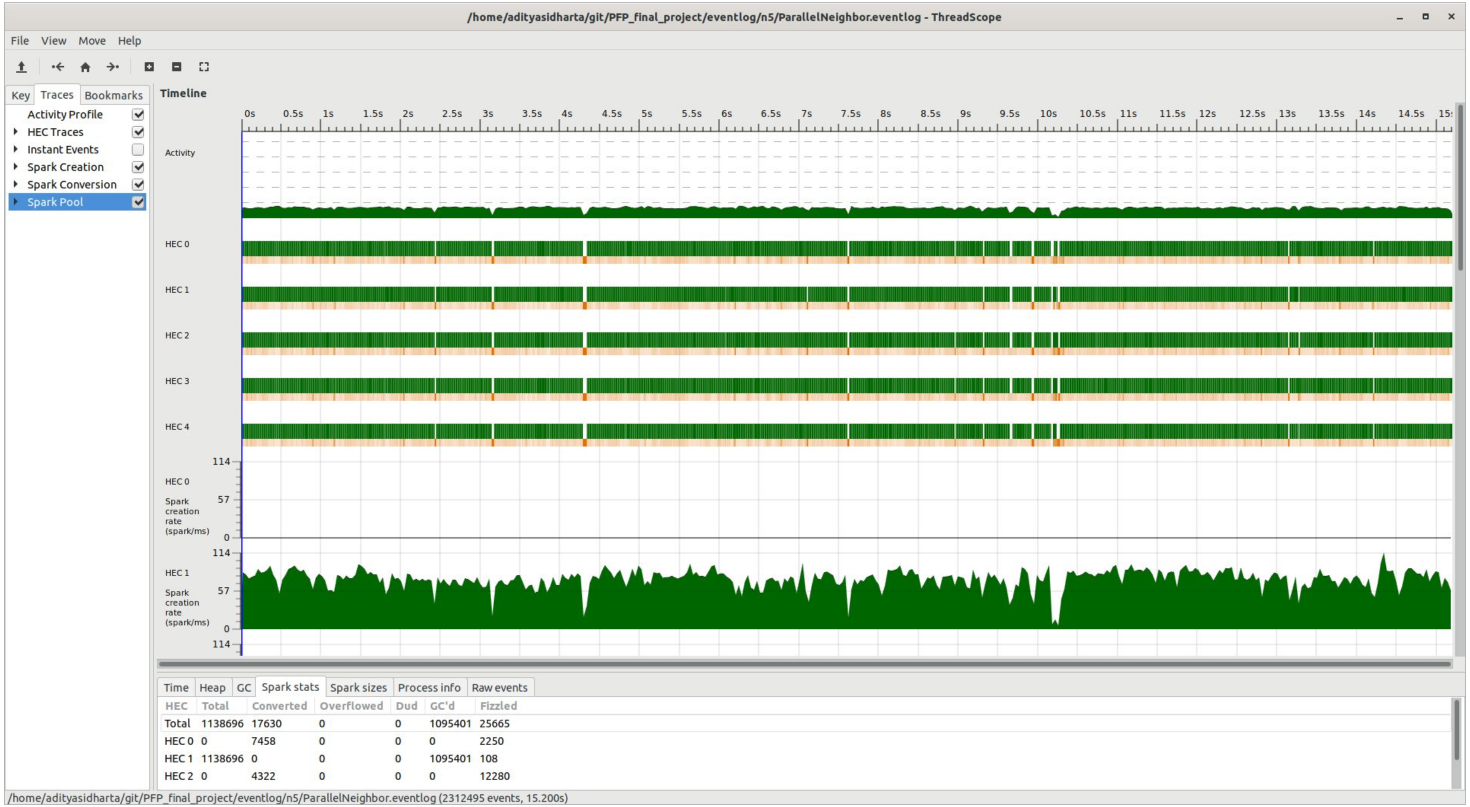
```
getAllNeighborPar:: PuzzleState -> Int -> [PuzzleState]
getAllNeighborPar p n = catMaybes (runEval $ do
    a <- rpar (getUpNeighbor p n)
    b <- rpar (getDownNeighbor p n)
    c <- rpar (getLeftNeighbor p n)
    d <- rpar (getRightNeighbor p n)
return [a, b, c, d])
```

Strategy 2: Parallelizing Neighbor Generation

	Sequential	ParallelNeighbor
1-Core	13.07	13.15
2-Core	12.89	13.38
3-Core	13.28	13.8
4-Core	13.8	14.7
5-Core	15.17	15.2

As expected, the algorithm does not work well because the calculation of the manhattan distance of a small array (4x4) is insignificant to the overhead from thread / spark creation

Strategy 2: Parallelizing Neighbor Generation



Strategy 3: Parallelism using K priority queue

- Intuition : Not all of the “best” candidate in a certain time-step will result in the optimal solution, as it is possible that the n-th best candidate in a certain time step will lead to the optimal solution
- Thus, putting K-best candidates into different priority queue, and take the result of the fastest thread as the final result might be faster.
- Each of the thread will have their own Hashmap to avoid locking problems.

Strategy 3: Parallelism using K priority queue

Algorithm

- While the valid neighbours in pq is less than K, run the sequential algorithm for the puzzle
- For (key, prio) in pq.extractMin():

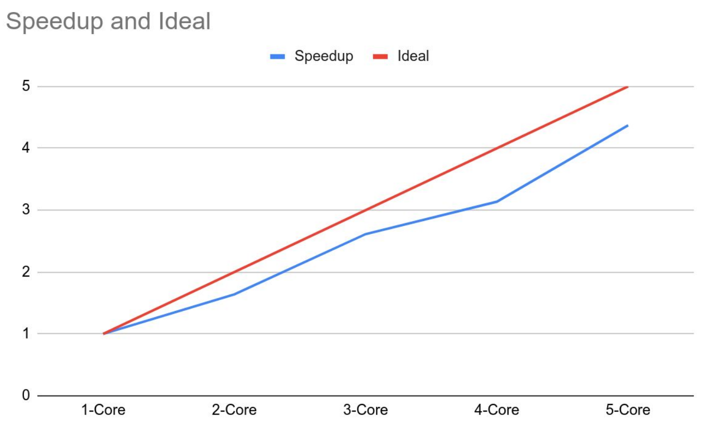
  Create copy of the current Hashmap hm

  Create a thread to solve the puzzle, with pq(key, prio) and hm
- Once any of the thread returns the result, kill all other threads and output the result from the finished thread
- Else, if all thread returns non-solvable, return non-solvable

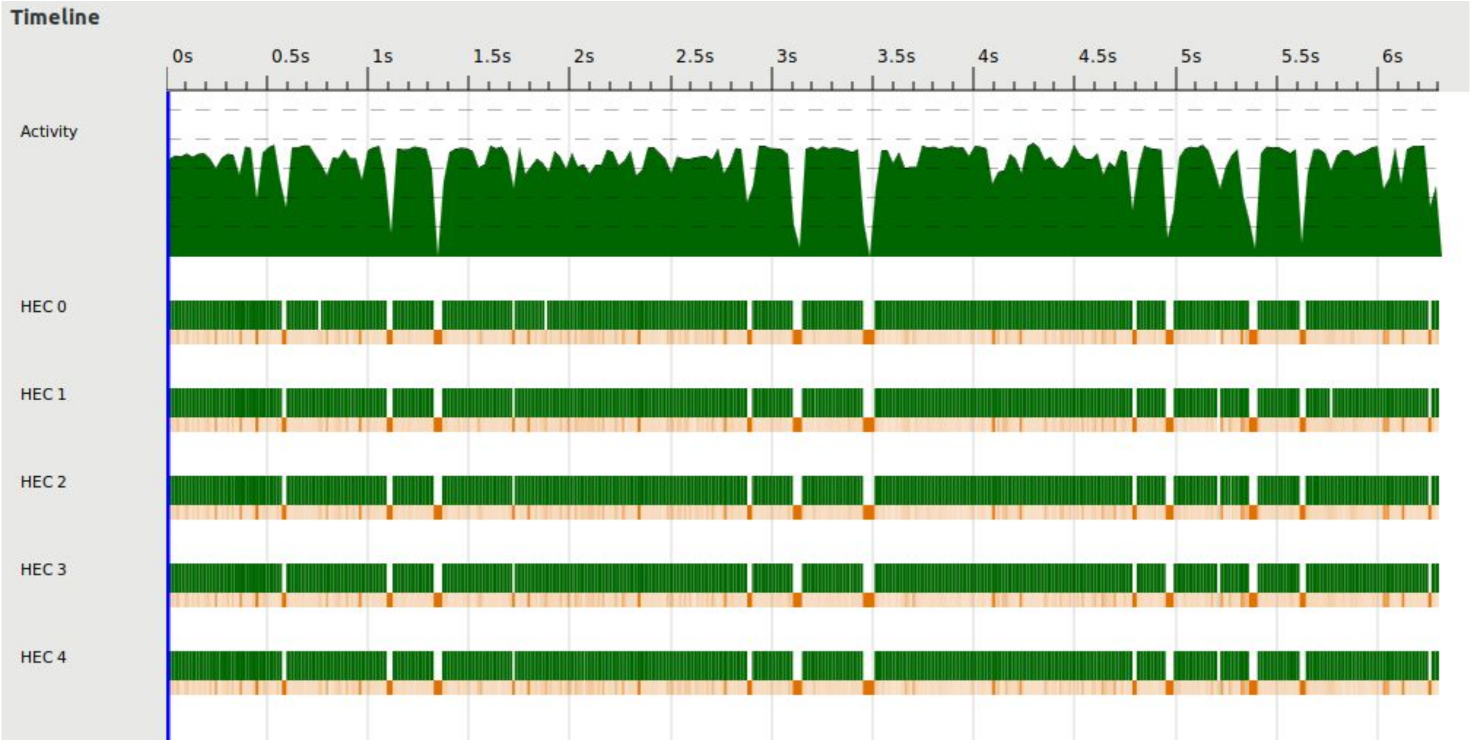
Strategy 3: Parallelism using K priority queue

	Sequential	ParallelPSQ (k=5)
1-Core	13.07	27.54
2-Core	12.89	16.8
3-Core	13.28	10.54
4-Core	13.8	8.78
5-Core	15.17	6.3

	ParallelPSQ (k=5)	Speedup
1-Core	27.54	1.00
2-Core	16.8	1.64
3-Core	10.54	2.61
4-Core	8.78	3.14
5-Core	6.3	4.37



Strategy 3: Parallelism using K priority queue



Summary

Runtime (s)	Sequential	ParallelNeighbor	ParallelPSQ (k=5)	ParallelPuzzle
1-Core	13.07	13.15	27.54	12.73
2-Core	12.89	13.38	16.8	7.16
3-Core	13.28	13.8	10.54	5.2
4-Core	13.8	14.7	8.78	4.53
5-Core	15.17	15.2	6.3	4.15

Acceleration (compared to sequential)	Sequential	ParallelNeighbor	ParallelPSQ (k=5)	ParallelPuzzle
1-Core	1.00	0.99	0.47	1.03
2-Core	1.01	0.98	0.78	**1.83**
3-Core	0.98	0.95	**1.24**	**2.51**
4-Core	0.95	0.89	**1.49**	**2.89**
5-Core	0.86	0.86	**2.07**	**3.15**

Future Works

- Implementing Shared memory between the threads to avoid extra computation (Concurrent Hash Map)
- Tune the number of cores and the number of priority queues to get the best result