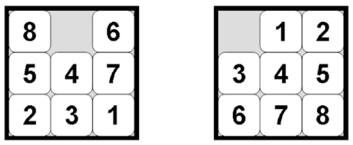
Parallel Functional Programming Fall 2021
Project Proposal – ParFifteenPuzzle

Kuan-Yao Huang - kh3120@columbia.edu
Aditya Sidharta - aks2266@columbia.edu

November 23, 2021

Problem Formulation

15 Puzzle is a sliding puzzle, which consists of $(N \times N)$ square tiles, where each squared tile is numbered from 1 to $(N^2 - 1)$, leaving a single square tile empty. Tiles that are located adjacent to the empty tile can be moved by sliding them horizontally, or vertically. The goal of the puzzle is to place the tiles in numerical order, leaving the last tile to be located at the bottom right corner of the frame.



It should be noted that not all of the initial state of 15 puzzle is solvable. 15 puzzle is solvable if:

1. N is odd
2. N is even, and the blank tile is on the even / odd row (counting from the bottom row), and the number of inversions is odd / even

Inversion is defined as the number of pairs (a, b), where $a > b$, but a appears before b if we were to flatten the number arrays into a single row. For example, [2 1 3 5 4 6 7 8] has 2 inversions $(2, 1), (5, 4)$.

Algorithm

For the 15 puzzle problem, we adapted the A^* algorithm to help us minimize the effort to backtrack all of the possible steps. A^* algorithm is an informed search algorithm, which aims to find a path to a given goal node having the smallest cost $f(n)$

$$f(n) = g(n) + h(n)$$

Where $g(n)$ is defined as the cost of the current state starting from the start node, and $h(n)$ is the heuristic function that estimates the cost of the cheapest path, attainable or not, from the current state to the goal state. In the 15 puzzle problem, the heuristic function can be defined as the sum of distances between the current entry and the target entry of the digits in that entry. The distance metrics can be chosen to be Manhattan distance or Hamming distance.

A priority queue of the possible configurations prioritizing minimal cost functions is kept during solving. We iteratively pop the most probable configuration, compute possible next steps and push them to the priority queue. The algorithm will stop when the configuration reaches the goal state.

Parallelism Strategy

The first parallelism strategy that comes into our mind for the A^* algorithm is to perform a parallel concurrent neighbor expansion, where the calculation of possible neighbors are parallelized. Nevertheless, as the Manhattan distance calculation might not be expensive, this will more likely create a massive overhead. Thus, a more effective solution is to perform parallelization by creating multiple sparks on expansion on the top-k ($k <= \|\text{pq}\|$) elements of the priority queue, the top-k potential path candidates. It is predictable that there is a trade-off between number of thread (k) and breadth of the searching tree. Since when k grows, we may expand some unnecessary nodes. However, if one of the expanded node reaches the goal state more rapidly, we can effectively reduce the search time.

Benchmark Comparison

In order to measure the effectiveness of our parallel algorithm, we will compare the running time of sequential implementation of the A* algorithm against our implementation of the parallel A^* algorithm for the 15 puzzle. In order for the running time difference to be tangible, it is required to solve a larger puzzle, i.e $N > 4$. Therefore, in our benchmark comparison, we will solve K puzzles with size $(N \times N)$, assuming that we have C number of cores. K, N, and C will be different parameters that we will inspect, in order to analyze the trend performance improvement as the number of cores or the problem complexity increases