Parallel DFS: Smarter, Faster Maze Solving

Letong Dai (ld3142), Samyukkta Suryanarayanan (ss7227)

Overview

Depth-First Search (DFS) algorithm is an algorithm that is widely used in various fields including Artificial Intelligence (AI), optimization, decision making, graph theory, and game solving. It searches a path from a given source node to a target node in a tree structure. The main advantage of DFS is that it requires less space compared to other algorithms. This makes it a good choice for processing a large amount of data.
Our goal is to design and implement an effective parallelization for this algorithm that changes its space advantage to faster running time.

Algorithm description

DFS is a widely used algorithm for traversing or searching through tree and graph data structures. It follows a systematic approach of exploring as far as possible along one branch before backtracking to explore other branches, ensuring a comprehensive traversal. Starting from a source node, DFS marks the node as visited and explores one of its unvisited neighbors, proceeding recursively (or iteratively using an explicit stack) until it reaches a node with no further unvisited neighbors. At this point, it backtracks to the previous node to continue the search along unexplored paths. By relying on a stack for its traversal mechanism, DFS ensures that all nodes and edges in the graph are visited, making it an efficient approach for exhaustive exploration. It has a time complexity of O(V+E), where V is the number of vertices and E is the number of edges, and a space complexity of O(V) for the visited list and recursion stack. However, if the input graph is a tree, then the space complexity can be reduced since not all visited nodes need to be stored.

Parallel strategy

There is a simple way to parallelize the algorithm, which is to let each thread doing DFS begin from a node that is adjacent to the source node. However, there are two problems related with this approach. The first is that for some inputs, like unbalanced trees, the workloads for threads are uneven. The second problem is that there needs to be a mechanism for threads to know which node is already visited. Therefore, we decided to take a different approach. We will have a single list for unvisited nodes and a list for visited nodes. For each thread, we maintain a separate stack for DFS. The program will be a loop that exits when a target node is visited. At the beginning of

the loop, sparks will be created, and each spark will run DFS starting at the node on top of the stacks. After reaching a certain depth, the spark will stop searching. The loop waits for all sparks to return and check whether a target node is visited. Finally, each stack is checked. If a stack is empty, choose an unvisited neighbor of the top node of another stack, add it to the empty stack and put it to the visited list. There are two arguments the program needs: number of threads available and the maximum depth each spark can run.

Data generation

To generate data for our project, we will create randomized 4x4 and 5x5 mazes as input datasets. Each maze will consist of a grid structure with predefined start and end points, where some paths are blocked by walls, and others are open to traversal. The maze generation process will ensure that each maze is solvable, with at least one valid path from the start to the goal. These mazes will be represented as adjacency matrices or lists, suitable for traversal using the Depth-First Search (DFS) algorithm.

Once the mazes are generated, we will solve each one using two versions of DFS: the sequential implementation and our parallelized version.The data collection process will involve running both implementations on the same set of mazes, with configurable parameters for the number of threads and maximum search depth for sparks in the parallel version.

We will record metrics such as runtime, memory usage, and scalability for both implementations, running each multiple times to account for variability. This process will enable us to compare the performance of the sequential and parallelized DFS algorithms across different maze sizes and evaluate the impact of parallelization on efficiency and load balancing. By generating diverse datasets and tuning the parameters of the parallel algorithm, we aim to comprehensively analyze its strengths and limitations.

Reference

1. Rao, V. N., & Kumar, V. (1987). Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, *16*(6), 479–499. https://doi.org/10.1007/bf01389000