Parallel DFS

BY: Sammu Suryanarayanan, Letong Dai

Overview

- DFS and its application in maze solver
- Sequential Implementation
- Our Parallel Implementation
- Testing

Depth First Search

- Graph algorithm to find path from a start node to a goal node
- Searches all nodes that can be reached from the current node by recursively search its neighbors
- Have a wide variety of applications
- Requires less space

DFS Maze Solver

- The maze is represented as a 2D grid (list of lists), where:
  - Open paths are represented as spaces (' '),
  - Walls are represented as '#'.
- Each grid cell is treated as a node in a graph.
- The valid neighbors of a node are the adjacent cells (up, down, left, right) that are not walls.
- The start point is the entry cell of the maze 's' in the maze
- The goal point is the exit cell ‘g’ in the maze
- The algorithm explores paths from the start node to try and reach the goal.

Sequential DFS

- What is Sequential DFS?
  - A depth-first search algorithm that explores a single path at a time using recursion.
- How It Works:
  - Starts at the initial position (start node).
  - Explores one path as deep as possible until:
    - The goal is found, or
    - A dead end is reached (backtrack to explore other paths).
- Key Features:
  - Uses a visited list to avoid revisiting nodes and prevent cycles.
  - Implements recursive backtracking to explore all possible paths systematically.
  - Constructs the solution path as the recursion unwinds.
- Advantages:
  - Simple to implement.
  - Guaranteed to find the goal in finite mazes.
- Limitations:
  - Single-threaded: Explores paths one at a time, making it slower for large mazes.
  - Not ideal for highly complex or large graphs where parallelism can help.

A simple approach to parallelize DFS

If current node has only one neighbor, search that neighbor. If it has more than one neighbor, creates a spark for each of its neighbors.

Problems:

1. Needs to wait for each spark to complete
2. Repetitive search of nodes

Problems of the simple approach

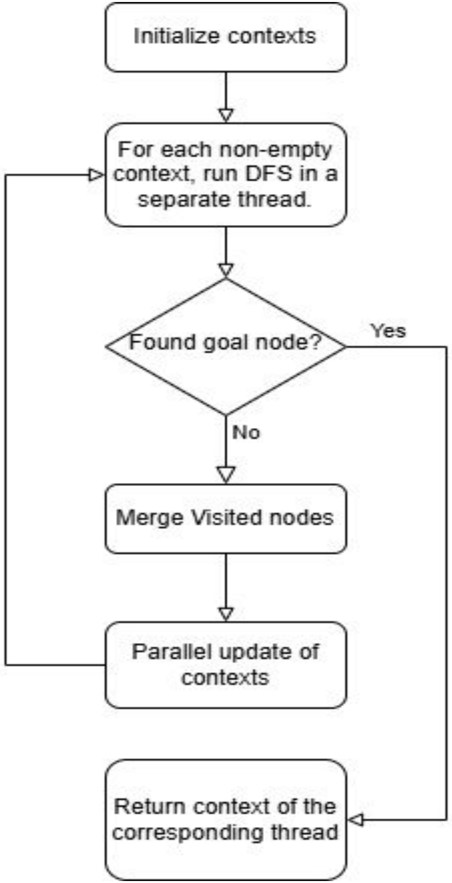
Needs to wait for each spark to complete.

Problems of the simple approach

Search already visited path may be a waste of time and computation resources

Parallel DFS

- Initialize contexts. Each context is a list of searched nodes. The first node in the context is next node to be searched
- Initialize a shared set of all visited nodes
- Run DFS in parallel for each non-empty context. Restrict the number of nodes each thread can search.
- If found goal node, return the context of that thread.
- Else merge new visited nodes to the shared set
- Assign unvisited nodes for each thread by adding them to each context. This step also has parallelization



Parallel DFS

- Each spark searches at most n nodes
  - Merge new visited nodes after every spark returns
  - Alleviate the problem of searching repeated nodes

- In Each iteration, the algorithm adjust the number of sparks dynamically
  - Avoid creating unnecessary sparks

- Update contexts in parallel
  - Further improvement to the algorithm

Test

- Input Data:
    - Mazes are provided as text files containing walls (#), paths (' '), start (S), and goal (G).
    - Different maze sizes (e.g., 70x70, 100x100) are used to test scalability.
- Execution:
    - Run both Sequential DFS and Parallel DFS on the same maze.
    - Measure execution time for each implementation.
- Performance Measurement:
    - Execution Time:
        - Measured using getCPUTime to calculate runtime in seconds.
        - Ensures accuracy by timing each function runtime independently.
    - Correctness:
        - Check if a valid path is returned.
        - Compare the outputs of Sequential and Parallel DFS.
- Comparison:
    - Evaluate speedup achieved by Parallel DFS over Sequential DFS.
    - Analyze the impact of parameters such as thread count and depth limit.
- Output:
    - Results include execution time and whether the goal was found.
    - Paths can optionally be visualized by marking them on the maze.

Thank you

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