CS 4995 Final Report

Parallel DFS

Letong Dai (ld3142), Samyukkta Suryanarayanan (ss7227)

Abstract:

This report explores the implementation and performance analysis of sequential and parallel Depth-First Search (DFS) algorithms, applied to maze-solving tasks. The parallel DFS leverages Haskell's parMap to enable simultaneous exploration of multiple paths, achieving significant speedups over the sequential approach in tests on various mazes. Performance evaluations, supported by ThreadScope graphs, reveal that parallel DFS benefits from increased thread counts and optimized depth settings, reducing execution time and finding shorter paths. However, challenges such as workload imbalance, threading overhead, and garbage collection pauses limit efficiency, especially with higher thread counts. These findings underscore the potential of parallel DFS to enhance computational efficiency while highlighting the need for optimizations in workload distribution, memory management, and dynamic depth tuning for scalable real-world applications.

Introduction

DFS is a graph searching algorithm which searches for a path connecting two vertices given a source vertex and a sink vertex. It is widely used in areas including Artificial Intelligence (AI), optimization, theorem proving, and graph theory.

The DFS algorithm searches the graph pathwise. Beginning from the source vertex, it is recursively executed on each of the unvisited neighboring vertices. This means it will search all paths beginning from the previous unvisited neighbor before searching the next unvisited neighbor. In our program, we chose to implement a maze solver using the DFS algorithm. We parallelized the DFS, tested its performance on different numbers of threads, and compared the speed up with the sequential version.

Algorithm Description

The algorithm takes the following parameters as input: a matrix representing the maze, a starting vertex and a goal vertex. The matrix is a two dimensional list with four types of elements: passages, start, goal, and walls. The starting vertex is a (Int, Int) index for the start, and the goal vertex is a (Int, Int) index for the goal. The pseudo-codes below shows the execution of the algorithm:

```
dfs :
        if start is goal:
                return start
        else:
                add start to visited node
                neighbors <- get neighbors of start
                        for neighbor in neighbors:
                                path <- dfs on neighbor
                                if path is not empty:
```

return start + path

The algorithm returns either a non-empty list containing the indices of nodes on a path from the start to the end or an empty list.

Sequential Implementation

The input for our Haskell program is a text file representing a maze. An example input is given below:

```
##########
#s       #
# ###### #
# #    # #
# # ## # #
# # ## # #
#    #   #
###### ###
#      g #
##########
```

Figure 1: Sample Maze

In the input file, the # signs represent walls, spaces represent passages, “s” represents the starting point or entrance, and “g” represents the goal or exit. The file is read and converted to a two dimensional list through the following function:

```
readMaze :: FilePath -> IO [[Char]]
readMaze filename = do
    content <- readFile filename
    return (lines content)
```

Figure 2: Read Text file

In our DFS implementation, each character except "#" is treated as a vertex of a graph, and two vertices are neighbors if they are adjacent in the input. For each vertex that is not the goal, we first add it to the list of visited vertices. Then we find its neighbors by constructing a list of all possible positions and filter walls:

```
getNeighbors :: [[Char]] -> (Int, Int) -> [(Int, Int)]
getNeighbors maze (row, col) =
    let possibleMoves = [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]
        validMove (r, c) = r >= 0 && r < length maze && c >= 0 && c < length (maze !! 0)
                           && (maze !! r !! c == ' ' || maze !! r !! c == 'g')
    in filter validMove possibleMoves
```

Figure 3: Get Neighbor

This list has a fixed order of [above, below, left, right], and this is the searching order of our DFS. We remove visited vertices in the returned result and construct a list of all paths from unvisited neighbors to goal by the following line:

```
paths = [sequentialDFS maze n goal newVisited | n <- neighbors]
```

Finally, we return an empty list if no path is found or the first non-empty path otherwise.

Parallel DFS

A basic idea for parallelizing the DFS is to create a task for each neighbor of the currently searching vertex if there is more than one unvisited neighbor. There are two problems with this method. The first problem is that the algorithm will not stop until all tasks return. This means that the algorithm may keep running for a long time after the goal is found. The second

problem is that the algorithm may create unnecessarily many tasks since tasks do not share their visited nodes.

To solve the first problem, we use contexts to track the progress of each thread. Each context is a path from the starting vertex to a currently searching vertex. An example context would be: [(1, 1), (1, 2), (2, 2), (2, 3)]. The initial contexts is a list that contains exactly one context [start]. And it will dynamically expand or shrink as the program executes. The contexts allows the algorithm to interrupt the threads periodically and check whether the goal is reached. Hence allows the program to return whenever the goal is reached. In addition, with contexts, the algorithm is able to perform other global operations periodically. And this provides a solution to the second problem, which is to merge visited nodes periodically. Bellow is the main part of our parallel implementation:

```
parallelDFS :: [[Char]] -> [[Node]] -> Node -> Set Node -> Int -> Int -> IO [Node]
parallelDFS maze contexts goal visited threads maxDepth = do
  let results = parallelStep maze contexts goal visited maxDepth
  let newVisited = union visited $ unions [v | (_, v, _) <- results]
  let newContexts = filter (not . null) [context | (_, _, context) <- results]
  let assigned = fromList [c !! 0 | c <- newContexts]
  let updatedContexts = filter (not . null) $ parallelUpdate maze newContexts newVisited assigned threads
  case filter (\(found, _, _) -> found) results of
    [] -> do
      -- If no solution is found, recursively call parallelDFS with updated contexts
      if null updatedContexts
        then return [] -- No more nodes to explore
        else parallelDFS maze updatedContexts goal newVisited threads maxDepth
    (_, _, ancestors):_ -> return ancestors
```

Figure 4: Parallel code

The algorithm first runs DFS in parallel for each context. The DFS part is shown below:

```
step :: [[Char]] -> Node -> Set Node -> Int -> [Node] -> (Bool, Set Node, [Node])
step _ _ _ _ [] = (False, empty, []) -- This happens when a thread has searched a whole maze
step maze goal visited depth (start:ancestors)
  | start == goal = (True, visited, start:ancestors)
  | depth == 0 = (False, visited, start:ancestors)
  | start `member` visited = step maze goal visited (depth - 1) $ nextNode maze visited ancestors
  | otherwise =
      let newVisited = insert start visited
          newAncestors = nextNode maze newVisited (start:ancestors)
      in step maze goal newVisited (depth - 1) newAncestors
```

Figure 5: Parallel code

The function will return if either the context is empty, the goal is found, or it has reached a maximum depth. Otherwise, it will call the nextNode function to get the next node to be searched and recursively search that node. The code for nextNode is:

```
nextNode :: [[Char]] -> Set Node -> [Node] -> [Node]
nextNode _ _ [] = []
nextNode maze visited (parent:ancestors) =
  let neighbors = (getNeighbors maze parent \\ (toList visited))
  in if null neighbors
     then nextNode maze visited ancestors
     else (neighbors !! 0):parent:ancestors
```

Figure 6

This method will add the first unvisited neighbor to the context (ancestors). If all neighbors are visited, it recursively pops the parent from the context, trying to find an unvisited node. If all the parents in the context are completely searched, it will pop all elements and return an empty context.

To parallelize the DFS process, we use the parMap function from Control.Monad.Par to map the contexts into parallel tasks:

```
parallelStep :: [[Char]] -> [[Node]] -> Node -> Set Node -> Int -> [(Bool, Set Node, [Node])]
parallelStep maze contexts goal visited maxDepth =
  runPar $ parMap (\context -> step maze goal visited maxDepth context) contexts
```

Figure 7

After the parallel evaluation of DFS, the parallelStep will return information including a boolean variable indicating whether the goal is found, a set of new visited nodes, and updated contexts. The algorithm will then first check whether a thread found the goal, and returns its context as the path if it found the goal. Otherwise the algorithm will merge all visited nodes and then update all contexts using the updated set of visited nodes. The function for updating the context in parallel is shown below. It also uses parMap to create parallel tasks:

```
parallelUpdate :: [[Char]] -> [[Node]] -> Set Node -> Set Node -> Int -> [[Node]]
parallelUpdate maze contexts visited assigned threads =
  let filledContexts = [[] | _ <- [0 .. (threads - length contexts - 1)]] ++ contexts
  in runPar $ parMap (\i -> updateContext maze filledContexts visited assigned i) [0 .. (threads - 1)]
```

Figure 8

It will first fill the contexts list to its maximum length. This allows the actual updating function to assign new nodes to empty threads. In the function above, the empty contexts are placed at the start of the list. This is because the actual updating function assigns the ith unvisited node to the ith element in the contexts list if it is empty. The reason for this assignment is to avoid repeat assignments. And if the empty contexts are placed at the end, it is possible that they are not assigned new nodes even if there are unvisited nodes. The actual updating function are shown below:

```haskell
updateContext :: [[Char]] -> [[Node]] -> Set Node -> Set Node -> Int -> [Node]
updateContext maze contexts visited assigned thread
  | not (null (contexts !! thread)) = contexts !! thread
  | length allAvailable <= thread = []
  | otherwise = allAvailable !! thread
  where excludedNodes = union visited assigned
        available _ [] = [[]]
        available m (node:rest)
          | node `member` assigned = available m rest -- To make sure each thread searches a different path
          | otherwise = case (getNeighbors m node) \\ toList excludedNodes of
              [] -> available m rest
              neighbor:_ -> (neighbor:node:rest):(available m rest)
        allAvailable = concat [filter (not . null) (available maze context) | context <- contexts]
```

Figure 9

Given the thread index i, if the ith context is not empty, it does not update and return the original context. This allows the DFS to resume in the next recursion. If the ith context is empty, then the function finds a new unvisited node for it. Since the first node of each non-empty context has not been searched yet and therefore not in the visited set, we pass a separate variable "assigned" to this function to exclude assignments of these nodes. The function finds unvisited nodes by searching all nodes in each context.

Test

We tested two versions on four mazes. The Sequential version has the following running time:

maze 1: 0.35s

maze 2: 0.04s

maze 3: 0.34s

maze 4: 0.29s

For each maze, we tested our parallelization using max depth 3 and threads: 1, 2, 3, 4, 8. The results of them are:

```
maze 1:
n1, d3: 0.06s
n2, d3: 0.06s
n3, d3: 0.08s
n4, d3: 0.19s
n8, d3: 0.47s

maze 2:
n1, d3: 0.05s
n2, d3: 0.04s
n3, d3: 0.05s
n4, d3: 0.06s
n8, d3: 0.09s

maze 3:
n1, d3: 0.06s
n2, d3: 0.05s
n3, d3: 0.04s
n4, d3: 0.19s
n8, d3: 0.14s

maze 4:
n1, d3: 0.06s
n2, d3: 0.08s
n3, d3: 0.07s
n4, d3: 0.15s
n8, d3: 0.43s
```

Figure 10: Run results

In the above results, parallel versions are significantly faster than the sequential version when the number of threads is small. However, as the number of threads increases, the time is also increasing instead of decreasing.

Figure 11: Threadscope graph maze 3, threads 8, depth 3

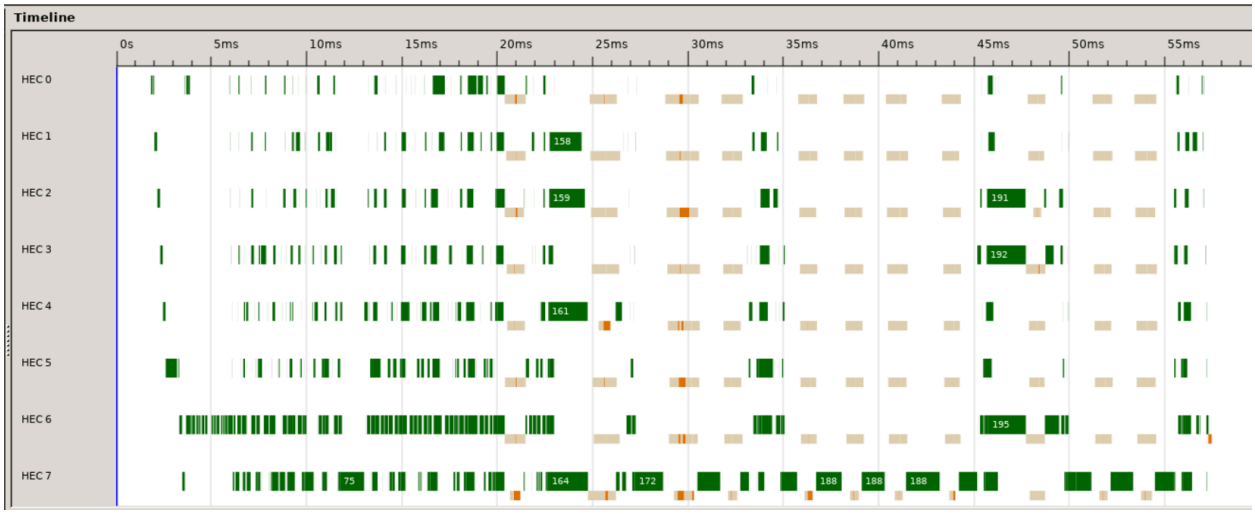

Figure 12 Threadscope Graph: maze 3, threads 8, depth 10

The ThreadScope graphs and data provided offer valuable insights into the performance of our Parallel DFS implementation. The analysis focuses on thread utilization, garbage collection, workload distribution, and the impact of parameters like thread count and depth. By exploring these elements, we can gain a comprehensive understanding of the algorithm's behavior and areas for improvement.

Figure 11:

The graph for Maze 3 with eight threads and a depth limit of three shows an active but uneven utilization of threads. Green bars indicate running threads, while orange bars represent garbage collection (GC). Although multiple threads are active, significant gaps are visible, implying idle periods where threads are not executing useful work. This suggests an imbalance in workload distribution. The frequent orange bars highlight that garbage collection is a major bottleneck in this configuration. High GC activity may stem from large, temporary data structures, such as the sets used for visited nodes or contexts, which are frequently created and discarded. This behavior

emphasizes the need for memory optimization, as excessive GC pauses reduce the potential benefits of parallel execution.

Figure 12:

Increasing the depth to 10 for the same maze results in a more balanced workload. The green bars in the graph are more uniform, indicating that threads are active for longer stretches without significant idle time. Moreover, orange bars are less frequent, suggesting a reduction in GC overhead. This improvement can be attributed to the increased depth, which allows threads to perform more work before synchronization is required. With deeper exploration per thread, the algorithm reduces the frequency of merging visited nodes, thereby minimizing redundant computations and unnecessary memory allocations. This configuration demonstrates the importance of depth tuning in achieving efficient parallel performance.

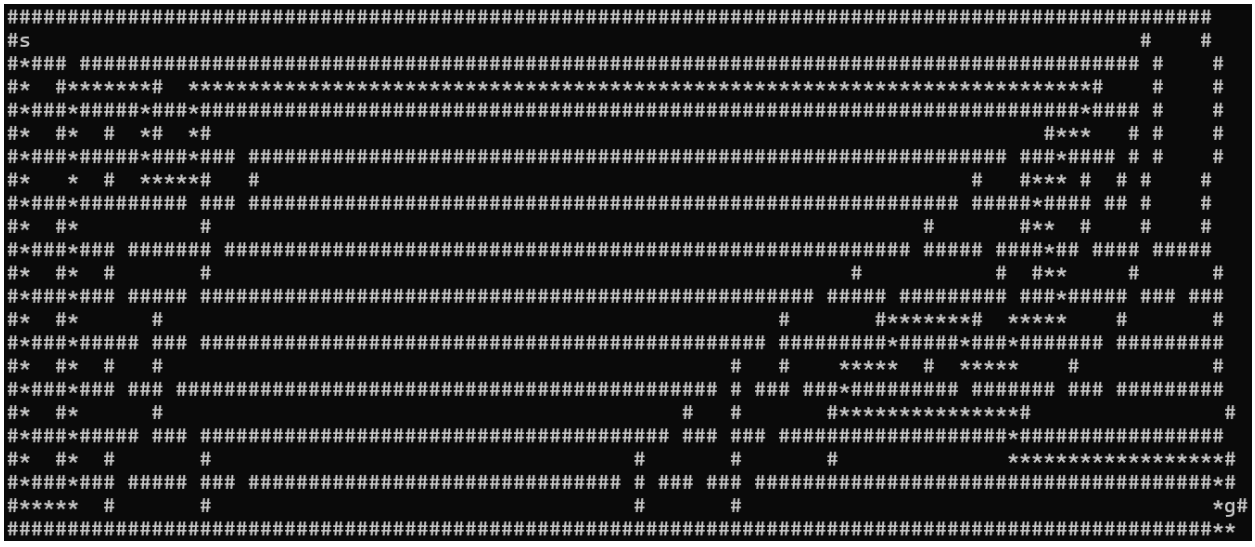

Figure 13: Solution for sequential

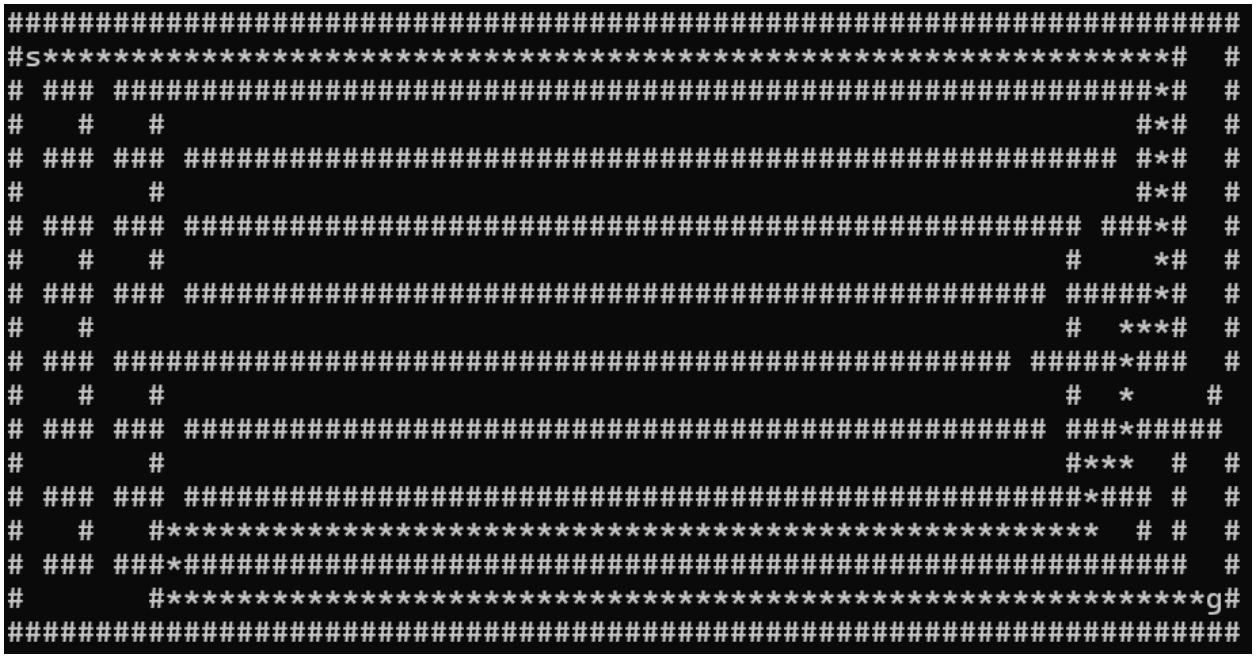

Figure 14: (Parallel solution of maze 1)

Sequential vs Parallel (Maze 1):

Comparing the sequential and parallel solutions for Maze 1 reveals the advantages of parallelism. The sequential graph shows a single-threaded execution with no parallel activity. The path found by the sequential algorithm is longer, as it searches exhaustively in one direction before exploring alternatives. In contrast, the parallel solution utilizes multiple threads to explore different paths simultaneously, leading to the discovery of a shorter path. The shorter path directly impacts the performance, as reaching the goal earlier reduces the overall search space. This difference highlights a key benefit of parallel DFS: its ability to achieve faster convergence by leveraging simultaneous exploration.

```
n8, d2: 0.53s
n8, d4: 0.31s
n8, d6: 0.2s
n8, d10: 0.24s
```

Figure 15: (maze 1, 8 threads, different depth)

```
n8, d2: 0.2s
n8, d4: 0.13s
n8, d6: 0.11s
```

Figure 16: (maze 2, 8 threads, different depth)

```
n8, d2: 0.29s
n8, d4: 0.19s
n8, d6: 0.11s
n8, d10: 0.06s
```

Figure 17: (maze 3m 8 threads, different depth)

```
n3, d2: 0.19s
n3, d4: 0.07s
n3, d6: 0.051s
n3, d10: 0.04s
```

Figure 18: (maze 4, 3 threads, different depth)

Effects of Depth on Parallel Performance:

Depth plays a crucial role in determining the efficiency of the parallel DFS algorithm. For Maze 1 with eight threads, increasing the depth initially reduces the execution time as threads perform more work independently, reducing the need for frequent synchronization. However, the

performance gains diminish beyond a certain depth. Excessive depth can lead to redundant work, where threads explore parts of the maze that other threads have already visited, wasting computational resources. A similar pattern is observed in Maze 4 with three threads. Here, the lower thread count underutilized the available cores, but varying the depth still highlights the trade-offs. Shallow depths increase synchronization overhead, while excessive depths introduce redundancy. The results emphasize the need to balance depth to optimize thread utilization and reduce redundant work.

Speedup and Bottlenecks:

The speedup achieved by the parallel DFS is significant when the number of threads is small, as it efficiently divides the workload among available threads. However, as the thread count increases, the speedup plateaus or even declines. This behavior can be attributed to threading overhead, including task creation, synchronization, and workload management. With more threads, the cost of coordinating their execution begins to outweigh the benefits of parallelism. Additionally, workload distribution becomes increasingly challenging, leading to idle threads that contribute little to overall performance.

Garbage collection is another major bottleneck in the current implementation. Frequent GC pauses, visible as orange bars in the ThreadScope graphs, interrupt execution and degrade performance. These pauses are likely caused by the creation of intermediate data structures, such as sets for visited nodes and contexts. Optimizing memory usage by employing strict data structures and reusing memory can significantly reduce GC overhead.

Discussion and Future Directions:

To enhance the performance of the parallel DFS algorithm, several strategies can be considered. First, dynamic workload balancing techniques, such as work-stealing, can help distribute tasks more evenly among threads, reducing idle time. Second, optimizing memory usage by employing strict data structures or preallocating memory can mitigate garbage collection overhead. Third, adaptive depth tuning, where each thread dynamically adjusts its depth based on workload, can strike a balance between synchronization frequency and redundancy. Finally, testing on larger and more complex mazes will provide a better understanding of the algorithm's scalability and its ability to handle real-world scenarios**.**

Conclusion:

The exploration of sequential and parallel implementations of Depth-First Search (DFS) in this report highlights the strengths and challenges of leveraging parallelism for graph search algorithms. The sequential implementation, while straightforward and effective for smaller problem sizes, struggles to scale efficiently with increasing maze complexity. In contrast, the parallel DFS demonstrates significant improvements in execution time, especially for configurations with smaller thread counts and optimized depth settings.

The analysis of ThreadScope graphs revealed critical insights into the behavior of the parallel algorithm. With adequate depth limits, parallel DFS effectively utilizes multiple threads to explore different paths simultaneously, resulting in shorter search times and faster convergence to the goal. However, the performance gains diminish with higher thread counts due to threading overhead, imbalanced workload distribution, and frequent garbage collection pauses. The data also emphasizes the importance of memory management and workload balancing in achieving

optimal performance. Parameters like thread count and maximum depth play a pivotal role in determining efficiency, requiring careful tuning based on the problem characteristics.

While the results demonstrate the potential of parallel DFS for solving complex mazes, there is room for further optimization. Techniques such as adaptive depth management, dynamic workload balancing, and stricter memory usage can help reduce overhead and improve scalability. Additionally, testing on larger and more complex mazes would provide deeper insights into the algorithm's performance in real-world scenarios.

This study underscores the power of parallelism in transforming traditional algorithms like DFS, enabling them to tackle problems more efficiently. However, it also highlights the delicate balance required between computation, synchronization, and memory management to fully harness the advantages of multi-threaded execution.

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