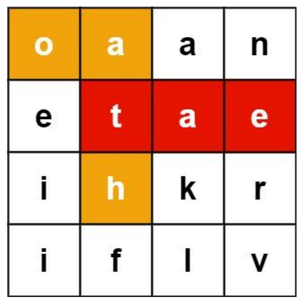
# Parallel Word Search in Haskell

COMS 4995: Parallel Functional Programming

Helen Chu (hc2932), Alexander Lindenbaum (al4008)

## Introduction

For the project, we aimed to implement a parallelized solution to the word search problem: given a board of characters and a dictionary of words, return all words on the board. We define a valid word as being constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring; the same letter cell may not be used more than once in a word. For example, the following board[1] would contain the words "oath" and "eat", as indicated by the highlights:




The word search problem has a naive solution, where at each cell of the grid, you initiate a depth-first search and search for the target words. A worst-case analysis of this algorithm gives an O($n^2 \cdot 4^n$) runtime complexity, where the grid is $n \times n$. On average however, most calls to DFS should be allowed to halt immediately, as most strings will not match to a prefix of some target word. A variation of the word search problem is to use an established dictionary as the list of target words, i.e. search for any valid words in the grid. In this variation, the average case matches the worst case runtime.

We will use cell-by-cell depth-first search combined with the usage of a Prefix Tree, also known as a Trie, to approach this problem.

[1] *Word search II*. LeetCode. (n.d.). Retrieved December 20, 2022, from https://leetcode.com/problems/word-search-ii/

## Sequential Implementation

To allow for easy access to specific grids in the board, we first parse and store the board as a Map where the key is the coordinate and the value is the Char value inside the grid. Then, we take the target word list and store them in a Trie. During DFS, we store the current word formed by our path and the "subtrie" corresponding to our position in the trie, and store any valid words discovered along the way.

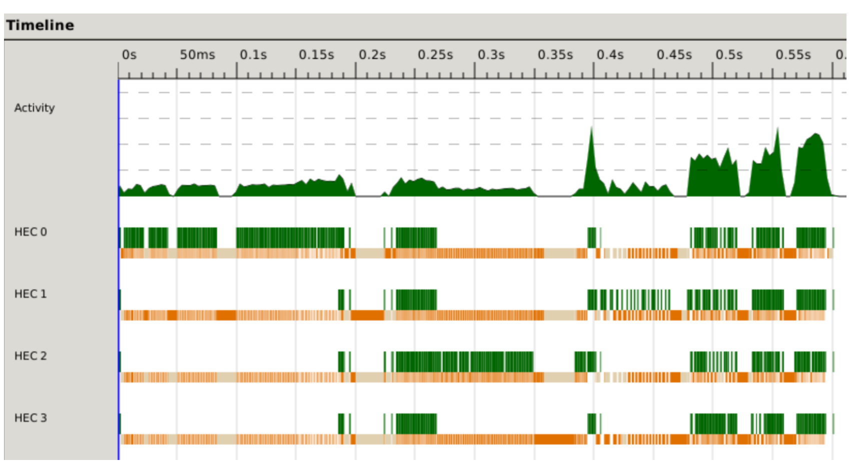
```
main :: IO()
main = do
  args <- getArgs
  filename <- case args of
    [filename] -> return filename
    _ -> do
      pn <- getProgName
      die $ "Usage: " ++ pn ++ " <filename>"
  puzzle <- readFile filename
  let (p, w) = parsePuzzle puzzle
      trie = mkTrie w
      f index _ wordList = wordList ++ dfs p trie index [] ""
      output = Map.foldrWithKey f [] p
  mapM_ putStrLn output
```

## Parallelization & Results

We aimed to parallelize running DFS starting from different grids on the board. Instead of using the naive "foldrWithKey" method, we planned to utilize parMap to allow for dynamic partitioning along with parallelization. The relevant code snippet is as follows:

```
let (p, w) = parsePuzzle puzzle
    trie = mkTrie w
    output = runEval $ do
      let result = parMap rdeepseq (\(index, _) -> dfs p trie index [] "") (Map.toList p)
      _ <- rseq result
      return result
```

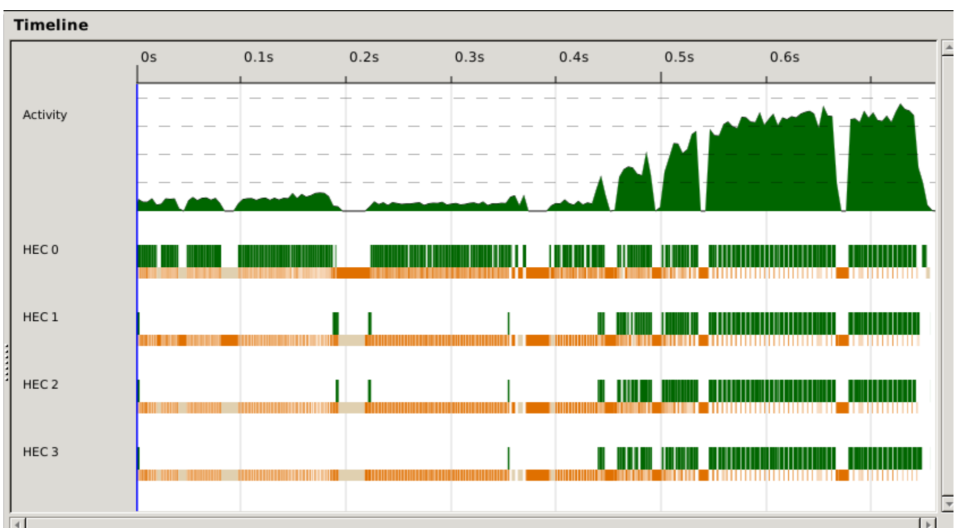
However, with the above implementation, we were consistently observing a longer runtime while running on multiple cores compared to a single core (Table 2). Upon running threadscope, we notice much time is wasted on garbage collection.



To isolate the issue, we further attempted to implement static partitioning, where we split the board into four segments and run DFS on each segment in parallel using four cores. The relevant code snippet is as follows:

```
let (p, w) = parsePuzzle puzzle
    trie = mkTrie w
    dfsWrapper (index, _) = dfs p trie index [] ""
    (q1, q2, q3, q4) = splitInQuarter (Map.toList p)
    (p1, p2, p3, p4) = runEval $ do
      q1Result <- rpar (force (map dfsWrapper q1))
      q2Result <- rpar (force (map dfsWrapper q2))
      q3Result <- rpar (force (map dfsWrapper q3))
      q4Result <- rpar (force (map dfsWrapper q4))
      _ <- rseq q1Result
      _ <- rseq q2Result
      _ <- rseq q3Result
      _ <- rseq q4Result
      return (q1Result, q2Result, q3Result, q4Result)
```

Still, we observed an increase in runtime when using 4 cores compared to using one core (Table 1). The Threadscope output is as follows:




The first half of the timeline seem to be the sequential formation of the coordinate mapping as well as the trie. Finally, we attempted to regulate the number of outstanding sparks by utilizing parBuffer; the relevant code snippet is as follows:

```
let (p, w) = parsePuzzle puzzle
    trie = mkTrie w
    output = runEval $ do
      let result = (withStrategy (parBuffer 16 rpar) . map (\(index, _) -> dfs p trie index [] "")) (Map.toList p)
      _ <- rseq result
      return result
```

Table 1: Splitting Grid Into Quarters with rpar and rseqs:

| Size of Input Grid | Average Total Time (1 Core) | Average Total Time (4 Cores) | Average Mutator Time (4 Cores) | Average GC Time (4 Cores) |
|---|---|---|---|---|
| 100 x 100 | 0.009s | 0.011s | 0.007s | 0.004s |
| 200 x 200 | 0.161s | 0.189s | 0.123s | 0.066s |
| 300 x 300 | 1.107s | 1.294s | 0.526s | 0.768s |

Table 2: Using parMap rdeepseq:

| Size of Input Grid | Average Total Time (1 Core) | Average Total Time (4 Cores) | Average Mutator Time (4 Cores) | Average GC Time (4 Cores) |
|---|---|---|---|---|
| 100 x 100 | 0.009s | 0.016s | 0.012s | 0.004s |
| 200 x 200 | 0.161s | 0.194s | 0.128s | 0.066s |
| 300 x 300 | 1.134s | 1.339s | 0.525s | 0.814s |

Table 3: Using parBuffer 16:

| Size of Input Grid | Average Total Time (4 Cores) | Average Mutator Time (4 Cores) | Average GC Time (4 Cores) |
|---|---|---|---|
| 100 x 100 | 0.019s | 0.015s | 0.003s |
| 200 x 200 | 0.205s | 0.137s | 0.067s |
| 300 x 300 | 1.356s | 0.545s | 0.811s |

## Code Listings

Input Parsing:

```haskell
module WordSearch.Tools
  (
    parsePuzzle
  , GridMap
  , Coord
  , splitInQuarter
  ) where


import Data.Map (Map, insert, empty)

type Coord = (Int, Int)
type GridMap = Map Coord Char


splitInQuarter :: [a] -> ([a], [a], [a], [a])
splitInQuarter l = (a,b,c,d)
  where (left, right) = splitInHalf l
        (a,b) = splitInHalf left
        (c,d) = splitInHalf right


splitInHalf :: [a] -> ([a], [a])
splitInHalf l = splitInHalfHelper l [] [] 0


splitInHalfHelper :: [a] -> [a] -> [a] -> Int -> ([a], [a])
splitInHalfHelper [] l r _ = (l, r)
splitInHalfHelper (x:xs) l r parity
  | parity == 0   = splitInHalfHelper xs (x:l) r 1
  | otherwise     = splitInHalfHelper xs l (x:r) 0


parsePuzzle :: String -> (GridMap, [String])
parsePuzzle p = (grid, wordList)
    where linesP = lines p
          grid = parseGrid gridStrings 0 empty
          gridStrings = take (length linesP - 2) linesP
          wordList = words . last $ linesP


parseGrid :: [String] -> Int -> GridMap -> GridMap
parseGrid [] _ grid = grid
parseGrid (x:xs) row grid = parseGrid xs (row + 1) (parseRow x (row, 0) grid)


parseRow :: String -> Coord -> GridMap -> GridMap
parseRow [] _ grid = grid
parseRow (x:xs) (row, col) grid = parseRow xs (row, col + 1) (insert (row, col) x grid)
```

Trie:

```haskell
{-
Implements a Trie; credit: http://mchaver.com/posts/2018-12-27-tries-in-haskell.html
-}

module WordSearch.Trie
  (
    mkTrie
  , Trie
  , getTrie
  , empty
  ) where


import qualified Data.Map.Lazy as Map
import Data.Maybe (fromMaybe)

data Trie a = Trie Bool (Map.Map a (Trie a)) deriving (Eq, Read, Show)

empty :: Trie a
empty = Trie False Map.empty

getTrie :: Trie a -> (Bool, Map.Map a (Trie a))
getTrie (Trie end nodes) = (end, nodes)

insert :: Ord a => [a] -> Trie a -> Trie a
insert []     (Trie _ nodes)     = Trie True nodes
insert (x:xs) (Trie end nodes) = Trie end (Map.alter (Just . insert xs . fromMaybe empty) x nodes)

-- Takes a list of words, makes into a trie
mkTrie :: Ord a => [[a]] -> Trie a
mkTrie as = mkTrie' as empty
  where
    mkTrie' []     trie = trie
    mkTrie' (x:xs) trie = mkTrie' xs $ insert x trie
```

DFS:

```haskell
module WordSearch.DFS
  (
    dfs
  ) where

import WordSearch.Trie(Trie, getTrie, empty)
import WordSearch.Tools(GridMap, Coord)
import Data.Map(member, lookup)
import Data.Maybe ( fromMaybe )

dfs :: GridMap -> Trie Char -> Coord -> [Coord] -> String -> [String]
dfs p trie index@(row, col) visited word =
  case Data.Map.lookup index p of
    Nothing -> []
    Just c  | index `elem` visited || not (member c children) -> []
            | isWord                                          -> newWord : recurse
            | otherwise                                       -> recurse
            where
              (_, children) = getTrie trie
              (isWord, _)   = getTrie child
              newWord       = word++[c]
              child         = fromMaybe empty (Data.Map.lookup c children)
              trav ind      = dfs p child ind (index:visited) newWord
              recurse       = trav (row, col-1) ++ trav (row, col+1) ++ trav (row-1, col) ++ trav (row+1, col)
```

Main:

```haskell
{-# OPTIONS_GHC -Wall #-}
module Main (main) where

import Lib (parsePuzzle, splitInQuarter, mkTrie, dfs)
import Data.List(nub)
import System.Environment(getArgs, getProgName)
import System.Exit(die);
import Data.List(union, foldl')
import Control.Parallel.Strategies
import qualified Data.Map as Map
import Control.DeepSeq

main :: IO()
main = do
  args <- getArgs
  filename <- case args of
    [filename] -> return filename
    _ -> do
      pn <- getProgName
      die $ "Usage: " ++ pn ++ " <filename>"
  puzzle <- readFile filename
  let (p, w) = parsePuzzle puzzle
      trie = mkTrie w
      output = runEval $ do
        let result = parMap rdeepseq (\(index, _) -> dfs p trie index [] "") (Map.toList p)
        -- Alternative attempt using parBuffer:
        -- let result = (withStrategy (parBuffer 16 rpar) . map (\(index, _) -> dfs p trie index [] "")) (Map.toList p)
        _ <- rseq result
        return result
  print $ foldl' union [] output

  -- Alternative attempt using static partitioning into 4 segments:
  -- let (p, w) = parsePuzzle puzzle
  --     trie = mkTrie w
  --     dfsWrapper (index, _) = dfs p trie index [] ""
  --     (q1, q2, q3, q4) = splitInQuarter (Map.toList p)
  --     (p1, p2, p3, p4) = runEval $ do
  --       q1Result <- rpar (force (map dfsWrapper q1))
  --       q2Result <- rpar (force (map dfsWrapper q2))
  --       q3Result <- rpar (force (map dfsWrapper q3))
  --       q4Result <- rpar (force (map dfsWrapper q4))
  --       _ <- rseq q1Result
  --       _ <- rseq q2Result
  --       _ <- rseq q3Result
  --       _ <- rseq q4Result
  --       return (q1Result, q2Result, q3Result, q4Result)
  -- -- print $ nub $ concat (p1 ++ p2 ++ p3 ++ p4)
```