COMS4995 PFP Final Project Report: Word Hunt Solver

Allison Liu (al4130)

Fall 2022

1 Background

Word Hunt is a game where a player is given a 4x4 board of tiles corresponding to letters, and the goal is to create as many words as possible from that board. The restrictions are that consecutive letters in the word must be adjacent to each other on the board (right, left, up, down, diagonal), and the player may not use the same tile twice in the same word. Each word must be at least 3 letters long. A board of $n \times n$ would have a maximum of $(n^2) \times (n^2)!$ possible sequences. The solver could blow up very easily since the basic version of Word Hunt is 4×4 board, which would result in trillions of paths. This project's goal is to find all possible words following these restrictions given a Word Hunt board.

For reference, the machine that I am running this on is a Quad-Core Intel i5 processor that can run eight threads.

2 General Approach

Given a board and a dictionary file, the general approach to solve this game is to first, read in a dictionary file. To find all the potential words, start at a given tile and do a depth first search in accordance to the restrictions of the game, accumulating letters for a potential word. At every depth, the search should go toward a neighbor that hasn't yet been visited. Every potential word should be checked against the dictionary file to verify its validity as a word. If it is a word, add it to the output list. After going through each tile and its respective paths, print out the output list, which is the list of all possible words that the board can create.

In my approach, I accepted the input arguments of the file path of a dictionary file, a board (a string of 16 characters), and the dimension of the board. My tests will maintain the dimension of the board at 4 to be consistent with the rules of Word Hunt. With these inputs, I had two major data structures. The first is the Board, which is a list of lists of Chars, basically a 2-D array of characters. The second is a dictionary as a Set of Strings.

3 Sequential

The sequential implementation is basically encompassed in the function `seqWordHuntSolver`. This function uses list comprehension to collect all potential words and filter through them based on if they are a member of the dictionary and if they are of length greater than two. `indices` are the coordinates of all the tiles on the board. `findWords` is the DFS portion of the code; it has a base case, keeping the calls within the bounds of the board, and making sure we don't visit the same tile twice in one word. Then, it has eight recursive calls for all of the tile's neighbors.

```haskell
seqWordHuntSolver :: Board -> Set String -> [String]
seqWordHuntSolver board dict =
  [word | (x,y) <- indices,
          word <- findWords (x,y) [] "",
          word `member` dict,
          length word > 2]
  where
    indices = [(x,y) | x <- [0..(length board - 1)],
                       y <- [0..(length (head board) - 1)]]

    findWords :: (Int, Int) -> [(Int, Int)] -> String -> [String]
    findWords (x,y) visited word
      | x < 0 || y < 0 || x >= length board || y >= length (head board) || (x,y) `elem` visited = [] -- base case
      | otherwise =
          let newWord = word ++ [board !! x !! y]
              newVisited = (x,y) : visited
          in  [newWord] ++
              (findWords (x-1,y-1) newVisited newWord ++
               findWords (x-1,y)   newVisited newWord ++
               findWords (x-1,y+1) newVisited newWord ++
               findWords (x,y-1)   newVisited newWord ++
               findWords (x,y+1)   newVisited newWord ++
               findWords (x+1,y-1) newVisited newWord ++
               findWords (x+1,y)   newVisited newWord ++
               findWords (x+1,y+1) newVisited newWord)
```

4 Parallelization

4.1 Parallelizing DFS

I parallelized the depth first search by making the depth first search on each tile run in parallel in `wordHuntSolver`, which would create 16 sparks (one spark per tile). This implementation is very similar to the sequential implementation with key differences on lines 37 and 38. My initial attempt at parallelizing used `parBuffer` and `rpar`. This resulted in 32 sparks with 17 fizzling. I then switched to `rseq` instead, and while the sparks were more efficient (only 16 created and 1 fizzled), the program almost exclusively ran on one thread. I initially thought that the issue was due to the fact that my parallelization only had 16 sparks, but after speaking with Professor Edwards, he diagnosed the issue to be more about using functions intended for normal form instead of weak head normal form. Now, with `rdeepseq`, my program is sufficiently parallel, still with 16 sparks. The results are displayed in the Testing and Results section. Running the program on one core (Figure 2) took 21.47 seconds. Running it on two and four cores (Figures 4 and 6) took 17.78 and 13.47 seconds respectively, so as it becomes more parallel, the efficiency of the program increases. After 4 cores, the time actually increases again, so 4 cores ended up being optimal.

```haskell
wordHuntSolver :: Board -> Set String -> [String]
wordHuntSolver board dict =
  [word | word <- concat allWords,
          word `member` dict,
          length word > 2]
  where
    -- Find all the indices (row and column) of the squares on the board
    indices = [(x,y) | x <- [0..(length board - 1)],
                       y <- [0..(length (head board) - 1)]]

    parFindWords = findWords [] ""
    allWords = Prelude.map parFindWords indices `using` parBuffer 2 rdeepseq  -- reduced to weak head normal form, rpar, deepseq instead
    -- allWords' = rdeepseq allWords

    -- Find all the words that can be formed starting at a given square
    -- and following a path of adjacent squares
    findWords :: [(Int, Int)] -> String -> (Int, Int) -> [String]
    findWords visited word (x,y)
      -- If the current square is out of bounds or has already been visited,
      -- there are no more words to be found
      | x < 0 || y < 0 || x >= length board || y >= length (head board) || (x,y) `elem` visited = []
      -- Otherwise, add the current square to the visited squares, add its
      -- character to the current word, and search for more words in all
      -- the adjacent squares
      | otherwise =
          let newWord = word ++ [board !! x !! y]
              newVisited = (x,y) : visited
          in  [newWord] ++
              (findWords newVisited newWord (x-1,y-1) ++
               findWords newVisited newWord (x-1,y)   ++
               findWords newVisited newWord (x-1,y+1) ++
               findWords newVisited newWord (x,y-1)   ++
               findWords newVisited newWord (x,y+1)   ++
               findWords newVisited newWord (x+1,y-1) ++
               findWords newVisited newWord (x+1,y)   ++
               findWords newVisited newWord (x+1,y+1))
```

4.2 In the Future

The plan before beginning this project was to also implement another parallelization to this algorithm. That parallelization was to read the dictionary in chunks, and parallelize checking the potential words against the dictionary. Due to the issues that I was having with the parallelization of the DFS, I didn't get to finish implementing this portion of the project but will continue working on it in the future!

5 Testing and Results

Both results are tested on the input board: oatrihpshtnrenei, which is equivalent to the board in Figure 1. The input dictionary was downloaded from `https://raw.githubusercontent.com/eneko/data-repository/master/data/words.txt`.

Figure 1: Input board

```
  26,768,614,976 bytes allocated in the heap
   8,283,549,056 bytes copied during GC
     357,339,320 bytes maximum residency (29 sample(s))
       3,324,000 bytes maximum slop
             825 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     25728 colls,     0 par    4.171s   4.386s     0.0002s    0.0072s
  Gen  1        29 colls,     0 par    3.346s   3.925s     0.1353s    0.3726s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 16 (4 converted, 0 overflowed, 0 dud, 0 GC'd, 12 fizzled)

  INIT    time    0.000s  (  0.005s elapsed)
  MUT     time   12.729s  ( 13.093s elapsed)
  GC      time    7.517s  (  8.311s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time   20.246s  ( 21.409s elapsed)

  Alloc rate    2,103,029,829 bytes per MUT second

  Productivity  62.9% of total user, 61.2% of total elapsed


real    0m21.471s
user    0m20.249s
sys     0m0.791s
```

Figure 2: Stats of 1 core

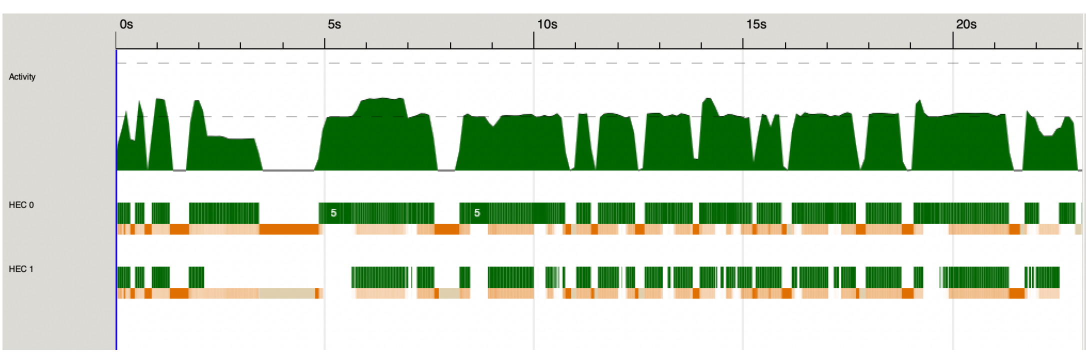
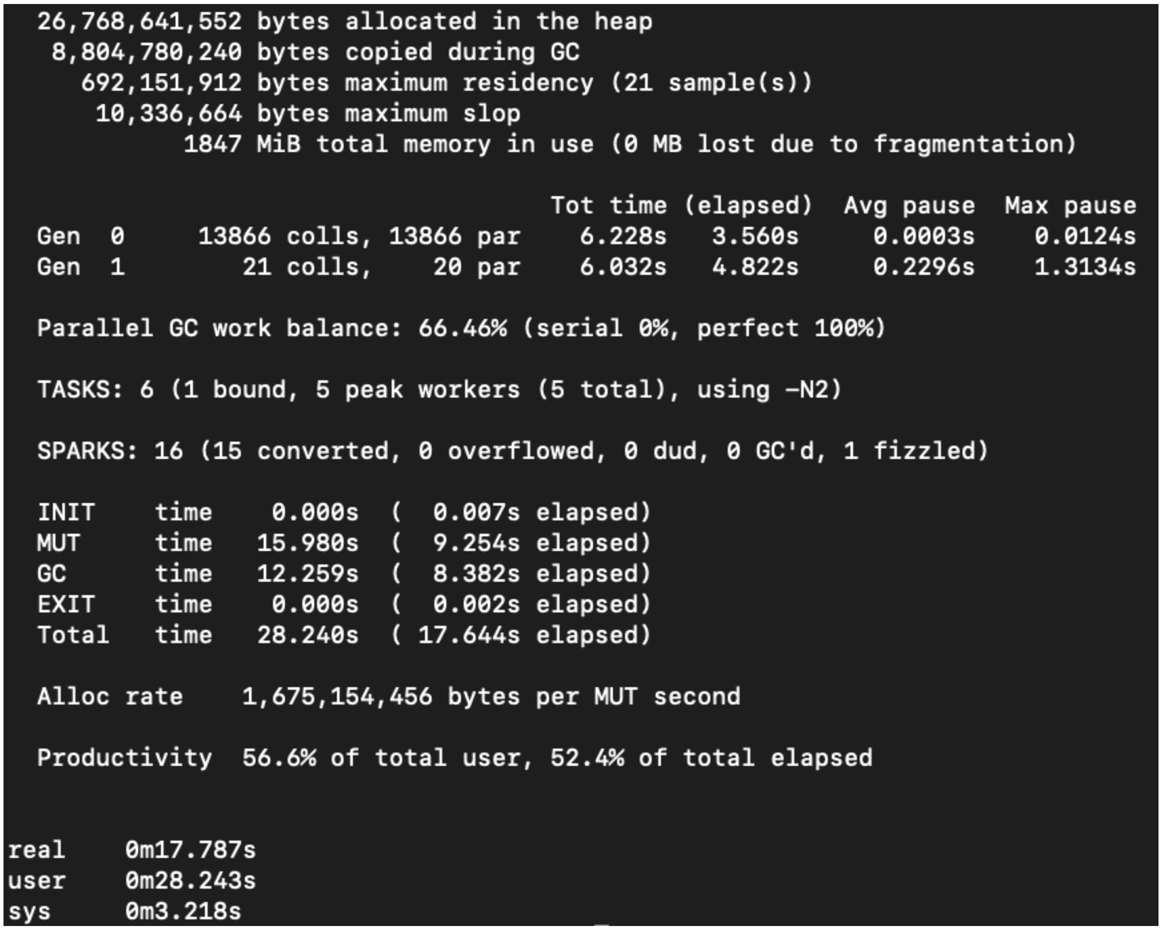


Figure 3: Threadscope of 2 cores



```
  26,768,641,552 bytes allocated in the heap
   8,804,780,240 bytes copied during GC
     692,151,912 bytes maximum residency (21 sample(s))
      10,336,664 bytes maximum slop
            1847 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     13866 colls, 13866 par    6.228s   3.560s     0.0003s    0.0124s
  Gen  1        21 colls,    20 par    6.032s   4.822s     0.2296s    1.3134s

  Parallel GC work balance: 66.46% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)

  SPARKS: 16 (15 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.000s  (  0.007s elapsed)
  MUT     time   15.980s  (  9.254s elapsed)
  GC      time   12.259s  (  8.382s elapsed)
  EXIT    time    0.000s  (  0.002s elapsed)
  Total   time   28.240s  ( 17.644s elapsed)

  Alloc rate    1,675,154,456 bytes per MUT second

  Productivity  56.6% of total user, 52.4% of total elapsed


real    0m17.787s
user    0m28.243s
sys     0m3.218s
```

Figure 4: Stats of 2 cores

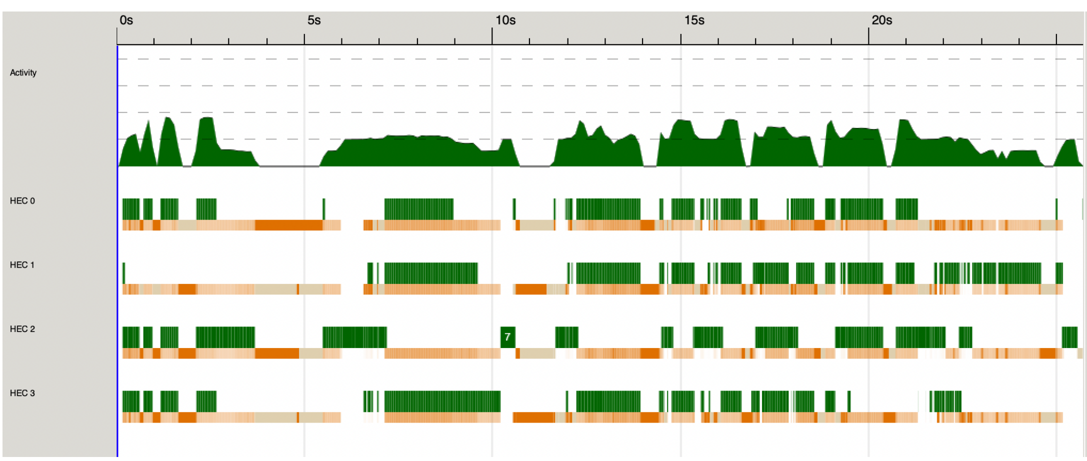


Figure 5: Threadscope of 4 cores

```
  26,768,685,344 bytes allocated in the heap
   8,457,272,760 bytes copied during GC
     666,808,680 bytes maximum residency (18 sample(s))
       9,735,832 bytes maximum slop
            1549 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     11850 colls, 11850 par   13.872s   2.573s     0.0002s    0.0043s
  Gen  1        18 colls,    17 par    7.655s   2.580s     0.1433s    0.3937s

  Parallel GC work balance: 60.24% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 16 (15 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.000s  (  0.005s elapsed)
  MUT     time   15.542s  (  8.175s elapsed)
  GC      time   21.527s  (  5.153s elapsed)
  EXIT    time    0.000s  (  0.007s elapsed)
  Total   time   37.070s  ( 13.340s elapsed)

  Alloc rate    1,722,304,399 bytes per MUT second

  Productivity  41.9% of total user, 61.3% of total elapsed


real    0m13.472s
user    0m37.072s
sys     0m3.451s
```

Figure 6: Stats of 4 cores

6 Code

```haskell

import System.IO as Sys

import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Control.Parallel.Strategies (parBuffer, using, rseq, rpar, parList,
    rdeepseq, rparWith)
import Data.Char
import System.Posix.IO
import System.Posix.Types
import Data.Set

-- A board is represented as a list of lists of characters
type Board = [[Char]]

makeBoard :: String -> Int -> Board
makeBoard [] _ = []
makeBoard input dim = Prelude.take dim input : makeBoard (Prelude.drop dim
    input) dim

-- | Sequential DFS
seqWordHuntSolver :: Board -> Set String -> [String]
seqWordHuntSolver board dict =
  [word | (x,y) <- indices,
          word <- findWords (x,y) [] "",
          word `member` dict,
          length word > 2]
  where
    indices = [(x,y) | x <- [0..(length board - 1)],
                       y <- [0..(length (head board) - 1)]]

    findWords :: (Int, Int) -> [(Int, Int)] -> String -> [String]
    findWords (x,y) visited word
      | x < 0 || y < 0 || x >= length board || y >= length (head board) ||
    (x,y) `elem` visited = [] -- base case
      | otherwise =
          let newWord = word ++ [board !! x !! y]
              newVisited = (x,y) : visited
          in  [newWord] ++
              (findWords (x-1,y-1) newVisited newWord ++
               findWords (x-1,y)   newVisited newWord ++
               findWords (x-1,y+1) newVisited newWord ++
               findWords (x,y-1)   newVisited newWord ++
               findWords (x,y+1)   newVisited newWord ++
               findWords (x+1,y-1) newVisited newWord ++
               findWords (x+1,y)   newVisited newWord ++
               findWords (x+1,y+1) newVisited newWord)


-- | Parallel DFS
wordHuntSolver :: Board -> Set String -> [String]
wordHuntSolver board dict =
```

```
  [word | word <- concat allWords,
          word `member` dict,
          length word > 2]
  where
    -- Find all the indices (row and column) of the squares on the board
    indices = [(x,y) | x <- [0..(length board - 1)],
                       y <- [0..(length (head board) - 1)]]

    parFindWords = findWords [] ""
    allWords = Prelude.map parFindWords indices `using` parBuffer 2 rdeepseq

    -- Find all the words that can be formed starting at a given square
    -- and following a path of adjacent squares
    findWords :: [(Int, Int)] -> String -> (Int, Int) -> [String]
    findWords visited word (x,y)
      -- If the current square is out of bounds or has already been visited,
      -- there are no more words to be found
      | x < 0 || y < 0 || x >= length board || y >= length (head board) || (x,y) `elem` visited = []
      -- Otherwise, add the current square to the visited squares, add its
      -- character to the current word, and search for more words in all
      -- the adjacent squares
      | otherwise =
          let newWord = word ++ [board !! x !! y]
              newVisited = (x,y) : visited
          in  [newWord] ++
              (findWords newVisited newWord (x-1,y-1) ++
               findWords newVisited newWord (x-1,y)   ++
               findWords newVisited newWord (x-1,y+1) ++
               findWords newVisited newWord (x,y-1)   ++
               findWords newVisited newWord (x,y+1)   ++
               findWords newVisited newWord (x+1,y-1) ++
               findWords newVisited newWord (x+1,y)   ++
               findWords newVisited newWord (x+1,y+1))


readDictionary :: FilePath -> IO [String]
readDictionary path = lines <$> readFile path

parseDict :: [[Char]] -> Set [Char]
parseDict dictionary = Data.Set.fromList (Prelude.map (Prelude.map toLower) dictionary)

main :: IO ()
main = do
         args <- getArgs
         case args of
            [dict, board, dim] -> do
               dictionary <- readDictionary dict
               let parsed =  parseDict dictionary
                   solved = wordHuntSolver (makeBoard board (read dim :: Int)) parsed
```

```haskell
                mapM_ putStrLn solved
             _ ->
                do pn <- getProgName
                   die $ "Usage: " ++pn++ " <dictionary-filename> <board> <dimension>"


-- | Testing just DFS
testDFS :: IO ()
testDFS = do
  let board = [
          "abcd",
          "efgh",
          "ijkl"
        ]

      dict = fromList ["a", "bef", "abe", "fgk", "jie", "goodness", "kgfb", "efg", "hello", "fkplhg"]
      -- expected = ["abe", "bef", "efg", "fgk", "jie", "kgfb", "hello"]
      expected = ["a", "bef", "abe", "fgk", "jie", "goodness", "kgfb", "efg", "hello", "fkplhg"]

  -- Check that the wordHuntSolver function returns the expected result
  assertEqual (wordHuntSolver board dict) expected

-- Assert that two values are equal
assertEqual :: (Eq a, Show a) => a -> a -> IO ()
assertEqual x y
  | x == y    = return ()
  | otherwise = error (show x ++ " /= " ++ show y)



-- | My attempt at the second parallelization
readChunks :: Fd -> IO [String]
readChunks fd = do
  -- fileSize <- hFileSize dict
  -- let fileMode   = Just (CMode 0440)
  --     part = fileSize `div` 8
  --     part' = part * parts
  --     size = if parts == (totalParts-1)
  --                 then part
  --                 else part +
  chunk <- fdRead fd 4096
  done <- isEOF
  -- if fst chunk == ""
  if done
    then return []
    else do
      rest <- readChunks fd
      -- putStrLn (fst chunk)
      return ( fst chunk : rest)
```