Trie Autocomplete

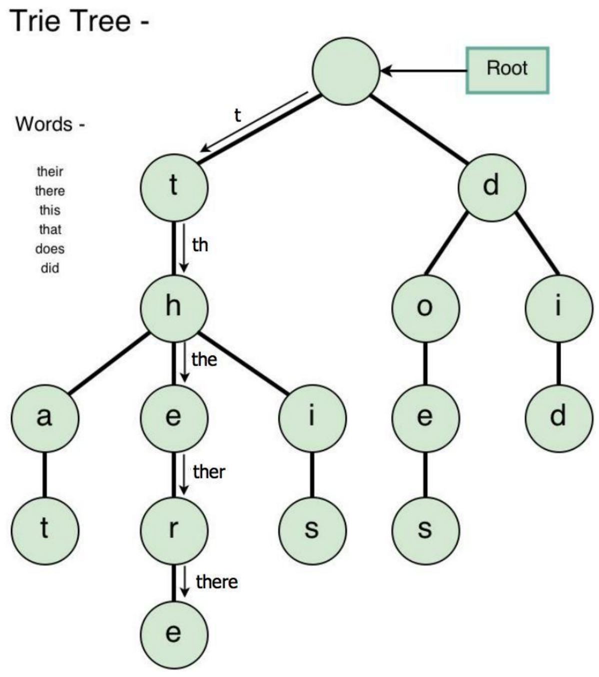
Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

I. Introduction

Our project is a word autocomplete feature, similar to what is commonly seen on search engines, where a word is partially typed and the suggestions are generated to complete the word. For example, typing "do" into Google may generate suggestions like "dog", "dominos", "docs", etc. The project takes in a file containing a cleaned corpus of words, in this case we use a dataset of all papers submitted to the 2015 NeurIPS conference [1]. The program then asks the user to enter a search term and returns the 10 most probable words, given the search string.

II. Trie

To perform this autocomplete task in the most efficient manner, we decided to use the Trie data structure [2].



Trie Example [2]

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

The trie is a 26-ary tree, with each child node representing a letter in the alphabet, along with a boolean flag to represent whether the current node can be the end of a word, and an integer for the number of occurrences of the word. This is an ideal data structure for our task because we are able to efficiently find all words starting with a certain string, and then traverse through its children to find all words starting with that string. Continuing with the example from earlier, if the string "do" is entered, the tree is traversed by first going to node "d" and then to its child "o", following which a depth-first-search is performed on every child tree that forms a word.
Once we find all corresponding words and their frequencies, we are able to sort according to frequency and return the *n* most frequently seen words.

III. Data

The dataset used is a collection of all papers submitted to the NeurIPS conference in 2015 [1]. This dataset was cleaned using a Python script to remove all words that contained numbers or special characters and any single-character words. This left us with a dataset containing 1,737,937 words (11MB). We also duplicated this data 6 times to create a dataset of 10,427,550 words (63MB), which was used for all the performance calculations.

IV. Workflow

In order to tackle the project effectively, our team breaks down the topic into these sequential steps:

- Word Frequency Mapper
- Word Frequency Reducer
- Populate the Trie
- Search and find phrase and its trie(s) suggestions
- Return the top 10 most frequent words

Our test device is an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz with 12 cores and 32 GB of RAM.

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

Word Frequency Mapper

Given the corpus, which has been split into a list of words, we map that list to create a new list which contains pairs of (word, 1) for each word in the corpus.

Sequentially, we iterate through the word list and map each word to (word, 1). Our runtime for this operation is consistently **6 seconds** on a 63MB text file, which will be the benchmark for our parallelled approach.

With the help of parallelism in Haskell, we were able to improve this runtime significantly. After realizing that instead of loading the entire file in memory and generating sparks at once (parList), we realize we can use **parBuffer** with **rdeepseq** strategy to fully evaluate the expression and avoid spark pool overflow by enforcing a constant buffer size. With some experiments, we concluded that parBuffer with buffer size of 10 would be ideal strategies for our use case. Below are two tables analyzing the two metrics, which clearly demonstrates the maximum performance at buffer of size 10.

Buffer Size	Runtime	Speed Up
4	2.916	3.084
10	2.872	3.128
100	3.447	2.553
1000	5.227	0.773

With buffer size = 10

Cores	Runtime	Speed Up
1	1.841	4.159
2	2.301	3.699
4	2.656	3.344
6	2.872	3.128
8	3.012	2.988

While buffer size is correlating with speed up time with a cap at size 10, we can observe that the runtime got worse as we increased the number of cores. This made sense because our test machine has a lot of memory, which is ideal for this trivial task of the project, so increasing the number of cores means more work to put the results together, garbage collecting, and dealing with fizzled sparks rather than doing meaningful work.

The parallel method is listed below:

```
wordMapperPar :: [String] -> [(String, Int)]
wordMapperPar wl = map (, 1) wl `using` parBuffer 10 rdeepseq
```

Word Frequency Reducer

After converting all words into frequency pairs in the form of (word, 1), our reducer will proceed to add them together sequentially using Map.fromListWith. The goal is to obtain the frequency of each word in the corpus. This is the most important part of the project since it takes the most time to compute.

Sequential

As stated above, we utilized the built-in fromListWith function from the package Map to go through and add all the entries together.

```
wordsToCounts word_list = Map.toList $ Map.fromListWith (+) $ map (, 1) word_list
```

Below is a table of runtimes on different size corpora, which indicate sharp increase in runtime when increasing the length of the corpus.

Corpus size	R1	R2	R3	Average
1737925	3.159	3.211	3.162	3.177
3475850	6.501	6.291	6.523	6.438
10427550	19.772	19.517	19.258	19.516

Parallel Approach

In order to mitigate the runtime increase observed above, we decided to tackle the reducing step by splitting the list of frequency pairs into chunks, with the initial intention to parallel the compute parts of the list in parallel instead of going through the list sequentially.

In order to do so, we experimented with different chunk sizes using **chunksOf**, which split a given list into multiple chunks of size **n** which we specified. In addition, we utilized **parMap** to map our chosen strategy to each chunk. After experimenting with both parBuffer and parList, we found that **rpar** has the best performance and executed what we intended the program to do.

After parallelization, we obtained a list of maps, which we then combined to create the final representation of word frequency in our corpus.

```
wordReducerPar :: (Ord k, Num a, NFData k, NFData a) => [(k, a)] -> [(k, a)]
wordReducerPar wl = Map.toList $ Map.unionsWith (+) (parMap rpar (Map.fromListWith (+)) chunks)
                  where chunks = chunksOf 500000 wl
```

Below is the statistics of of different configurations with their average runtimes, and a Threadscope demonstration of our best run with 6 cores:

Cores

Cores	Runtime
1	21.674
2	16.132
4	10.262
6	8.543
8	10.165

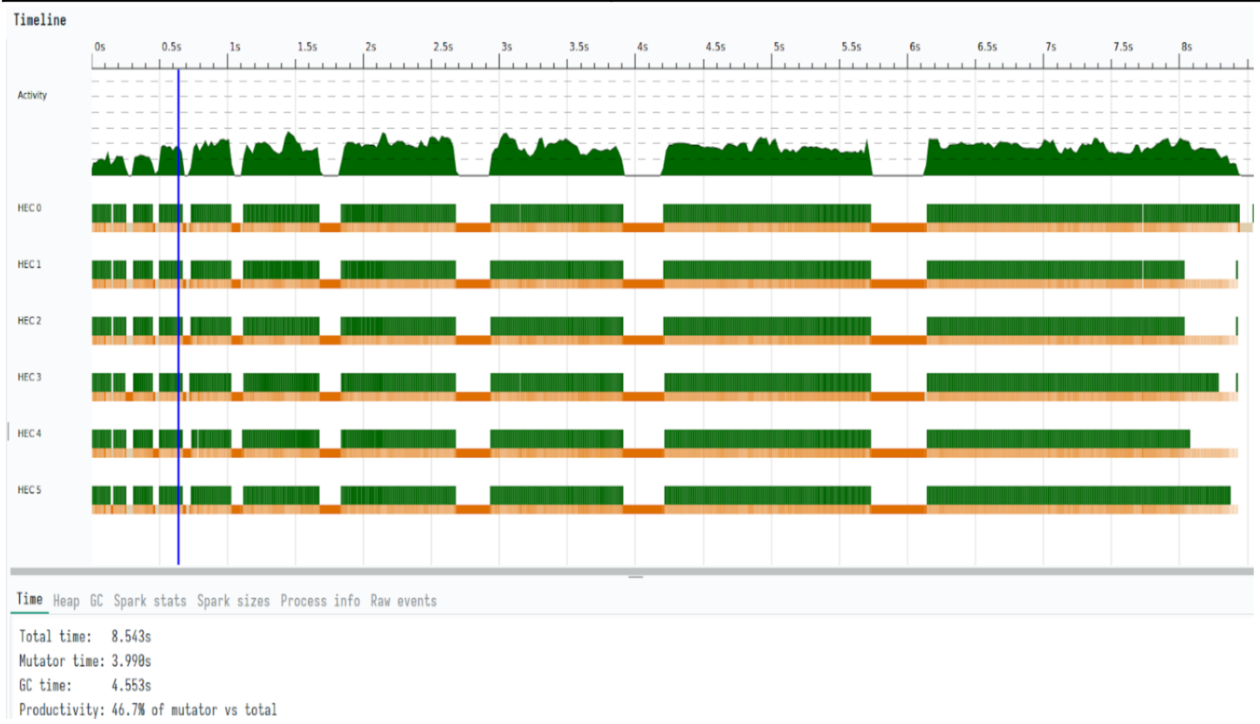
Chunk size

Chunk Size	Runtime
50	20.776
100	20.487
500	15.767
5000	12.693

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

500000	8.964
5000000	17.633



```
Total time:   8.543s
Mutator time: 3.990s
GC time:      4.553s
Productivity: 46.7% of mutator vs total
```

With sequential data as the benchmark and our experiments, we decided to go with the configurations of 6 cores and chunkSize of 500000. This will produce less sparks than smaller sizes, which reduces the overall number of overflows and garbage collection.

Trie Search

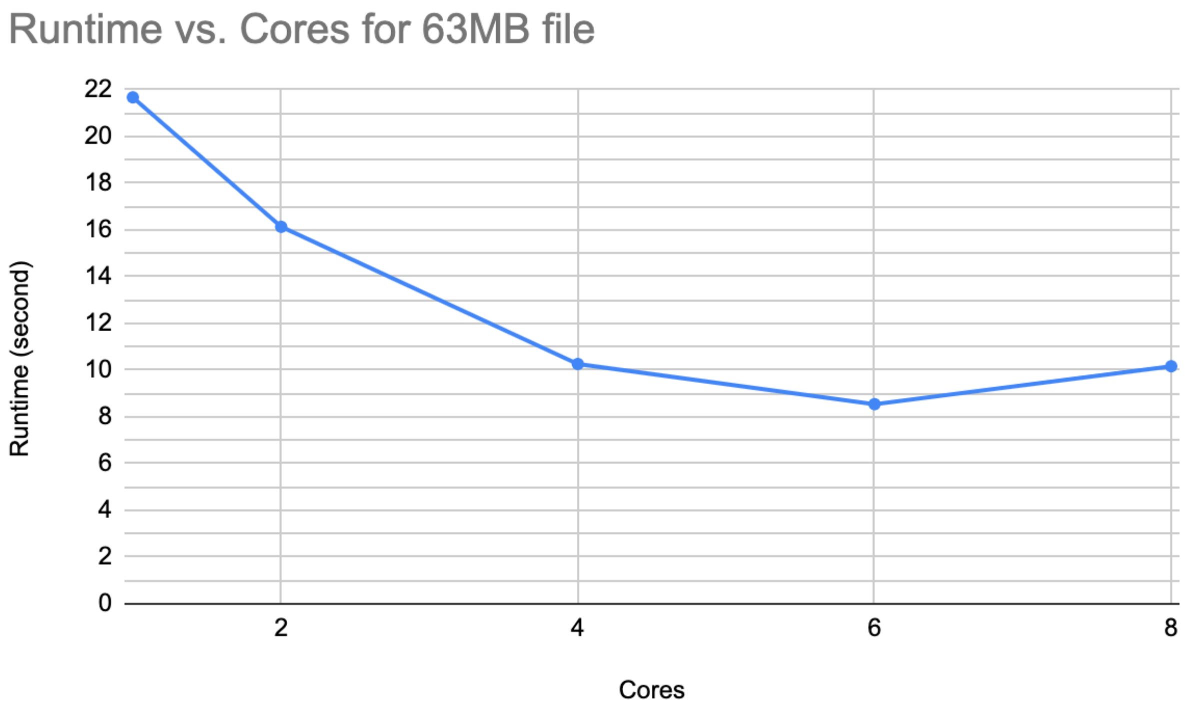
Once the data has been map reduced, we populate the Trie and then perform search. We found that the Trie population and search is very efficient and took less than 100ms. This testing was performed by populating the Trie with all words in an English dictionary, which would represent the worst case scenario. Trying to optimize the search, we experimented with a number of strategies, but the runtime remained within a reasonable error. Hence, we utilized a parallel approach that seemed theoretically the most efficient: Each child node evaluation was sparked in parallel, using rpar, but only to a depth of 4. We chose a depth of 4 since most English words have a length of 4 or 5.

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

Evaluation

File Size	Sequential (s)	Parallel (s)	Speed Up (x)
11MB	3.670	2.702	1.36x
42MB	12.130	6.602	1.84x
63MB	19.532	9.340	2.09x



Our original goal with this project was to optimize the Trie search, but we quickly realized that the true efficiency gains would be found in optimizing the mapreduce process, which made up the primary computation time. Hence, we attempted a number of optimizations to the mapping and reducing process, which gave us over a 2x improvement for 63MB file.With some more improvements to the reducing algorithm, like recursively combining the reduced chunks (similar to merge sort), may improve this further.

Surprisingly we found no noticeable improvement going from sequential to parallel. We put this down to the high efficiency of the Trie structure for this task. Overall, we were satisfied with the performance gains achieved in the data loading process, and if this was the sole purpose we may have been able to create more elaborate algorithms to further reduce this time.

V. References

1. https://archive.ics.uci.edu/ml/datasets/NIPS+Conference+Papers+1987-2015
2. http://theoryofprogramming.azurewebsites.net/wp-content/uploads/2015/06/trie12.jpg

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

VI. Code

Main.hs

```
module Main where

import qualified Lib as L
import qualified Trie as T
import Data.List()
import qualified Data.Map as Map
import Control.Monad()
import System.Environment(getArgs, getProgName)
import System.Exit(die)


main :: IO ()
main = do args <- getArgs
          pn <- getProgName
          case args of
            [corpus, method] -> do
              wordList <- readFile corpus
              let ws = words wordList

              if method /= "linear" && method /= "parallel"
                then die "Invalid Method"
              else do
                let word_counts = if method == "parallel" then
L.wordsToCountPar ws else L.wordsToCounts ws

                let root = T.populateTrie (T.TrieNode Map.empty 0 False)
word_counts

                print "Please Enter Search Term below:"
                search <- getLine
                let searchNode = T.searchTrie search root ""
                let results = T.fanTriePar searchNode search 4

                putStrLn ""
                mapM_ (putStrLn . L.showPair) (L.topNWords 10 results)
            _ -> die $ "Usage: " ++ pn ++ " <corpus-path> <running-method
linear|parallel>"
```

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

Lib.hs

```
{-# LANGUAGE TupleSections #-}
module Lib
    (
      wordsToCounts,
      wordsToCountPar,
      topNWords,
      showPair
    ) where
import qualified Data.List as List
import Data.List.Split(chunksOf)
import qualified Data.Map as Map

import Control.Parallel.Strategies (using, parBuffer, rdeepseq, parList,
NFData, rpar, parMap)


wordsToCounts :: [String] -> [(String, Int)]
wordsToCounts [] = []
wordsToCounts word_list = Map.toList $ Map.fromListWith (+) $ map (, 1)
word_list

wordMapperPar :: [String] -> [(String, Int)]
wordMapperPar wl = map (, 1) wl `using` parBuffer 10 rdeepseq

wordReducer :: (Ord k, Num a) => [(k, a)] -> [(k, a)]
wordReducer wl =  Map.toList $ Map.fromListWith (+) wl

wordReducerPar :: (Ord k, Num a, NFData k, NFData a) => [(k, a)] -> [(k, a)]
wordReducerPar wl = Map.toList $ Map.unionsWith (+) (parMap rpar
(Map.fromListWith (+)) chunks)
                    where chunks = chunksOf 500000 wl

wordsToCountPar :: [String] -> [(String, Int)]
wordsToCountPar word_list = wordReducerPar $ wordMapperPar word_list

topNWords :: Int -> [(String, Int)] -> [(String, Int)]
topNWords n l = take n $ List.sortBy (\(_, a) (_, b) -> compare b a) l

showPair :: (String, Int) -> [Char]
showPair (x, y) = show y ++ "  " ++ x
```

Trie Autocomplete

Thang Nguyen (tn2468) and Siddharth Pittie (sp4013)

Trie.hs

```haskell
module Trie (Trie(..), insertIntoTrie, populateTrie, searchTrie, fanTrie,
fanTriePar) where

import qualified Data.List as List
import qualified Data.Map as Map
import Control.Monad
import Data.Maybe
import Control.Parallel.Strategies (parMap, runEval, using, rdeepseq, parList,
rpar)

data Trie = TrieNode (Map.Map Char Trie) Int Bool | EmptyNode deriving (Show,
Read, Eq)

insertIntoTrie :: String -> Int -> Trie -> Trie
insertIntoTrie _ _ EmptyNode = EmptyNode
insertIntoTrie [] _ t = t
insertIntoTrie [c] count (TrieNode children freq end) =
    case Map.lookup c children of
      Nothing -> TrieNode (Map.insert c (TrieNode Map.empty count True)
children) freq end
      Just EmptyNode -> TrieNode (Map.insert c (TrieNode Map.empty count True)
children) freq end
      Just (TrieNode m _ _) -> TrieNode (Map.insert c (TrieNode m count True)
children) freq end

insertIntoTrie (c:cs) count (TrieNode children freq end) =
    case Map.lookup c children of
      Nothing -> TrieNode (Map.insert c (insertIntoTrie cs count (TrieNode
Map.empty 0 False)) children) freq end
      Just v -> TrieNode (Map.insert c (insertIntoTrie cs count v) children)
freq end

populateTrie :: Foldable t => p -> t (String, Int) -> Trie
populateTrie word_counts = foldl (\ x (w, f) -> insertIntoTrie w f x)
(TrieNode Map.empty 0 False)

searchTrie :: String -> Trie -> String -> Trie
searchTrie _ EmptyNode _ = error "Bad Trie"
searchTrie [] trie ans = trie
searchTrie (c:cs) (TrieNode trie freq end) ans =
  case Map.lookup c trie of
    Nothing -> TrieNode trie freq end
    Just node -> searchTrie cs node (ans ++ [c])


fanTrie :: Trie -> String -> [(String, Int)]
fanTrie EmptyNode _ = []
fanTrie (TrieNode m freq end) word_so_far
  | end && not (null child_list) = (word_so_far, freq) : foldl1 (++) (map
(\(c, node) -> fanTrie node (word_so_far ++ [c])) child_list)
  | end = [(word_so_far, freq)]
  | not (null child_list) = foldl1 (++) (map (\(c, node) -> fanTrie node
(word_so_far ++ [c])) child_list)
  | otherwise = []
  where
    child_list = Map.toList m


fanTriePar :: Trie -> [Char] -> Int -> [(String, Int)]
fanTriePar EmptyNode _ _ = []
fanTriePar (TrieNode m freq end) word_so_far depth
  | end && not (null child_list) = do
    if depth > 0
      then
        (word_so_far, freq) : foldl1 (++) (map (\(c, node) -> fanTriePar node
(word_so_far ++ [c]) (depth - 1)) child_list `using` parList rdeepseq)
      else
        (word_so_far, freq) : foldl1 (++) (map (\(c, node) -> fanTriePar node
(word_so_far ++ [c]) (depth - 1)) child_list)
  | end = [(word_so_far, freq)]
  | not (null child_list) = do
    if depth > 0
      then
        foldl1 (++) (map (\(c, node) -> fanTriePar node (word_so_far ++ [c])
(depth - 1)) child_list `using` parList rdeepseq)
      else
        foldl1 (++) (map (\(c, node) -> fanTriePar node (word_so_far ++ [c])
(depth - 1)) child_list)
  | otherwise = []
  where
    child_list = Map.toList m
```