COMS 4995: Parallel Functional Programming (Fall '19) December, 2019

Parallel Autocomplete

Authors: Sambhav Anand (sa3433), Kanishk Vashisht (kv2295)

Abstract

The task of autocompletion is returning an arbitrary number of popular words that begin with a suggested prefix. Our project was centered around parallelising the entire autocomplete workflow. We look at parallelly constructing a word frequency table and then traversing a trie to autocomplete a given word. We compare single core implementations versus multiple core implementations on small and large bodies of text.

Contents

1 Introduction **1**

2 Data Gathering **1**

3 Sequential approach **1**
3.1 Word Frequencies 1
3.2 Word Completion 1

4 Parallel approach **1**
4.1 Word Frequencies 1
4.2 Word Completion 2

5 Results **2**
5.1 Observations 2
5.2 Results 2
5.2.1 Word Frequencies 2
5.3 Conclusion 2

6 Appendix **3**
6.1 Core Usage 3
6.2 Code Listing 4

1 Introduction

The problem we undertook was that given a corpus of text, could we return the k most popular words that began with a suggested prefix. Popularity is defined on the basis of frequency of the word in the original corpus.

We break down the original problem into two sub problems.

1. **Word Frequencies:** This is the step where the corpus is examined and reduced to a dictionary of words that includes the frequencies. This is an intermediate step that only needs to be run once for an infinite number of autocomplete queries.

2. **Word Completion:** In this step we get two inputs from the user - the prefix p and k, the number of words to return. We load the words as a trie and then pick the k most popular children with the prefix p and return them.

2 Data Gathering

We used a dataset that contains 1,097,592 Amazon reviews spanning May 1996 - July 2014 in the CDs/Vinyl section. There are a total of 200M words of text with 200K unique words in this dataset. This dataset can be found here.

3 Sequential approach

3.1 Word Frequencies

We implement word frequencies using map-reduce. Our mapping function takes a filename, reads the content from it and then generates a frequency table. Our reducer function takes these frequency tables and then combines them into one bigger frequency table. We convert this frequency table to a list of (word, frequency) pairs which we then write to a file.

3.2 Word Completion

When its time for word completion, we load our file into a trie. At this point we ask the user for an input and return the k most common words that begin with the provided input.

4 Parallel approach

4.1 Word Frequencies

Our parallel approach is similar to our sequential approach except we run the mapping functions in parallel on different files.

4.2 Word Completion

We tried to parallelize our word completion by querying the trie in parallel. We find the parent node that represents the prefix, then we traverse all of its children nodes in parallel and return the k most common words from each. After this we find the k most common words in total and return those. However, we found that for even the most basic prefixes such as "pre" or "pr" the sequential approach was outperforming our rudimentary parallel approaches as there were only close to 4-5000 words to sort through and pick k from. We believed that parallelising this step would not change much so instead we focused on improving the efficiency of word frequencies to scale.

5 Results

5.1 Observations

One interesting thing we observed was that if the files were not distributed well then the cores did not provide much optimization. This is obvious since we parallelize over files. This is why for the general results that follow we made sure to divide our dataset into 55 evenly distributed files.

5.2 Results

5.2.1 Word Frequencies

The following table shows our results for the word frequency part. We look at how things change by making the corpus larger and by adding more cores.

# Words / Cores	1 core	4 cores	8 cores
417,017,098	966s	416s	238s
208,508,549	355s	140s	136s
75,667,556	160s	49s	37s
1,899,4220	40s	16s	16s

We've also attached the output artifacts (in the appendix section) showing the usage of our cores for the 400 million word data set. As one can see all our cores are being used well and as we're adding more cores they are getting busier as well. This shows that our algorithm is using parallelization efficiently. This is also evident in the fact that we can deal with a 417m word corpus in 238s.

5.3 Conclusion

In conclusion, we have:

1. Come up with a full algorithm for word completion that is broken into word frequencies and word completion
2. We are able to parallelise word frequencies to a large scale (417m words).

6 Appendix

6.1 Core Usage

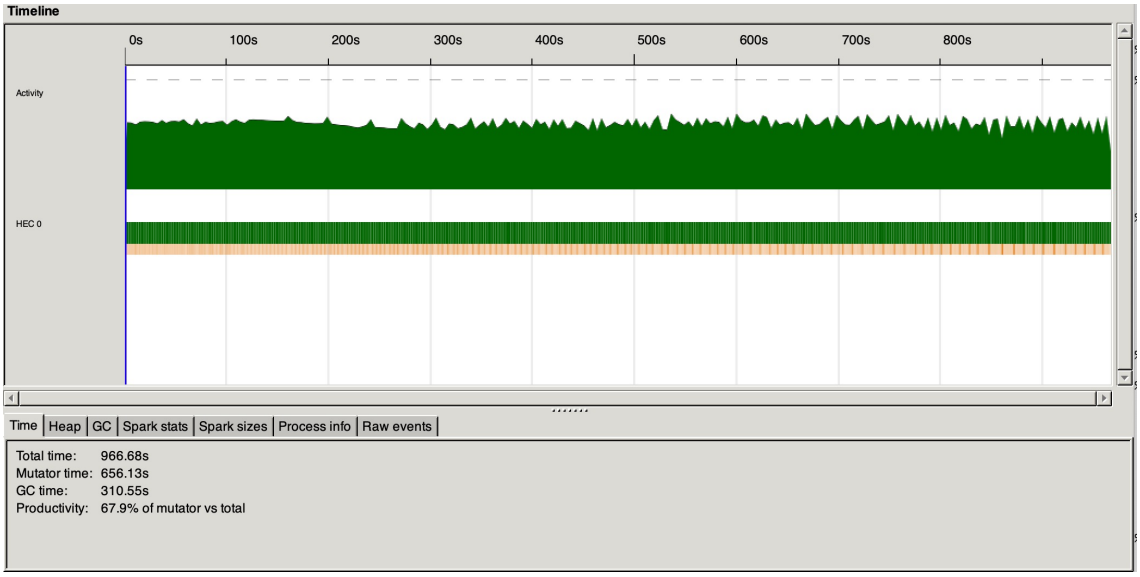

Figure 1: One Core

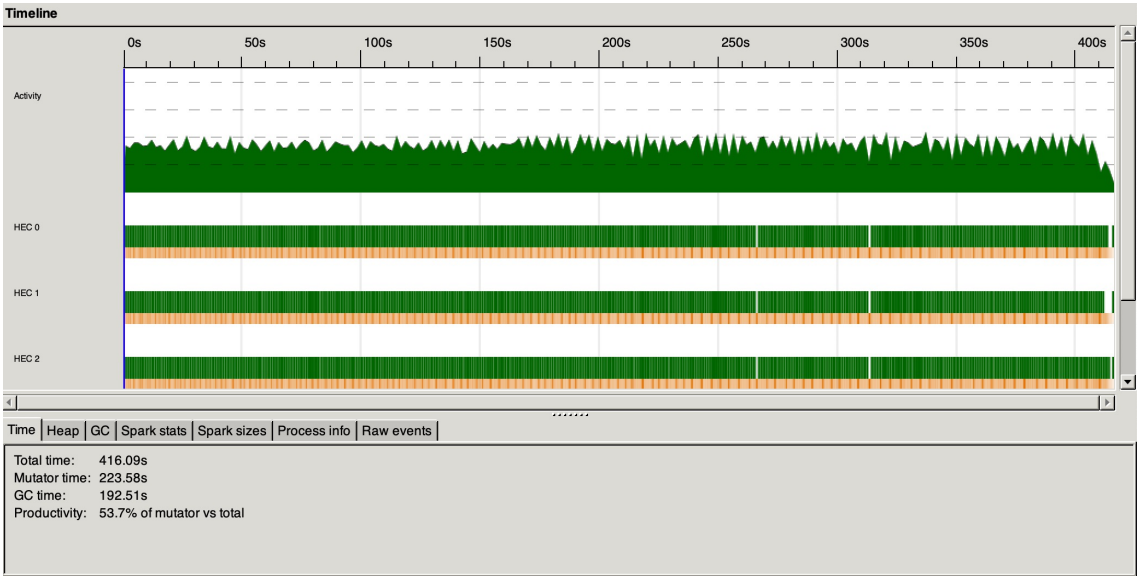

Figure 2: Four Cores

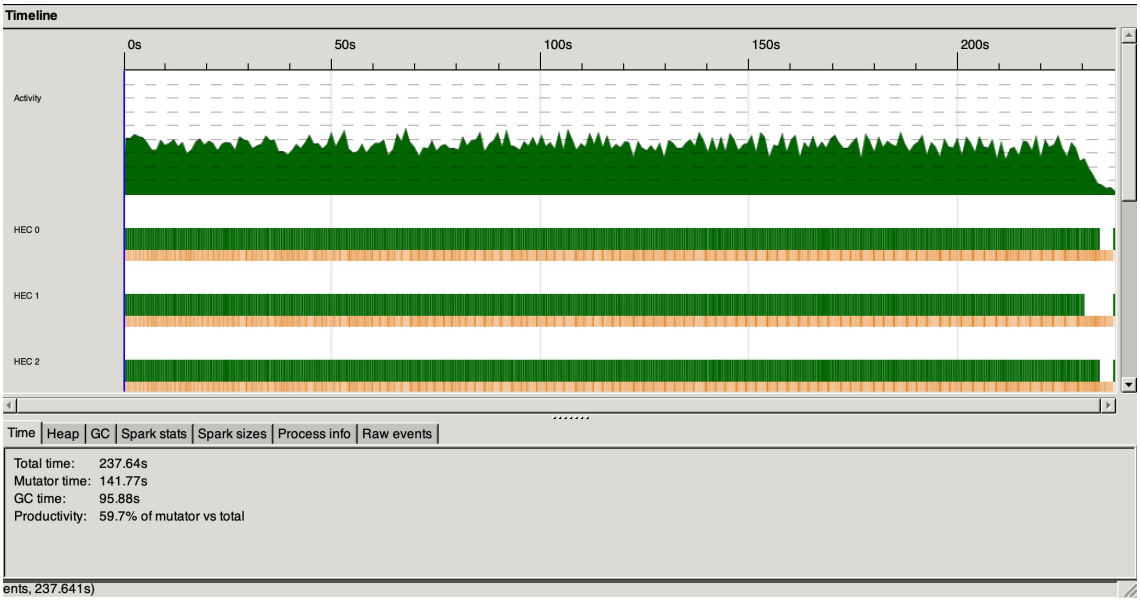

Figure 3: Eight Cores

6.2 Code Listing

```
module Main where

import Data.List (sort, group, sortBy, groupBy, isPrefixOf, isInfixOf, sortOn)
import Data.Text (isSuffixOf)
import Control.Parallel
import Control.Parallel.Strategies
import Data.Char (isAlpha, isSpace, toLower)
import qualified Data.ByteString.Char8 as BC
import qualified Data.Trie as T
import qualified Data.Map as M
import System.Directory
import System.IO

mapReduce :: NFData b1 => (a -> b1) -> ([b1] -> b2) -> [a] -> b2
mapReduce mapper reducer input = pseq mOutput rOutput
  where mOutput = parMap (rpar `dot` rdeepseq) mapper input
        rOutput = reducer mOutput `using` rseq


mappingFunc :: String -> M.Map String Int
mappingFunc document = getWordFreqMap $ (strToList) document


getWordFreqMap :: [String] -> M.Map String Int
getWordFreqMap tokens = M.fromListWith (+) (map (\x -> (x, 1)) tokens)

strToList  ::  String -> [String]
strToList  str  = words $ filter (\char -> isAlpha char || isSpace char) $ map toLower str

```

```
reducingFunc :: [M.Map String Int] -> [(String, Int)]
reducingFunc maps = M.toList $ foldl (\foldMap (word, count) -> M.insertWith (+) word count foldMap) (M.empty
listOfMaps
                    where listOfMaps = concat (map M.toList maps)

loadList :: FilePath -> IO (M.Map String Int)
loadList fname = do
     filedata <- readFile fname
     contents <- return (read filedata :: [(String, Int)])
     let loadedMap = M.fromList contents
     return loadedMap

saveList :: Show a => a -> FilePath -> IO ()
saveList ls fname= writeFile fname (show ls)

FilePath -> IO ()
buildWordList directory = do
                          files <- listDirectory directory
                          let myFiles = filter (\x -> not (elem x [".", ".."])) (map (\x -> directory ++ x) files)
                          parsedFiles <- mapM readFile myFiles
                          let freqMap = mapReduce mappingFunc reducingFunc parsedFiles
                          putStrLn $ "Found " ++ (show.length) freqMap ++ " words!"
                          saveList freqMap "word_counts.txt"

buildTrie :: M.Map String Int -> T.Trie Int
buildTrie loadedMap = T.fromList (map (\x -> (BC.pack (fst x), snd x::Int)) (M.toList loadedMap))


Ord b1 => T.Trie b1 -> IO b2
userInput trie = do
   putStrLn "enter word"
   pre <- getLine
   putStrLn "Enter number of words you want..."
   k <- getLine
   let topk = take (read k::Int) (sortBy (\(_,a) (_,b) -> flip compare a b) $ T.toList (T.submap (BC.pack pre
   putStrLn "Found the following words"
   putStrLn " - Start - "
   mapM_ (\(a,_) -> putStrLn $ BC.unpack a) topk
   putStrLn " - Finish - "
   userInput trie


main :: IO ()
main = do
--    buildWordList "./data/"
     loadedMap <- loadList "data.txt"
     let trie = buildTrie loadedMap
     putStrLn $ "Found " ++ (show $ T.size trie) ++ " words"
     userInput trie
```

Citation: the format for this code can be found here