Predictive Text Au|tocomplete

By: Sebastian Hereu (smh2278) & Elisa Luo (eyl2130)

Introduction

We built an n-gram language model in Haskell that when given a corpus of English text and a prefix of a word in a context (e.g., *He was y…*) can complete the word (e.g., *He was yelling).* Through the use of parallelization, we were able to speed up the construction of the n-gram model by 3.5x and generate predictions on-the-fly. We used corpus data from the “Linear Text” section of, https://www.corpusdata.org/formats.asp

Overview

The program takes the name of the directory containing assorted corpus files. The program will first build the n-gram model. Then, when the model is built, the program will repeatedly ask for the user to provide a prefix, and will return a prediction until the user terminates the program. Since the predictions are relatively fast to generate once the language model is built, the model can quickly[1] spit out a prediction given the prefix. The test datasets provided in the code include a corpus of 14 million words (in `fcorp`), and a smaller test corpus of ~5 million words (in `bcorp`). With our implementation, it takes about 45s to build the language model using `bcorp`, and we are able to generate plausible, context-based predictions using the language model. The contents of `fcorp` and `bcorp` are shown below:

```
fcorp: coca-text_acad.txt, coca-text_blog.txt,
coca-text_fic.txt, coca-text_mag.txt, coca-text_news.txt,
coca-text_spok.txt, coca-text_tvm.txt, coca-text_web.txt,
movies_text.txt, tv_text.txt, wiki.txt

bcorp: coca-text_acad.txt, coca-text_mag.txt, tv_text.txt,
wiki.txt
```

`wiki.txt` contains 1.8mw, `movies_text.txt` 1.6M, `tv_test.txt` 2.1mw, and the `coca-text*` files collectively contain about 8.9mw.

[1] On average, ~100ms

Sequential Implementation

Although the program was initially designed with parallelism in mind, we provide sequential versions of most functions for comparison purposes.

Building the Language Model

Processing Input

Since the corpus can be of arbitrary size (i.e., very large), reading it all into memory at once, then processing it, would be a waste of memory. Thus, we decided to lazily read the directory of files specified by the command line argument using `Data.Text.Lazy`. Then, tokenizing the content and turning everything to lowercase, before we start generating the Tries.

Storing the prefixes and their potential matches in Tries

To store individual words from the corpi and generate potential matches for an incomplete word, we use a trie data structure. Each node in a trie will represent a prefix of a string and the subtrees of a given node contain all words the current node is a prefix of. For example, the node representing prefix “fa” might have the children “family”, “farm”, “fame”, and “farce” contained in its subtrees. We know that a node represents a valid word from the corpus if the isEnd boolean flag is set at a given node. The Trie is asymptotically the best data structure for string storage and prefix lookup, allowing for logarithmic prefix lookup times.

Using multiple corpus files from different contexts, e.g. television, magazines, and wikipedia articles, we created a forest of n-gram Tries. We choose to create a separate trie for each corpus for two reasons. First, each corpus has words from a different context, so it makes organizational sense to keep the Tries separate. Second, a forest, or array of tries, works handsomely with Haskell's concurrency, as the tries could be built and searched in parallel. Our trie data structure is shown below:

```
data Trie = Node Bool Int (M.Map Char Trie) | Empty deriving (Eq, Read).
```

The creation of the forest of Tries is done concurrently in the `makeForest` function, which also reads and processes the raw corpus files:

```
makeForest :: [String] -> IO [Trie]
makeForest filteredDirs = do
                      let docs = map TIO.readFile filteredDirs
                      let tok = TL.splitOn (TL.pack " ")
                      let fn =  (return::(a -> IO a)) (tok.TL.toLower) -- lowercase
                      let ws = map (fn <*>) docs
                      sequence (S.parMap S.rpar (return buildTreeT <*>) ws)
```

Building the Language Model

To build the language model, we needed to create a map of n-gram counts, which are used to generate the scores for word-completion predictions. We knew that we ultimately were going to parallelize the building of the n-gram maps (via `computeNGramFrequencies`) using the popular mapreduce paradigm, but started with a started with a sequential version that essentially moves a sliding window of size n across all the documents in a corpus to generate ngrams.

```
computeNGramFrequencies :: Int -> Corpus -> M.Map NGram Int
computeNGramFrequencies n corp = let tupCounts = map (\x -> (x, 1::Int)) (computeNGrams n corp)
                                     histogram = M.toList $ M.fromListWith (+) tupCounts
                                     in M.fromList histogram
```

We then called `computeNGramFrequcies` sequentially to compute the n-gram frequencies for all the n-grams up to and including n (e.g., 1-grams, 2-grams, 3-grams, … n-grams) via our `createNGramMap` function, shown below.

```
createNGramMap :: Int -> Corpus -> M.Map NGram Int
createNGramMap n corp = let mps = map (`computeNGramFrequencies` corp) [1..n] in
foldl (M.unionWith (+)) M.empty mps
```

Making Predictions

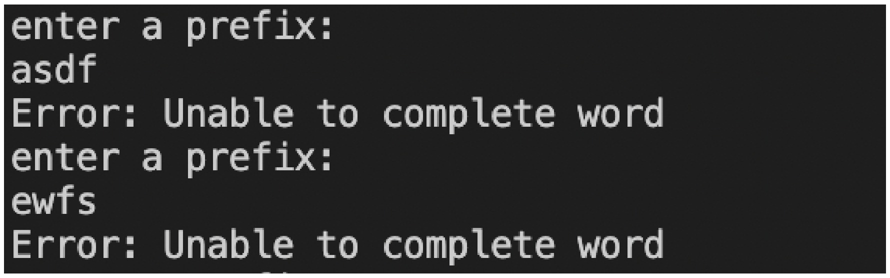
Now that the language model is built, we can finally use it to complete sentences! The function `getPrediction` takes a sentence with an incomplete word as its end. Using the forest of tries generated by makeForest, `getPrediction` generates all potential matches with the incomplete word. For each of these 'guesses' (as it's called in code below), we use the preceding words in the sentence to generate a context-based prediction via the score function (also shown below), which uses the ngram maps.

```
getPrediction :: String -> [M.Map NGram Int] -> [Trie] -> String
getPrediction sent mps tries = let     sentence = map T.pack (words sent)
                                      curNgram = delLast sentence
                                      lst = last sentence
                                      guesses = map (\x -> (T.pack $ reverse $ fst x, snd
x)) (generateMatches (T.unpack lst) tries)
                                      scores = S.parMap S.rdeepseq (\x -> (T.unpack $ fst x,
snd x)) (map (getScore curNgram mps) guesses)
                                      in (fst $ head (reverse $ sortOn snd scores))
```

To improve predictions for >2-gram inputs, we also consider the 2-gram score in addition to the n-gram score (code omitted for brevity).

```
score :: NGram -> [M.Map NGram Int] -> (Token, Int) -> (Token, Double)
score curNgram mps (guess, _) = let freq = searchMaps (curNgram ++[guess]) mps in (guess,
fromIntegral freq / fromIntegral (searchMaps curNgram mps))
```

In the case where no matches were found in the Tries, such as in the case of a nonsensical prefix (e.g., “asdf” or “ewfs”), `getPrediction` will return an error. We think this behavior is justifiable as an autocomplete program should not be expected to complete nonsensical words, and the model is able to complete most prefixes.



```
enter a prefix:
asdf
Error: Unable to complete word
enter a prefix:
ewfs
Error: Unable to complete word
```

Parallelization Strategies

The design of our program clearly allowed for many opportunities for parallelism. The slowest and most memory-hungry part of the program is, as mentioned above, generating the language model. Getting the predictions is relatively fast, even for the sequential implementation, but we can make it nearly instantaneous through parallelization, too. Below outlines the parallelization strategies we employed (or attempted to employ) in our program. Note that when comparing non-concurrent and concurrent aspects of our code, we mostly replaced maps with ParMaps.

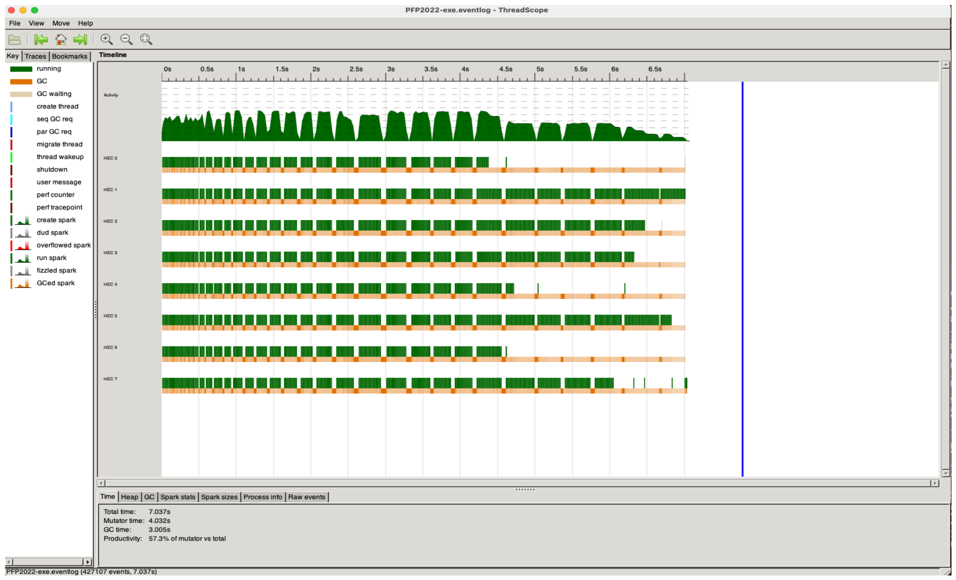
All outputs shown are run on an M1 mac with 32gb of RAM and a 2.3 GHz 8-Core Intel Core i9 processor. We ran our program with the -N8 flag when testing to fully take advantage of the cores. Note that we tested separate modules in our code by simply commenting out the non-relevant code for a test.

Searching for a prefix within the Forest of Tries

When searching for a common prefix within a forest of tries in parallel, we get up to a 5X speedup compared to our single-threaded version. The two Threadscope screenshots below give great insight into the search for the matches with the prefix “app” within fcorp. We achieve this parallelism by simply changing the `map` to a `parMap` in our `generateMatches` function:

```
generateMatches ::  String -> [Trie] ->  [(String,Int)]
generateMatches sent tries = concat $ S.parMap S.rdeepseq (prefixNodePar 0 sent "") tries
```

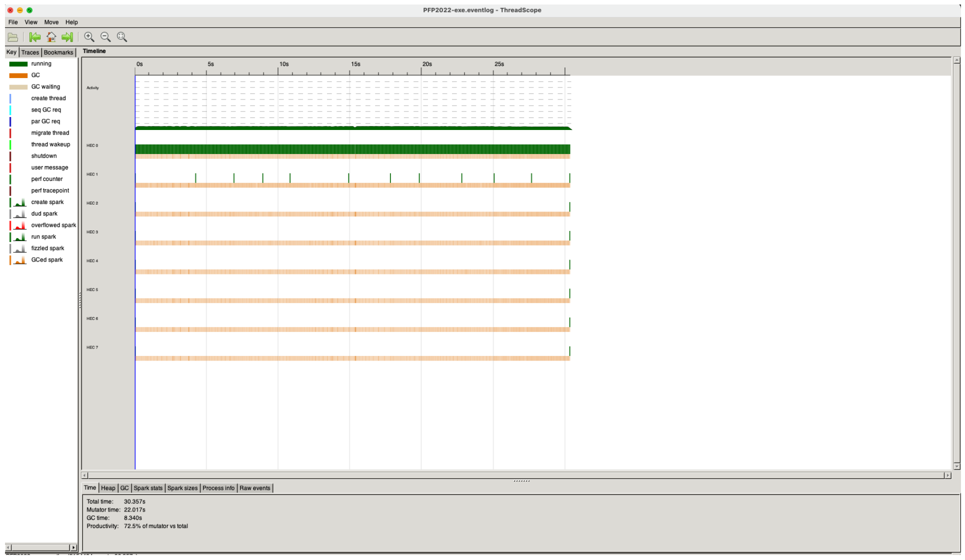
In the parallel implementation, we see all cores are busy, although the second and sixth cores have more work than the other cores (Figure 1). This can be explained by an imbalance in the size of the tries :some of the corpi in bcorp differ in size by as much as 3M words . We have a total time here of t = 7.007s. The speedup may have been limited by garbage collection, which can be shown by the “choppiness” of each thread in the output (Figure 1).



```
Total time:     7.037s
Mutator time:   4.032s
GC time:        3.005s
Productivity:   57.3% of mutator vs total
```

PFP2022-exe.eventlog (427107 events, 7.037s)

Figure 1: Multi-core execution of prefix search



```
Total time:     30.357s
Mutator time:   22.017s
GC time:        8.340s
Productivity:   72.5% of mutator vs total
```

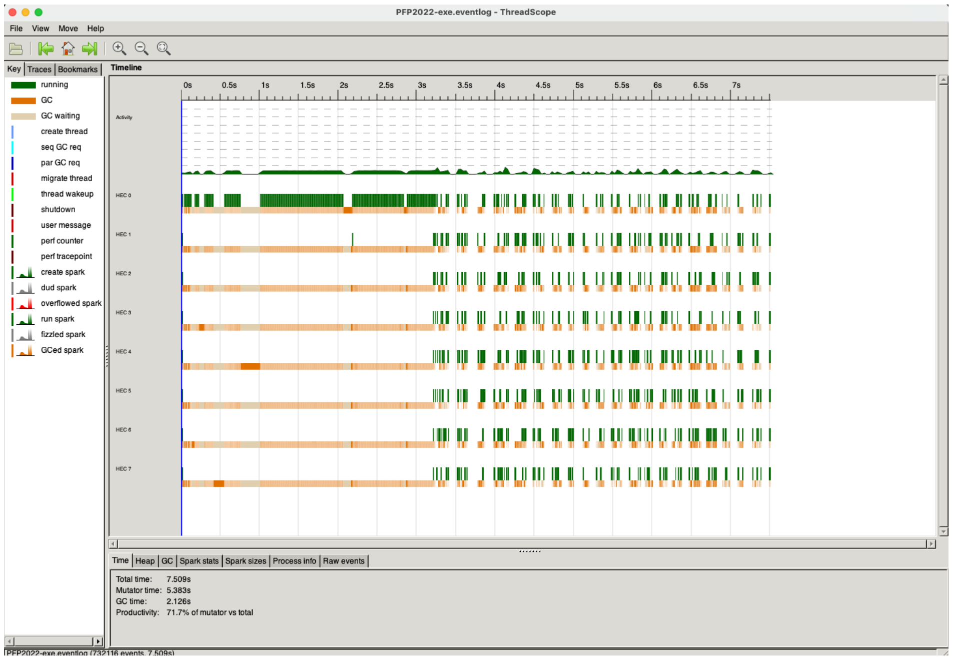
Figure 2: Single-core execution of prefix search. (total time: 30s)

Parallelising DFS searches on an individual Tree

Unfortunately, we obtained no speedup when parallelising the DFS search for prefixes on a single Trie, regardless of depth (this is explained below). The overhead of the parallelism negated any benefit over a regular, non-parallelized search for prefixes. This likely has to do with the fact that most words get reused in common English writing, news, and entertainment, meaning that for a given prefix, there are few (asymptotically speaking) children in the trie. Also, the parallelism could not begin via DFS until we found the prefix node parent, which could potentially be deep in the trie. We parallelized the DFS search for prefixes with our `prefixSearchPar` function, which allowed us to test for several depths of parallelism on the trie.

```haskell
prefixSearchPar:: Int -> (String, Trie) -> [(String,Int)]
prefixSearchPar depth (scat, Node isEnd count children)
       | depth > 0 = let childList = zipWith (\a b -> (fst a:b, snd a) ) (M.toList children) (replicate (length children) scat)
                         curr = if isEnd then scat else ""
                         in (curr, count): (concat $ S.parMap S.rseq (prefixSearchPar (depth - 1)) childList)
       | otherwise = prefixSearchSeq (scat, (Node isEnd count children))
prefixSearchPar _ _ = error "empty"
```

In an isolated experiment with depth = 3, we noticed no speedup over the single-core case. We can see that a large portion of the computation is sequential in nature (building the Trie and finding the parent prefix node), and the actual DFS produced a lot of sparks that were GC'd and fizzled. The overhead of bookkeeping and the "overkill" of parallelism explains this extremely choppy threadscope output (Figure 3). The Threadscope output looked similar for depths = 5, 4, and 2, 1, with the output having less choppiness at lower depths.



```
Total time:     7.509s
Mutator time:   5.383s
GC time:        2.126s
Productivity:   71.7% of mutator vs total
```

PFP2022-exe.eventlog (732116 events, 7.509s)

Figure 3: Paralellized DFS search on a Trie with n=3

Generating the n-grams maps and frequencies in parallel

Computing the n-gram maps and the n-gram frequencies is actually sequential for each n, (e.g. for 2-grams it must compute the n-grams map and then compute the frequencies), so in theory we would get a significant speedup from computing both steps in parallel. This is achieved by using `parmap` on both the functions `createNGramMapPar` and `computeNGramFrequenciesPar`. For `computeNGramFrequenciesPar` in particular, we used a mapreduce approach in which we divided the corpus documents into chunks corresponding to each core, computed the frequencies in parallel, and combined these results into one map.

createNGramMapPar → generate up to n n-grams (e.g. 1-grams, 2-grams, 3-grams…)

By using `parmap` and `deepseq` in this function, we achieve 1 level of parallelism with generating the 1-gram, 2-grams, 3-grams, .. n-grams. This in isolation generates n sparks total. There are a couple drawbacks to this approach (in isolation). If we have 8 cores, then and n<8, then only n cores are being used. Also, creating the n-gram maps for some n is faster than others, so we don't get great utilization of all cores (Figure 4).

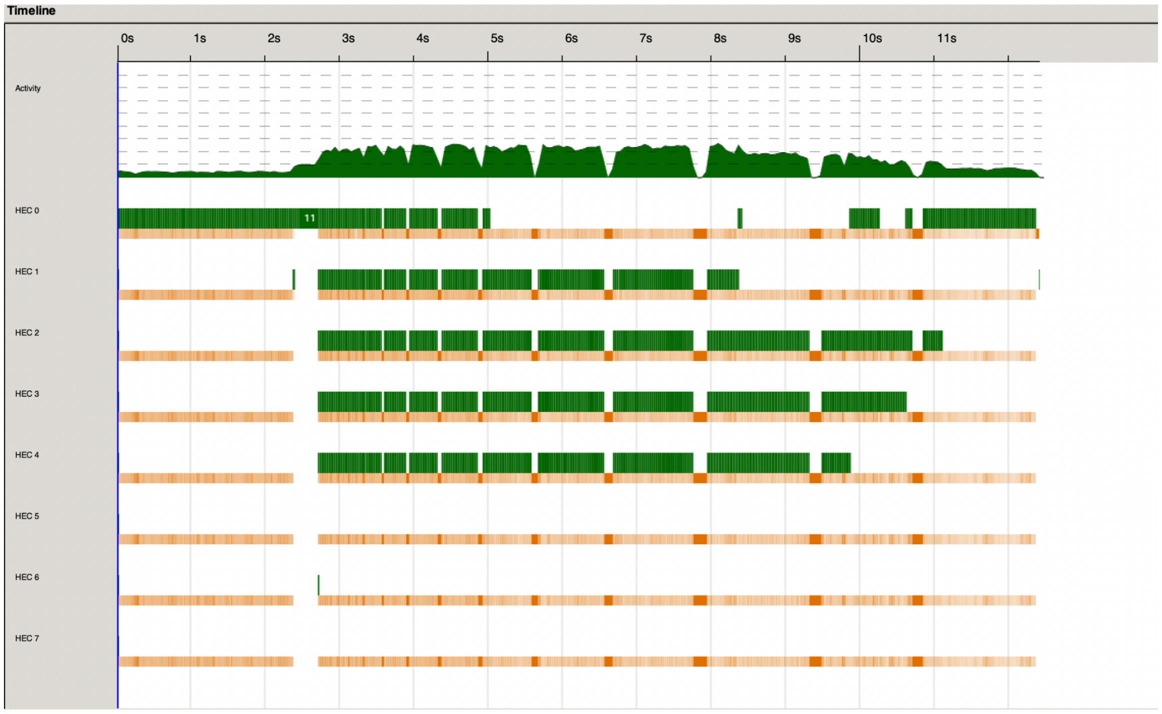


Figure 4: Threadscope output for createNGramMap Par, for n=5. As shown, only 5 cores are being used (despite running with 8 threads of execution), and some cores are being less utilized than others.

```
createNGramMapPar :: Int -> Corpus -> M.Map NGram Int
createNGramMapPar n corp = let mps = S.parMap S.rdeepseq (flip
(computeNGramFrequenciesPar 8) corp) [1..n] in foldl (M.unionWith (+)) M.empty mps
```

computeNGramFrequenciesPar → split corpus into *c* chunks and compute n-gram frequencies

Another way we parallelized is by splitting corpus into *c* chunks (Theoretically 1 for each core). This in isolation generates *c* sparks. From the threadscope output (Figure 5), we realized that creating the n-gram maps and computing the n-gram frequencies is actually sequential for each n, so we see that when creating the n-gram maps it only uses 1 core.

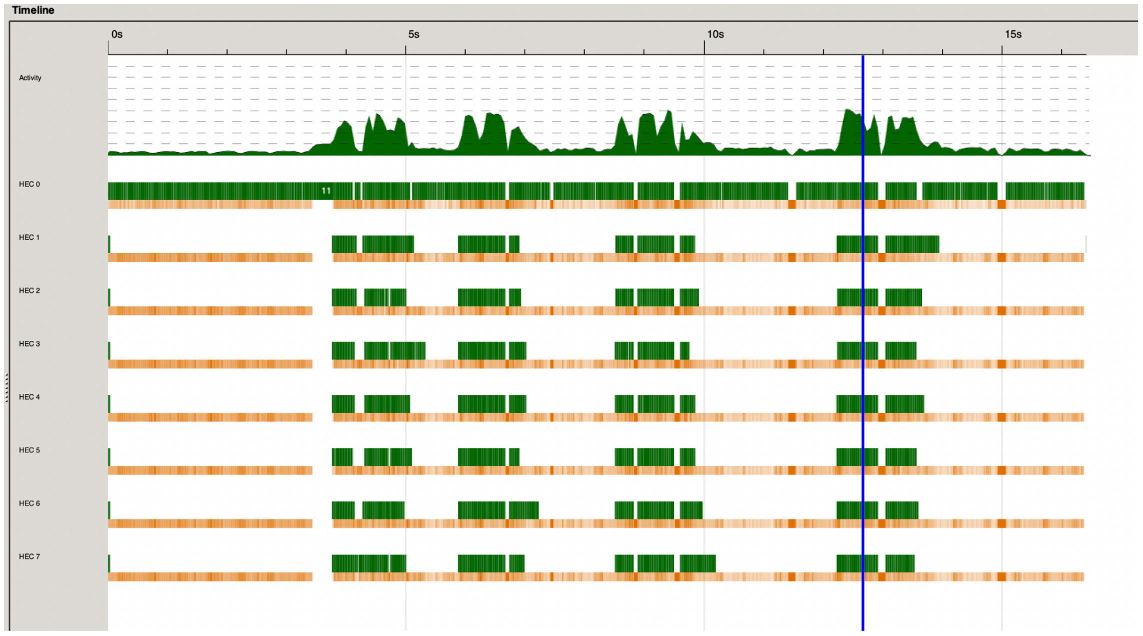


Figure 5: Threadscope output for computeNGramFrequenciesPar. This function splits the corpus into 8 chunks, and computes the n-gram frequencies for each of those chunks in parallel.

```haskell
computeNGramFrequencies :: Int -> Corpus -> M.Map NGram Int
computeNGramFrequencies n corp = let tupCounts = map (\x -> (x, 1::Int)) (computeNGrams n corp)
                                     histogram = M.toList $ M.fromListWith (+) tupCounts
                                     in M.fromList histogram


computeNGramFrequenciesPar :: Int -> Int -> Corpus -> M.Map NGram Int
computeNGramFrequenciesPar nochunks n corp = let chunks = splitInto nochunks corp in foldl (M.unionWith
(+)) M.empty (S.parMap S.rdeepseq (computeNGramFrequencies n) chunks)
```

Parellizing Both (the fastest option)

When combining both parallelization strategies of computing the 1-grams, 2-, … n-grams in parallel and the n-gram frequencies of chunks of the corpus in parallel, we achieved a ~4x speedup compared to the sequential implementation. This generates n*c sparks total, where n is the n-grams generated (we used up to 5-grams), and c is the number of chunks to split the corpus into.

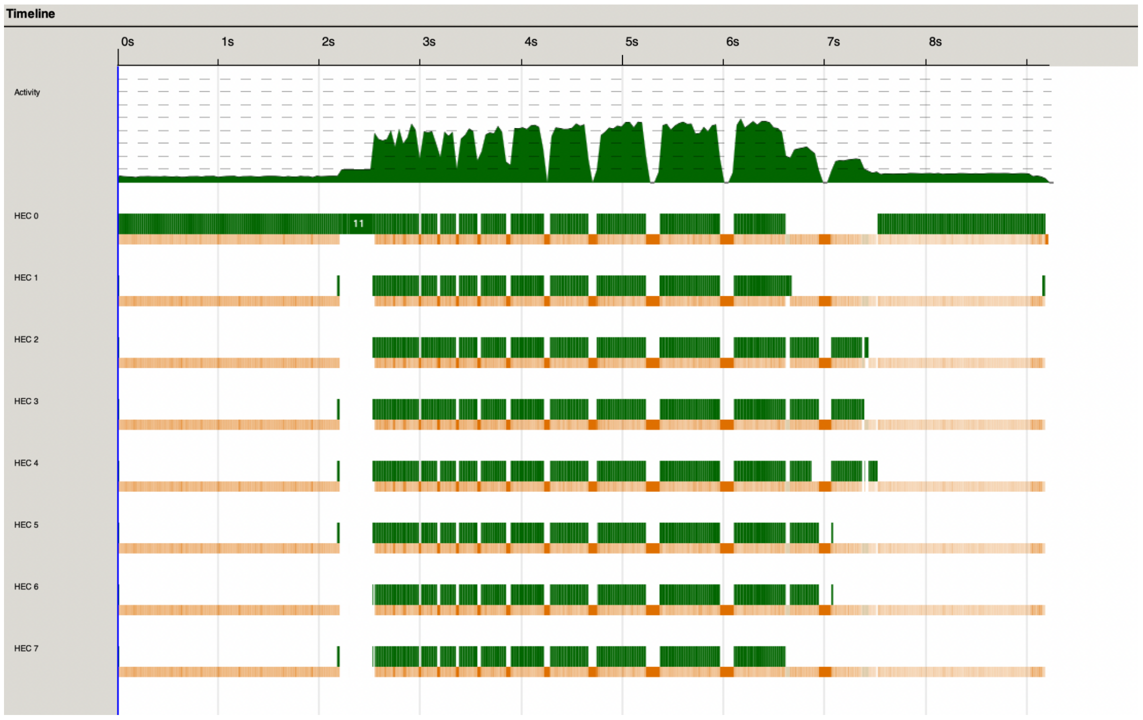
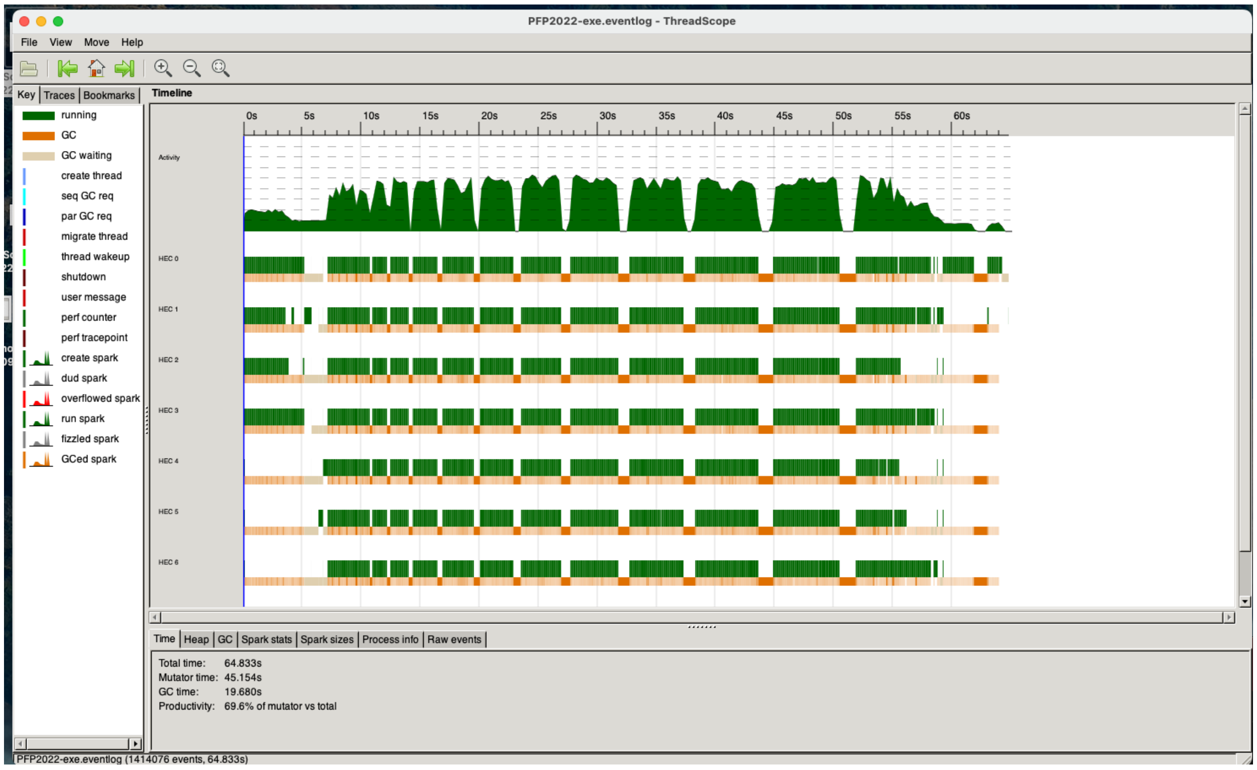


Figure 6: Threadscope output for generating the n-gram maps and the frequencies in parallel. The dips are likely due to garbage collection, but they are not significant.

Overall Performance of building the language model (Predicting in parallel):

Using our good results from the parallel prefix search on the Tries and map-reduce on the ngrams, we knew that we could obtain a sizable overall speedup, from generating the language model to returning a single prediction.

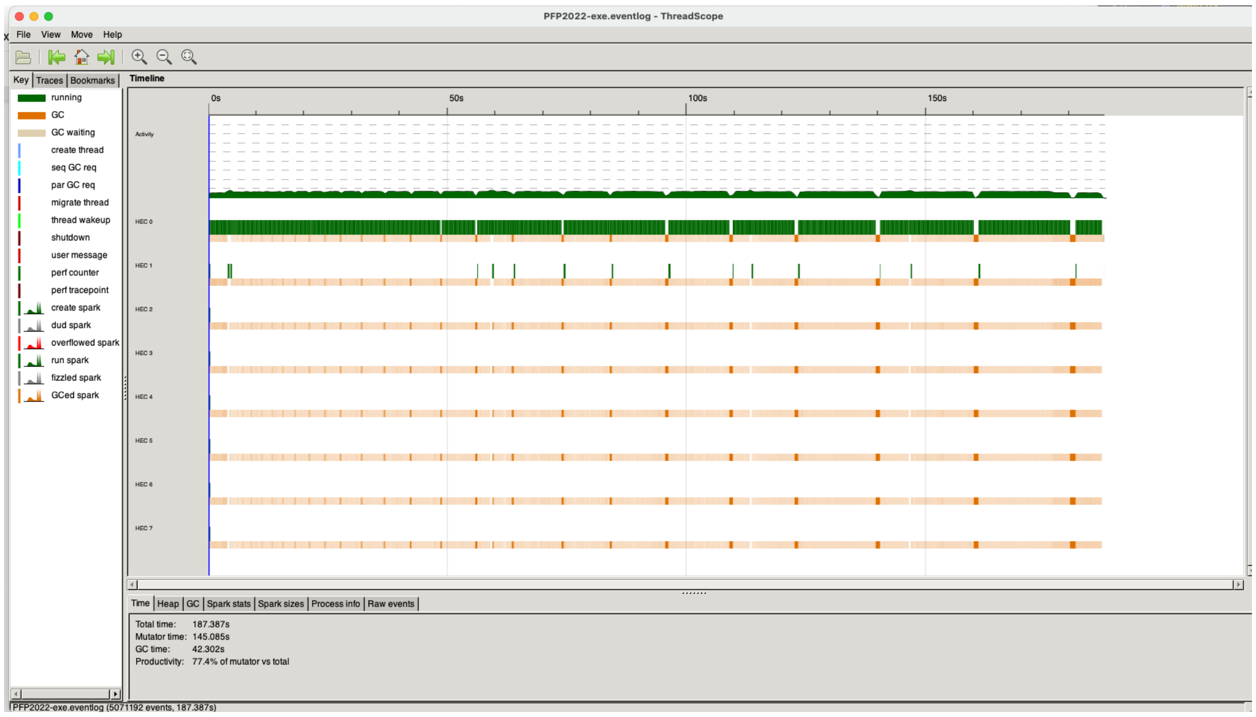
By changing including the parMaps above, it took our parallel program around 60s to predict “yelling” for, “he was y”. This represents about a 3X speedup over the non sequential version overall!



```
Total time:    64.833s
Mutator time:  45.154s
GC time:       19.680s
Productivity:  69.6% of mutator vs total
```

```
PFP2022-exe.eventlog (1414076 events, 64.833s)
```

Figure 7: Parallel implementation



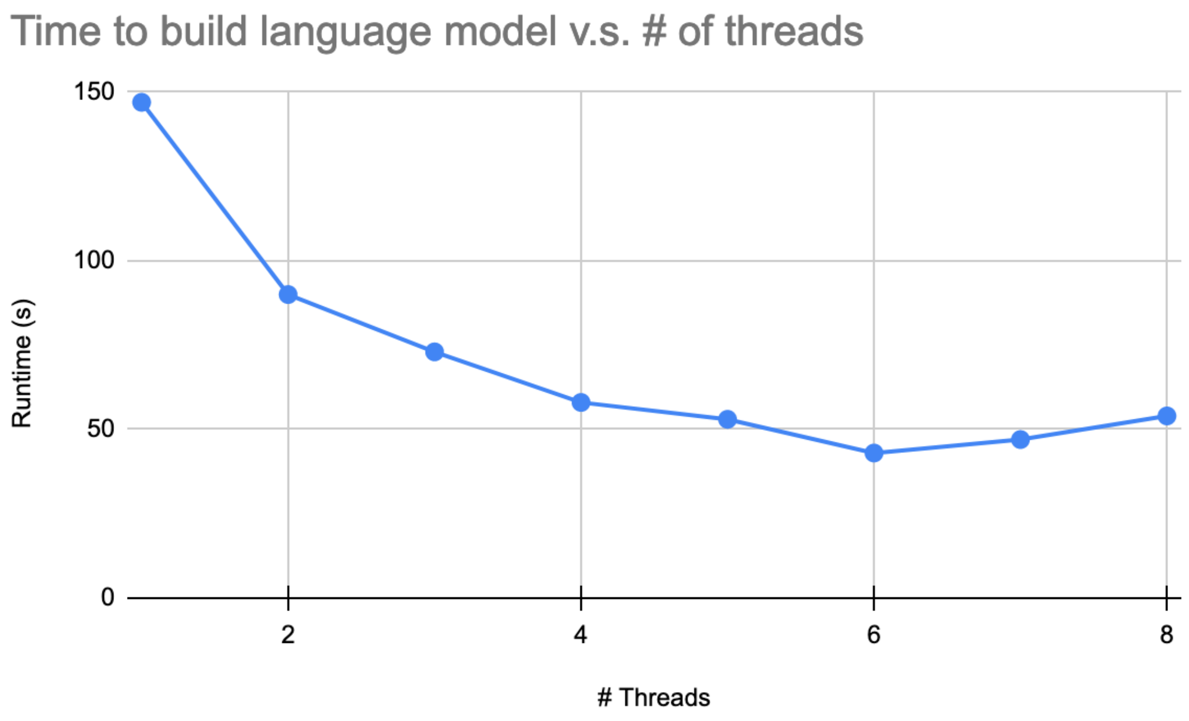
```
Total time:    187.387s
Mutator time:  145.085s
GC time:       42.302s
Productivity:  77.4% of mutator vs total
```

```
PFP2022-exe.eventlog (5071192 events, 187.387s)
```

Figure 8: Sequential Implementation: took a whopping 187s to predict “yelling” for, “he was y”.

Overall performance - # of HEC's vs runtime

Given our overall program - i.e, ingest the corpus, build the language model, and make 1 prediction - we measured the performance for different numbers of threads of execution. Overall, we found a maximum of a ~3x improvement by using 6 cores, on a 2021 Apple M1 MacBook Pro with 8 cores (6 “performance”, 2 “efficiency”).



We see that performance increases to a maximum of 6 threads. We predict that this is either because the 2 “efficiency” cores on the MacbookPro suck, or other processors where being used by other programs (e.g. Firefox, Spotify, Slack, etc.). We didn’t bother trying with over -N8 as the GHC documentation strongly suggests against asking GHC to create more threads than the physical capabilities of the machine[2] anyways.

[2] https://downloads.haskell.org/ghc/latest/docs/users_guide/using-concurrent.html#rts-flag--maxN%20%E2%9F%A8x%E2%9F%A9

Sample Predictions

```
spongebob s → squarepants
return of the j → jedi
hotel ca → california
java is a p → problem
programmers are very b → bitter
very b → bad
columbia u → university
fight for your r  → rights
rebel without a c → cause
my favorite sport is b → basketball
```

How to Run

Please retrieve the code, including the corpus collections, from our GitHub repository at https://github.com/cbass1127/PFP2022.git. We recommend trying the program w/ bcorp first as it's smaller and took ~ 60s to build on our machine. fcorp took ~ 130s to build the language mode.

```
git clone git@github.com:cbass1127/PFP2022.git
```

When opening the tarball from CourseWorks, you may need to run stack init before the following steps:

```
stack init
```

compile[3] the code with

```
stack build
```

To obtain the path to the generated executable, run the command:

```
stack exec  -- which PFP2022-exe
```

Now, we can run our project on 8 cores (assuming bcorp is in your working directory) with the command below. The -l flag will also generate a .eventlog file to be used with Threadscope.

[3] We used LTS Haskell 19.23 (ghc-9.0.2)

```
/Users/sebastianhereu/Desktop/haskell/PFP2022/.stack-work/install/x86_64-os
x/2ab99416df1ad388088601365117d1e2769e99a1b2ed03357ed91c390ee86786/9.0.2/bi
n/PFP2022-exe bcorp +RTS -N8 -l
```

Note the path to the `PFP2022-exe` executable will be different for you.

Code Listing

Lib.hs

```haskell
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use tuple-section" #-}
module Lib
   ( parseCorpus,
   computeNGramFrequenciesPar,
   computeNGramFrequencies,
   docDelim,
   elimDots,
   tokeniseDoc,
   isSafeChar,
   prefixNodePar,
   insertTrie,
   createNGramMapPar,
   createNGramMap,
   getPrediction,
   computeNGrams,
   computeNGramsPar,
   prefixSearchSeq,
   buildTree,
   makeForest,
   listDir,
   drive,
   Trie (..)
   ) where

import qualified Data.Text as T
import qualified Data.ByteString.Char8 as B
import Data.Text.Encoding as E (decodeUtf8)
import GHC.Unicode as U ( isSpace, isAlpha, toLower )
import Control.Parallel.Strategies as S
import qualified Data.Map.Strict as M
import Data.List.Split as Split
import Data.List (sortOn)
import Control.DeepSeq as DS
import System.Directory
   ( getCurrentDirectory, getDirectoryContents, setCurrentDirectory )
import System.Directory.Internal.Prelude (getArgs)
import System.Exit(die)
import System.Environment (getProgName)
import qualified Data.Text.Lazy as TL
import qualified Data.Text.Lazy.IO as TIO
import Data.Time

-- Data Constructors --

data Trie = Node Bool Int (M.Map Char Trie) | Empty deriving (Eq, Read)

instance Show Trie where
       show (Node bool count tmap) = "("++show count++") " ++ show bool ++ ": " ++ show tmap
       show _ = error "empty Trie"

instance NFData Trie where
       rnf Empty = ()
       rnf (Node _ _ mp) = DS.rnf mp `seq` ()

-- Types --

type Token = T.Text
type NGram = [Token]
type Line = [Token]
type Document = [Line]
type Corpus = [Document]

-- Constants --

docDelim :: T.Text
docDelim = T.pack "@@"

lineDelim :: T.Text
lineDelim = T.pack " . "

space :: T.Text
space = T.pack ""

threeDots :: T.Text
threeDots = T.pack ". . ."

fourDots :: T.Text
fourDots = T.pack ". . . ."

startDelim :: T.Text
startDelim = T.pack "<s/>"

smooth :: Int
smooth = 13

gramWeight:: Double
gramWeight = 5
```

```haskell
-- Utility Functions --

splitInto :: Int -> [e] -> [[e]]
splitInto n xs = Split.chunksOf ((ceiling :: Double -> Int) (fromIntegral ( length xs `div`
n)))  xs

delLast :: [a] -> [a]
delLast []     = error "Empty list!"
delLast [_]    = []
delLast (h:t)  = h : delLast t

listDir:: String -> IO[String]
listDir path = do
   setCurrentDirectory path
   cd <- getCurrentDirectory
   putStrLn ("Entering directory at: " ++ show cd)
   getDirectoryContents cd

lastN :: Int -> [a] -> [a]
lastN n xs = foldl (const . drop 1) xs (drop n xs)

addScores :: (Token, Double) -> (Token, Double) -> (Token, Double)
addScores (a, s1) (_, s2) = (a, (gramWeight * s1) + s2)

-- Parsing, Cleaning, and Tokenizing functions --

isSafeChar :: Char -> Bool
isSafeChar c = isAlpha c || isSpace c || c == '@' || c == '-' || c == '.'

elimDots :: T.Text -> T.Text
elimDots t = T.replace fourDots space (T.replace threeDots space t)

tokeniseDoc :: T.Text -> Document
tokeniseDoc t = map T.words (T.splitOn lineDelim t)

parseCorpus:: B.ByteString -> Corpus
parseCorpus file = do
       let cleaned_corpus = decodeUtf8 (B.map toLower file)
       let docs = T.splitOn docDelim cleaned_corpus
       let text_docs = map elimDots docs
       S.parMap S.rdeepseq tokeniseDoc text_docs

-- Functions to generate Ngrams --

computeNGramsPar :: Int -> Corpus -> [NGram]
computeNGramsPar n corpus = let preppedDocs = map (prepareforNGram n) corpus in  concat
(concatMap (S.parMap S.rdeepseq (ngram n)) preppedDocs)

computeNGrams :: Int -> Corpus -> [NGram]
computeNGrams n corpus = let preppedDocs = map (prepareforNGram n) corpus in  concat
(concatMap (map (ngram n)) preppedDocs)
```

```haskell
prepareforNGram:: Int -> Document -> Document
prepareforNGram n = map (replicate n startDelim ++ )

ngram :: Int -> Line -> [NGram]
ngram n xs
 | n <= length xs = take n xs : ngram n (drop 1 xs)
 | otherwise = [xs]

-- Functions to generate maps of Ngram frequencies --

makeMaps :: Int -> [Corpus] -> [M.Map NGram Int]
makeMaps n  = S.parMap S.rdeepseq (createNGramMapPar n)

searchMaps :: NGram -> [M.Map NGram Int] -> Int
searchMaps gram maps = foldl (+) smooth (S.parMap S.rdeepseq (M.findWithDefault 0 gram)
maps)

computeNGramFrequencies :: Int -> Corpus -> M.Map NGram Int
computeNGramFrequencies n corp = let tupCounts = map (\x -> (x, 1::Int)) (computeNGrams n
corp)
                                     histogram = M.toList $ M.fromListWith (+) tupCounts
                                     in M.fromList histogram

computeNGramFrequenciesPar :: Int -> Int -> Corpus -> M.Map NGram Int
computeNGramFrequenciesPar nochunks n corp = let chunks = splitInto nochunks corp in foldl
(M.unionWith (+)) M.empty (S.parMap S.rdeepseq (computeNGramFrequencies n) chunks)

createNGramMapPar :: Int -> Corpus -> M.Map NGram Int
createNGramMapPar n corp = let mps = S.parMap S.rdeepseq (flip (computeNGramFrequenciesPar
8) corp) [1..n] in foldl (M.unionWith (+)) M.empty mps

createNGramMap :: Int -> Corpus -> M.Map NGram Int
createNGramMap n corp = let mps = map (`computeNGramFrequencies` corp) [1..n] in foldl
(M.unionWith (+)) M.empty mps

-- Functions for Tries --

makeForest :: [String] -> IO [Trie]
makeForest filteredDirs = do
                        let docs = map TIO.readFile filteredDirs
                        let tok = TL.splitOn (TL.pack " ")
                        let fn =  (return::(a -> IO a)) (tok.TL.toLower) -- lowercase
                        let ws = map (fn <*>) docs
                        sequence (S.parMap S.rpar (return buildTreeT <*>) ws)

generateMatches ::  String -> [Trie] ->  [(String,Int)]
generateMatches sent tries = concat $ S.parMap S.rdeepseq (prefixNodePar 0 sent "") tries

insertTrieT :: TL.Text -> Trie -> Trie
insertTrieT _ Empty = error "insert into empty Trie"
insertTrieT txt (Node bool count tmap)
       | txt == TL.empty = Node True (count + 1) tmap
       | M.member x tmap = let map' = M.insert x (insertTrieT xs (tmap M.! x)) tmap in (Node
bool count map')
       | otherwise = let map' = M.insert x (insertTrieT xs (Node False 0 M.empty)) tmap in
(Node bool count map')
       where
               x = TL.head txt
               xs = TL.tail txt

insertTrie :: String -> Trie -> Trie
insertTrie _ Empty = error "insert into empty Trie"
insertTrie (x:xs) (Node bool count tmap)
       | M.member x tmap = let map' = M.insert x (insertTrie xs (tmap M.! x)) tmap in (Node
bool count map')
       | otherwise = let map' = M.insert x (insertTrie xs (Node False 0 M.empty)) tmap in
(Node bool count map')
insertTrie [] (Node _ count tmap) = Node True (count + 1) tmap

prefixNodePar :: Int -> String -> String -> Trie -> [(String, Int)]
prefixNodePar depth [] scat (Node isEnd count children) = prefixSearchPar depth (scat, (Node
isEnd count children))
prefixNodePar depth (x:xs) scat (Node _ _ children)
       | M.member x children = prefixNodePar depth xs (x:scat) (children M.! x)
       | otherwise = []
prefixNodePar _ _ _ _ = error "args"


prefixSearchSeq::(String, Trie) -> [(String,Int)]
prefixSearchSeq (scat, (Node isEnd count children)) = let childList = zipWith (\a b -> ((fst
a):b, snd a) ) (M.toList children) (replicate (length children) scat)
                                                         curr = if isEnd then scat else ""
                                                         in (curr, count):concatMap
prefixSearchSeq childList
prefixSearchSeq _ = error "args"

prefixSearchPar:: Int -> (String, Trie) -> [(String,Int)]
prefixSearchPar depth (scat, Node isEnd count children)
       | depth > 0 = let childList = zipWith (\a b -> (fst a:b, snd a) ) (M.toList children)
(replicate (length children) scat)
                         curr = if isEnd then scat else ""
                         in (curr, count): (concat $ S.parMap S.rseq (prefixSearchPar (depth
- 1)) childList)
       | otherwise = prefixSearchSeq (scat, (Node isEnd count children))
prefixSearchPar _ _ = error "empty"

buildTree :: [String] -> Trie
buildTree = foldl (flip insertTrie) (Node True 0 M.empty)

buildTreeT :: [TL.Text] -> Trie
buildTreeT = foldl (flip insertTrieT) (Node True 0 M.empty)


-- Functions to generate predictions --
```

```
getPrediction :: String -> [M.Map NGram Int] -> [Trie] -> String
getPrediction [] _ _ = "Error: invalid input"
getPrediction sent mps tries = fst $ last' (sortOn snd scores) where
       sentence = map T.pack (words sent)
       curNgram = delLast sentence
       lst = last sentence
       guesses = map (\x -> (T.pack $ reverse $ fst x, snd x)) (generateMatches (T.unpack
lst) tries)
       scores = S.parMap S.rdeepseq (\x -> (T.unpack $ fst x, snd x)) (map (getScore
curNgram mps) guesses)

last' :: [(String, Double)] -> (String, Double)
last' [] = ("Error: Unable to complete word", 0.0)
last' [x] = x
last' (_:xs) = last' xs


getScore :: NGram -> [M.Map NGram Int] -> (Token, Int) -> (Token, Double)
getScore curNgram mps token
       | length curNgram > 2 = let twoGramScore = score (lastN 2 curNgram) mps token
                                   nGramScore =  score curNgram mps token
                                   in addScores nGramScore twoGramScore
       | otherwise = score curNgram mps token

score :: NGram -> [M.Map NGram Int] -> (Token, Int) -> (Token, Double)
score curNgram mps (guess, _) = let freq = searchMaps (curNgram ++[guess]) mps in (guess,
fromIntegral freq / fromIntegral (searchMaps curNgram mps))

-- Driver Functions --


makeCorpi :: [String] -> IO [Corpus]
makeCorpi filteredDirs = do
                       let docs = map B.readFile filteredDirs
                       let fn = (return::(a -> IO a)) parseCorpus
                       -- let fn =  (return::(a -> IO a)) (words.(map toLower.filter
isSafeChar))
                       let corpi = S.parMap S.rpar (fn <*>) docs
                       sequence corpi

drive :: Int -> IO ()
drive n  = do
       args <- getArgs
       dirName <- case args of
               [fn] -> return fn
               _ -> do pn <- getProgName
                       die $ "Usage: "++pn++" <corpus-directory>"

       dirs <- listDir dirName -- directory of files
       let filteredDirs = filter (\x -> x /= "." && x /= "..") dirs
       start <- getCurrentTime
       tries <- makeForest filteredDirs
       corpi <- makeCorpi filteredDirs

       let maps = makeMaps n corpi
       let prediction = getPrediction "he was y" maps tries -- dummy prediction to force
language model to be built fully
       end <- prediction `deepseq` getCurrentTime

       putStrLn ("language model built successfully in " ++ show (diffUTCTime end start))

       _ <- printPredictions maps tries
       return ()

printPredictions :: [M.Map NGram Int] -> [Trie] -> IO b
printPredictions maps tries = do
       putStrLn "enter a prefix: "
       prefix <- getLine
       -- putStrLn ("prediction for: " ++ prefix)
       let prediction = getPrediction prefix maps tries
       putStrLn prediction
       printPredictions maps tries
```

Main.hs

```
module Main (main) where

import Lib

main:: IO()
main = drive 5 -- up to of n-grams to use
```

Package.yaml

```
name:                PFP2022
version:             0.1.0.0
github:              "githubuser/PFP2022"
license:             BSD3
author:              "Author name here"
maintainer:          "example@example.com"
copyright:           "2022 Author name here"

extra-source-files:
- README.md
- CHANGELOG.md

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web
```

```yaml
# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:         Please see the README on GitHub at
<https://github.com/githubuser/PFP2022#readme>

dependencies:
- base >= 4.7 && < 5
- parallel
- text
- bytestring
- split
- containers
- directory
- mtl
- deepseq
- directory
- time

ghc-options:
- -Wall
- -Wcompat
- -Widentities
- -Wincomplete-record-updates
- -Wincomplete-uni-patterns
- -Wmissing-export-lists
- -Wmissing-home-modules
- -Wpartial-fields
- -Wredundant-constraints
- -eventlog

library:
 source-dirs: src

executables:
 PFP2022-exe:
   main:                Main.hs
   source-dirs:         app
   ghc-options:
   - -threaded
   - -rtsopts
   - -with-rtsopts=-N

   dependencies:
   - PFP2022

tests:
 PFP2022-test:
   main:                Spec.hs
   source-dirs:         test
   ghc-options:
   - -threaded
- -rtsopts
- -with-rtsopts=-N
dependencies:
- PFP2022
```