Parallel Autocomplete

Eugene Kim
ek3192

Report

As outlined in my proposal, this program parses a corpus of English text for word counts and provides at most K autocomplete suggestions by descending frequency to each input string given by the user.

I followed the Details, Parallelization, Evaluation, and Data sections of my proposal. Please see the README.md for execution details.

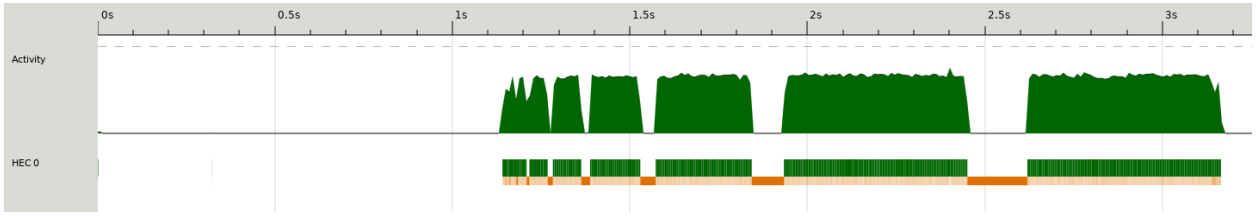
Results

These results were gathered on a quad-core machine.

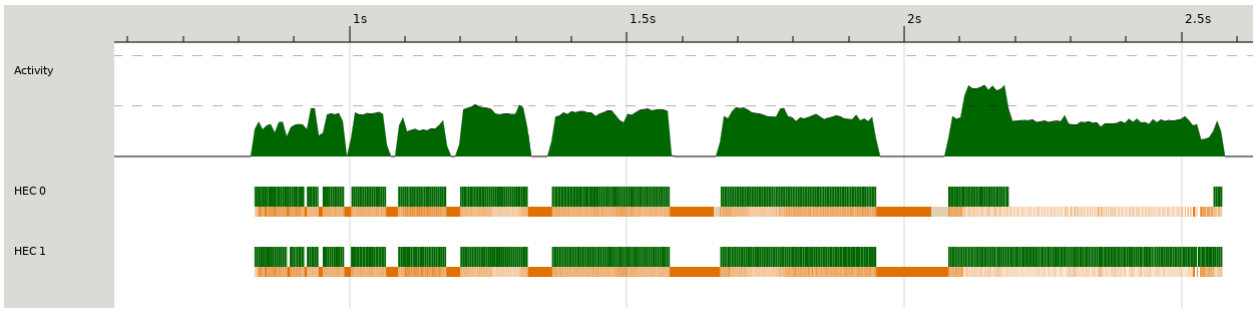
Step 1

The first step for evaluation is seeing how the sequential and parallelized program ran in regards to step 1 of the program, creating the trie from word counts gathered from the corpus. The sequential program sequentially counts the words of the corpus, whereas the parallel program chunks the corpus and performs Map/Reduce. I used the Shakespeare corpus 100-0.txt from HW4, which comprises approximately 1,000,000 words.

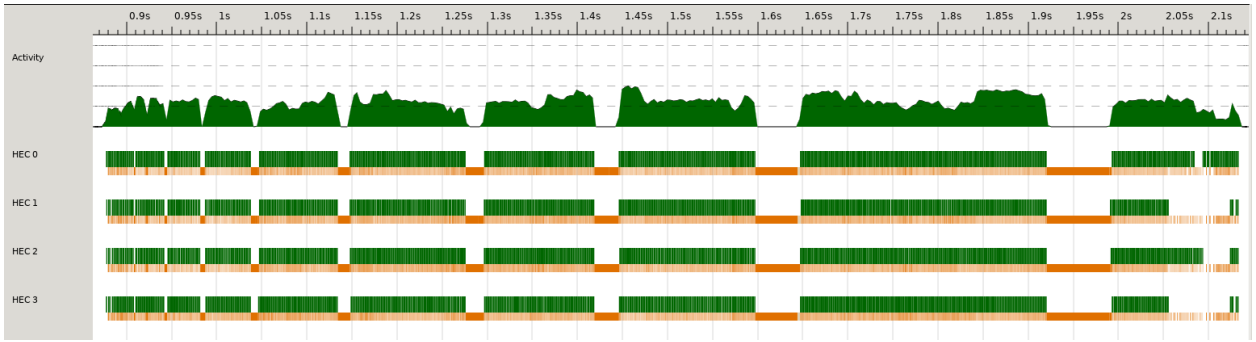
Sequential - 2.05 s:



Parallel N2 - 1.75 s:



Parallel N4 - 1.22 s:



Parallel N8 - 4.17 s:

# of Cores utilized	1	2	4	8
Time elapsed	2.05 s	1.75 s	1.22 s	4.17 s

We see a sort of “bathtub” distribution. Asking multiple cores to help handle the work helps speed the program to a certain extent. However, after a point, this adds more bookkeeping overhead that exceeds the diminishing marginal gains from parallelism.

There is still performance left to be desired, as the activity row at the top shows a total utilization value of 2 cores at best. There is a bit of garbage-collection going on which I tried to minimize by using parBuffer. There are also unavoidable sequential operations, like disk IO for the large corpus, reconstructing the count map and lists. Also, the initial population of the trie remained sequential in both program versions, as there can be race conditions if done in parallel.

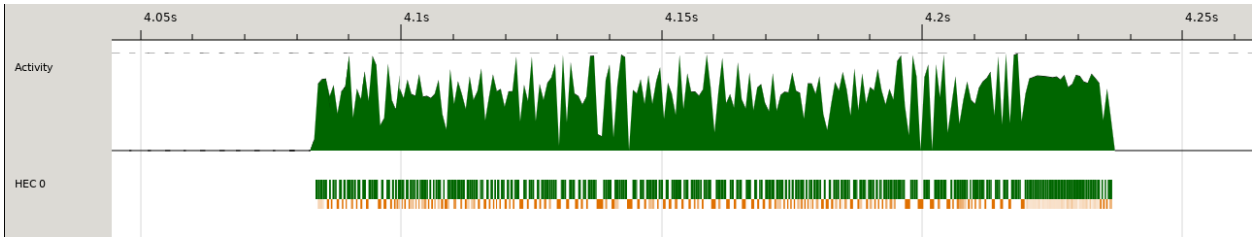
Step 2

The second step for evaluation is seeing how the sequential and parallelized program ran in regards to step 2 of the program, providing K most relevant autocomplete suggestions to the

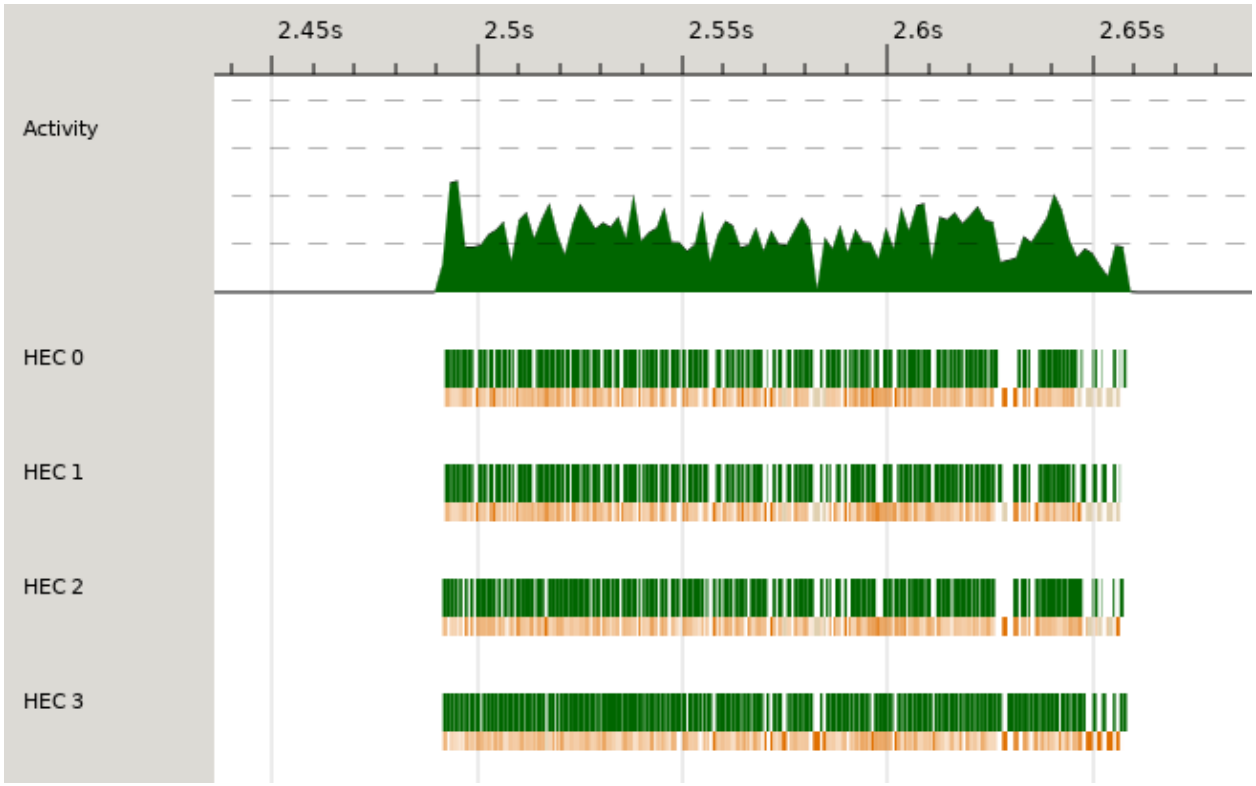
user's prompt. Both programs traverse the populated trie using the prompt. The sequential program performs DFS on all possible descendents sequentially, whereas the parallel program performs DFS on all possible descendents of each node in parallel with each other. Afterwards, both programs sort by descending word count and return the top K candidates in frequency.

The most differentiating input case would be the empty string, since all words in the corpus are potential autocomplete candidates.

Sequential - 0.15 s:



Parallel N4 - 0.16 s:



The time elapsed was very small for DFS across the entire trie, and was even smaller for all other queries. As you can see, the time elapsed for both sequential and parallel were very similar, despite the parallelization graph's activity row showing at best ~2 core utilization value. This was also the case for many other test inputs and # of cores, all varying in the possible number of autocompletes. I have tried numerous parallelization approaches for step 2, including parBuffer + rdeepseq (current implementation), parList + rseq, using bang (!) to force eager evaluation, etc. however they did not seem to produce meaningful speedup.

A potential theory I have as for why this is the case is that there is an inherently sequential nature to a graph traversal algorithm like DFS through a tree. The way I have implemented DFS requires the program to fully traverse to the leaf nodes of the trie in order to terminate. The reason for this is that it is somewhat difficult to synchronize an early termination in the case there are parallel traversals going on. Also, the nature of my autocomplete program is such that it wants to find the most relevant candidates to suggest, and the most popular words may be stored deep within the trie. Also, the DFS traversal doesn't benefit from "smart" memoization of results gathered in parallel like a fibonacci program would, and so the total expected work in either implementation would be expected to be similar.

Conclusion

Parallelization has been a successful means to speeding up a prototypical autocomplete program by utilizing a Map/Reduce approach to gathering word counts of a corpus to populate a trie. It has been difficult to show meaningful speedup in the trie DFS traversal however.

Some future improvements would involve utilizing parallel file IO with Haskell's POSIX-style IO to help reduce the notorious disk IO sequential bottleneck.

References

Trie data structure
https://gist.github.com/orclev/1929451

Map/Reduce
https://stackoverflow.com/questions/27115894/haskell-parallel-word-count-using-mapreduce-framework-control-parallel-strate

Code

app/Main.hs

```
module Main (main) where

import Lib
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import Data.Char(isAlpha, toLower, isSpace)
import Data.Map(fromListWith, toList, unionsWith)
import Data.List(sortBy)
import Control.Parallel.Strategies(runEval, parBuffer, rdeepseq, parMap, rpar)

main :: IO ()
main = do args <- getArgs
          case args of
```

```
            [filename, k, version] -> do
              if version == "seq"
                then do
                  contents <- readFile filename
                  let cWords = zip (getWords contents) $ repeat 1
                      counts = toList $ fromListWith (+) cWords
                      trie = initTrie counts
                  autocomplete trie k
              else if version == "par"
                then do
                  contents <- readFile filename
                  let pWords = getWords contents
                      pairs = runEval ((parBuffer 100 rdeepseq) (map (\x -> (x,
1)) pWords))
                      counts = toList $ unionsWith (+) (parMap rpar
(fromListWith (+)) (chunk 200000 pairs))
                      trie = initTrie counts
                  parAutocomplete trie k
              else do
                pn <- getProgName
                die $ "Usage: " ++ (fst $ span (/= '.') pn) ++ " <filename> <k>
<seq|par>"
            _ -> do pn <- getProgName
                    die $ "Usage: " ++ (fst $ span (/= '.') pn) ++ " <filename>
<k> <seq|par>"

getWords :: [Char] -> [String]
getWords = words . map toLower . filter (\c -> isAlpha c || isSpace c)

autocomplete :: Trie -> String -> IO b
autocomplete trie k = do putStrLn "Please enter a query: "
                         line <- getLine
                         let node = traverseTrie line trie
                             counts = dfs node line
                             sorted = sortBy (\(_,a) (_,b) -> compare b a)
counts
                         mapM_ (putStrLn . fst) $ take (read k :: Int) sorted
                         autocomplete trie k

parAutocomplete :: Trie -> String -> IO b
parAutocomplete trie k = do putStrLn "Please enter a query: "
                            line <- getLine
                            let node = traverseTrie line trie
                                counts = parDFS node line
                                sorted = sortBy (\(_,a) (_,b) -> compare b a)
```

```
counts
                            mapM_ (putStrLn . fst) $ take (read k :: Int)
sorted
                            parAutocomplete trie k

chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (as,bs) = splitAt n xs in as : chunk n bs
```

src/Lib.hs

```
module Lib (Trie, initTrie, traverseTrie, dfs, parDFS) where

import qualified Data.Map as M
import Control.Parallel.Strategies(runEval, parBuffer, rdeepseq)

data Trie = Node Int (M.Map Char Trie)

insert :: [Char] -> Int -> Trie -> Trie
insert [] count (Node n m) = Node (n + count) m
insert (x:xs) count (Node n m) =
    case M.lookup x m of
        Nothing -> Node n (M.insert x (insert xs count (Node 0 M.empty)) m)
        Just child -> Node n (M.insert x (insert xs count child) m)

initTrie :: Foldable t => t ([Char], Int) -> Trie
initTrie counts = foldl (\t (w, c) -> insert w c t) (Node 0 M.empty) counts

traverseTrie :: [Char] -> Trie -> Trie
traverseTrie [] node = node
traverseTrie (x:xs) (Node _ m) =
    case M.lookup x m of
        Nothing -> (Node 0 M.empty)
        Just child -> traverseTrie xs child

dfs :: Trie -> [Char] -> [([Char], Int)]
dfs (Node n m) suffix
  | n > 0 = foldl (\l (letter, child) -> l ++ (dfs child (suffix ++ [letter])))
[(suffix, n)] pairs
  | otherwise = foldl (\l (letter, child) -> l ++ (dfs child (suffix ++
[letter]))) [] pairs
  where pairs = M.toList m

parDFS :: Trie -> [Char] -> [([Char], Int)]
parDFS (Node n m) suffix
```

```
  | n > 0 = runEval ((parBuffer 100 rdeepseq) (foldl (\l (letter, child) -> l
++ (parDFS child (suffix ++ [letter])))) [(suffix, n)] pairs))
  | otherwise = runEval ((parBuffer 100 rdeepseq) (foldl (\l (letter, child) ->
l ++ (parDFS child (suffix ++ [letter])))) [] pairs))
  where pairs = M.toList m
```