Parallel MapReduce Applications in Haskell

Jeremy Carin

Prelude

The original goal of my project was to implement the PageRank algorithm in parallel, as described Larry Page and Sergey Brin in their prototype research project called “Google”. PageRank uses the link structure of the web to measure the importance of web pages. The idea behind this approach is that a web page with many high-quality incoming links from other reputable web pages is likely to be more relevant and useful than a web page with fewer or lower-quality incoming links.

PageRank makes use of MapReduce, a framework to do distributed parallelizable computations on a large number of machines. My implementation would scale across cores of a single machine, rather than multiple machines.

MapReduce Implementation

I first wrote a parallel MapReduce function using rdeepseq, so that each iteration fully evaluates its argument. The generic MapReduce is as follows:

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

mapReduce is a higher-order function that takes three arguments: a mapper function mapper, a reducer function reducer, and a list of inputs input. It applies the mapper function to each element in the input list in parallel using the parMap function from the Control.Parallel library and the rpar and rdeepseq combinators. The rpar combinator indicates that a computation can be run in parallel, while rdeepseq forces evaluation of the result of the computation, ensuring that the computation is fully evaluated before being passed to the reducer function.

The result of the parMap function is then passed to the reducer function, which is applied using the using function from the Control.Parallel.Strategies library. The using function allows you to specify an evaluation strategy to use when applying the reducer function. In this case, the rseq strategy is used, which evaluates the computation to weak head normal form and returns it.

Finally, the result of the reducer function is passed to the pseq function, which fully evaluates the result of the computation before returning it.

PageRank Implementation

PageRank works by defining a probability distribution over the nodes in the graph, with the probability of a node being proportional to the number and quality of links to and from the node. The PageRank value of a node is then calculated as the expected value of the probability distribution over the nodes.

To calculate the PageRank values of the nodes in a graph, the PageRank algorithm performs the following steps:

1. Initialize the PageRank values of all the nodes in the graph to an equal value.
2. Iteratively update the PageRank values of the nodes based on the PageRank values of the nodes that link to them, using a damping factor to dampen the effect of the update. The damping factor is a constant value between 0 and 1 that is used to control the convergence of the PageRank values.
3. Repeat step 2 until the PageRank values converge or a maximum number of iterations has been reached.
4. In each iteration, the PageRank values of the nodes are updated using the following formula:

   PR(u) = (1 - dampingFactor) / |V| + dampingFactor * (PR(v) / C(v))

where:

- PR(u) is the PageRank value of node u.
- |V| is the total number of nodes in the graph.
- dampingFactor is the damping factor.
- PR(v) is the PageRank value of node v.
- C(v) is the number of out-edges

I define the following types to be used in the PageRank algorithm,

```
type PRVal = Double
type PRValues = M.Map Node PRVal
type Node = String
type InEdges = M.Map Node [Node]
type OutEdges = M.Map Node [Node]
```

The types are somewhat self-explanatory, but PRValues represents the outcome of the PageRank algorithm, the InEdges and OutEdges represent the graph itself.

I implement the mapper and reducer as such:

```
mapper :: (PRVal, [Node]) -> PRValues
mapper (pr, outNodes) =
  let pr_ = pr / fromIntegral (length outNodes)
  in M.fromList $ zip outNodes (repeat pr_)

reducer :: [PRValues] -> PRValues
reducer = foldr (M.unionWith (+)) M.empty
```

Complications

Unfortunately, I was unable to fully parallelize my PageRank algorithm, and I'm still not sure what went wrong. Originally, I suspected the garbage collector had too high of an overhead, as there were fluctuations in CPU usage that matched when the GC was doing work. Disabling the parallel GC produces similar results, and continues inconsistent usage across cores. Figures 1 and 2 compare the Parallel vs Sequential GC. Figure 3 shows the relatively small speedup when using more cores, which more clearly demonstrates how parallelism does not benefit my algorithm. I am left to assume that there is a fundamental flaw in my approach which did not allow full parallelism, as such I decided to approach a different problem

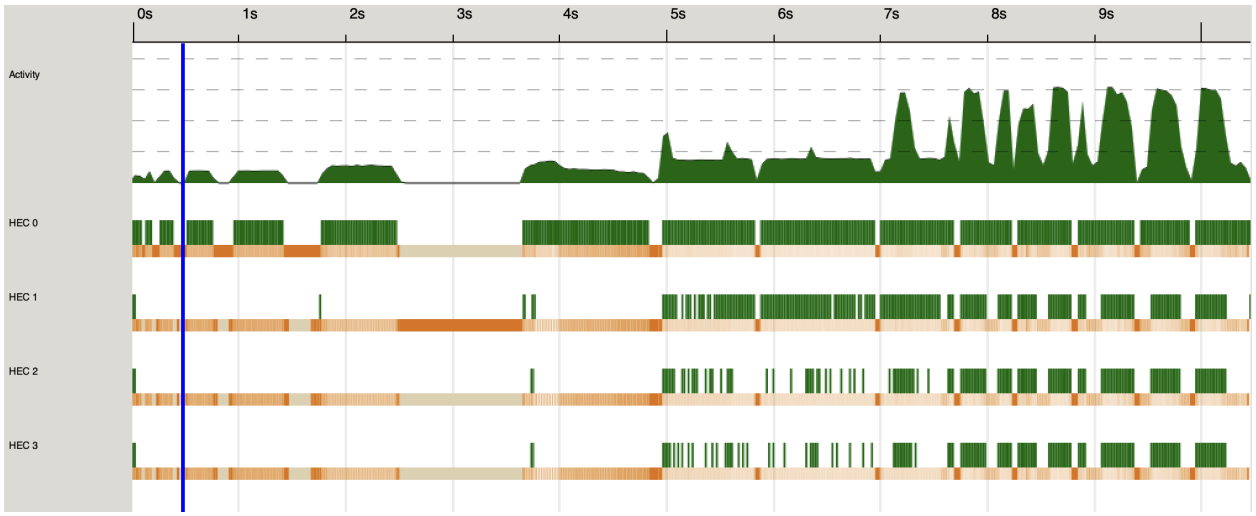

Figure 1 – Parallel GC

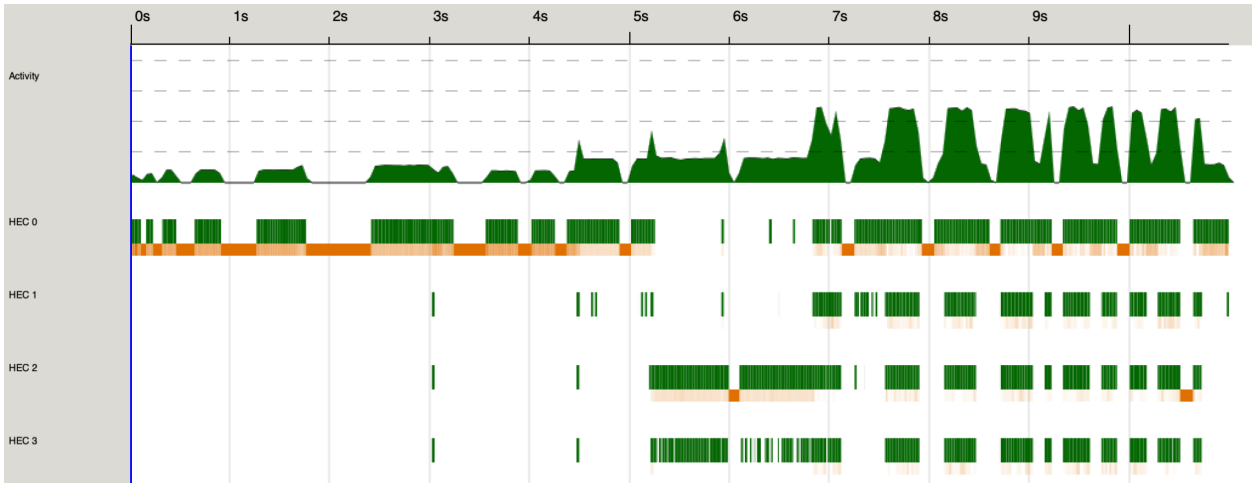

Figure 2 – Sequential GC

HEC count	time (s)	Speedup vs 1 core
1	12.729	–
2	12.305	3.3%
3	11.128	12.8%
4	10.608	16.7%

Figure 3 – Core count and execution time

Onward

As I was unable to get PageRank fully parallelized, I took my unassuming yet mighty MapReduce generic function and applied it to different problems. As somewhat of a sanity check and precursor to my larger program, I implemented a word count program. I used the same MapReduce function as before, but created new functions to actually do the mapping and reducing.

I also implemented a sequential version, which would work essentially the same way as running the parallel version with 1 core. The only difference is the MapReduce function, which is done sequentially now rather than with parallelism.

```
seqMapReduce :: (a -> b) -> ([b] -> c) -> [a] -> c
seqMapReduce mapper reducer input = reduceResult
  where mapResult = map mapper input
        reduceResult = reducer mapResult
```

I implement mapper and reducer functions as such:

```
mapper :: String -> M.Map String Int
mapper = getWordFreqMap . getWords

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

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

reducer :: [M.Map String Int] -> [(String, Int)]
reducer = M.toList . foldl (M.unionWith (+)) M.empty
```

My word count program is able to scale to extremely large sizes. To show this, I provide a test script that downloads a 100 MB corpus of information, split into a number of different files. Figures 6, 7 and 8 refer to a slightly different dataset, but demonstrate a similar scaling and show effective parallelism. The sequential version performs essentially the same as a core count of 1.

HEC count	time (s)	Speedup vs 1 core
1	63.798	–
2	42.591	33%
3	29.334	54%
4	18.608	71%

Figure 4 – Core count and execution time

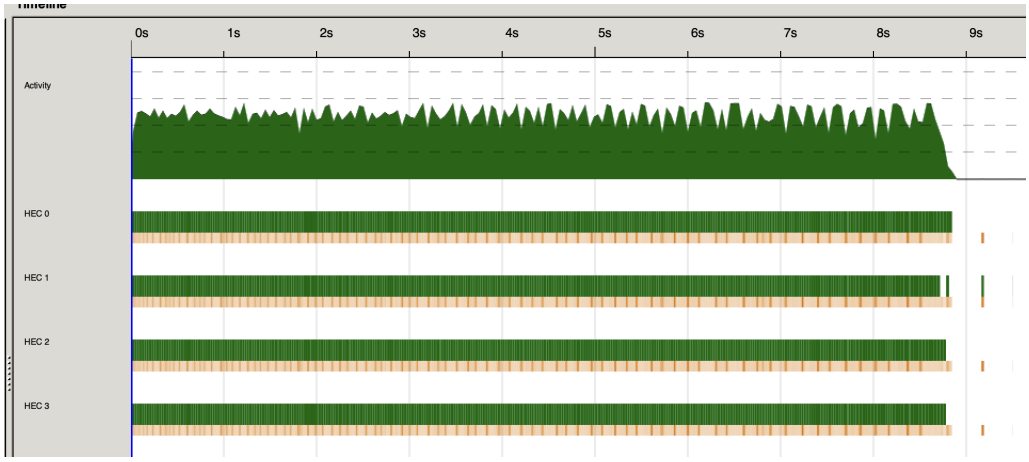

Figure 6 – 4 cores

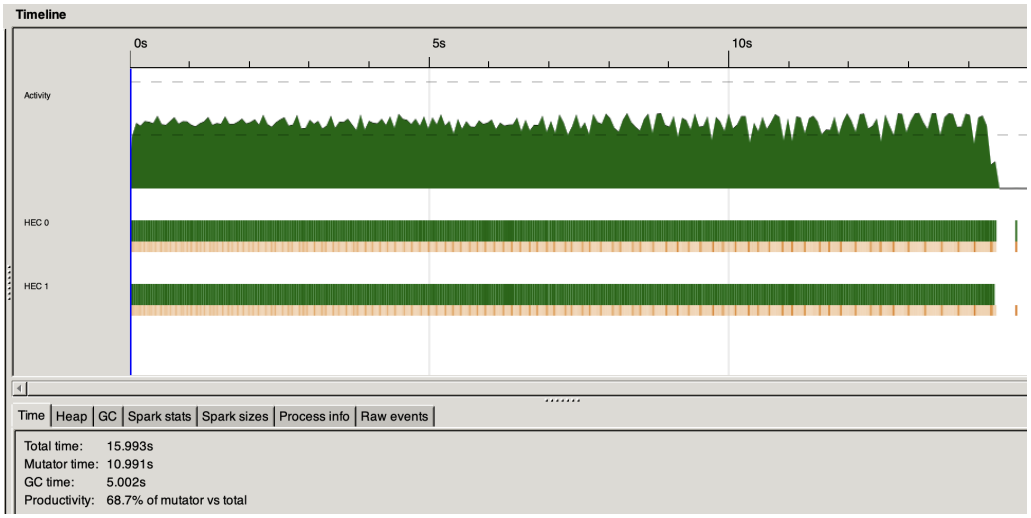

Figure 7 – 2 cores

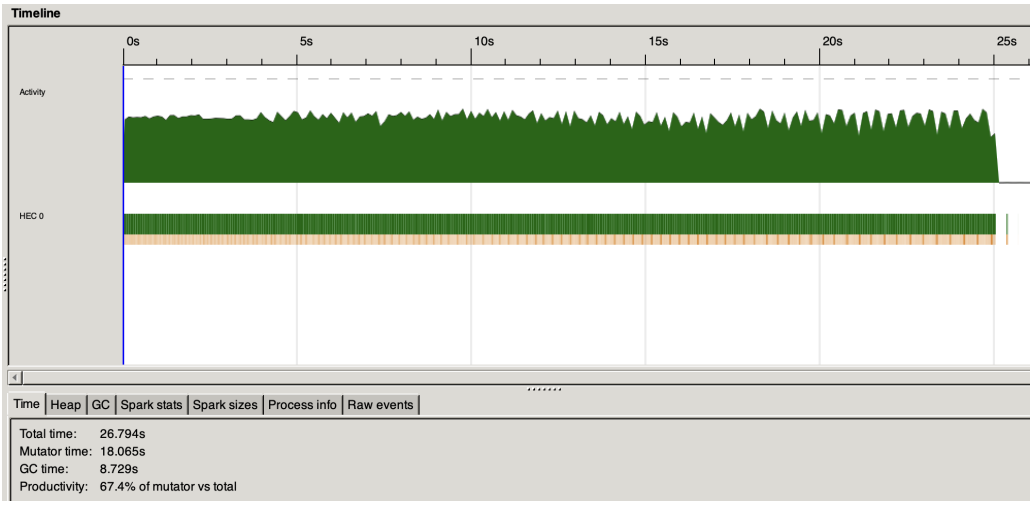

Figure 8 – 1 core

AutoComplete implementation

Using this new word count program, I was able to implement an autocomplete based on a large corpus of text. I changed the output of the word counter program so that it writes to a file, and then used the Trie data structure to structure word counts. I found that a word count file was always an order of magnitude smaller than the corpus, so there was very little parallelism that would speed up this part of the process.

An example of the program running is below:

```
Enter word:
hi
Found the following words:
1. his
2. him
3. himself
4. high
```

5. hinted
6. hidden
7. history
8. higher
9. hint
10. hitherto

Conclusion

The best laid plans of mice and men…

I'm still happy with what I created in the end, although I hope to get PageRank working at some point because it frustrated me for far too long for it to win.

Code listings:

Main.hs

```
module Main (main) where

import WordCount
import AutoComplete
import System.Environment (getArgs)
import System.Directory
import Control.Monad
main :: IO ()
main = do
  args <- getArgs
  case args of
    ["count", dir] -> wordCount dir "-"
    ["count", dir, output] -> wordCount dir output
    ["auto", dir] -> auto dir
    ["auto", dir, "new"] -> do
        fileExists <- doesFileExist ".autocomplete_out"
        when fileExists $ removeFile ".autocomplete_out"
        auto dir
    _ -> putStrLn "invalid option. specify `count [dir] (output)`,
`auto [dir]`, or `auto [dir] new`"
```

AutoComplete.hs

```
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE ScopedTypeVariables #-}

module AutoComplete (auto) where


import Data.List (sortBy)
import Data.List.Split (splitOn)

import qualified Data.ByteString.Char8 as BC
import qualified Data.Trie as T
import qualified Data.Map as M
import qualified Data.Bifunctor as B
import System.Directory
import WordCount

import Control.Exception (Exception, throw)

newtype FormatError = FormatError String
    deriving (Show, Eq)
```

```
instance Exception FormatError

loadWords :: FilePath -> IO (M.Map String Int)
loadWords fname = do
    filedata <- readFile fname
    let contents = map (\line -> let ws = splitOn "," line in
                                 case ws of
                                     [s, i] -> (s, read i :: Int)
                                     _ -> throw (FormatError line))
(lines filedata)
    let loadedMap = M.fromList contents
    return loadedMap


createTrie :: M.Map String Int -> T.Trie Int
createTrie = T.fromList . fmap (B.first BC.pack) . M.toList

autocomplete :: Ord a => T.Trie a -> IO ()
autocomplete trie = do
  putStrLn "Enter word:"
  prefix <- getLine
  let subTrie = T.submap (BC.pack prefix) trie
      topk = take 10 $ sortBy (\(_, a) (_, b) -> compare b a) $
T.toList subTrie
  if null topk
    then putStrLn "No words found"
    else do
      putStrLn "Found the following words:"
      mapM_ (\(n :: Int, (a, _)) -> putStrLn $ show n ++ ". " ++
BC.unpack a) (zip [1..] topk)
  putStrLn ""
  autocomplete trie


auto :: FilePath -> IO ()
auto directory = do
    outputExists <- doesFileExist ".autocomplete_out"
    if outputExists
        then putStrLn "Output file already exists, skipping
wordCount. run with -F to re-read files."
        else wordCount directory ".autocomplete_out"
    loadedMap <- loadWords ".autocomplete_out"
    let trie = createTrie loadedMap
    autocomplete trie
```

MapReduce.hs

```
module MapReduce (mapReduce, seqMapReduce)

where

import Control.Parallel
import Control.Parallel.Strategies


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

seqMapReduce :: (a -> b) -> ([b] -> c) -> [a] -> c
seqMapReduce mapper reducer input = reduceResult
  where mapResult = map mapper input
        reduceResult = reducer mapResult
```

WordCount.hs

```
{-# LANGUAGE TupleSections #-}

module WordCount
( wordCount
)

where

import Data.Char (isAlpha, isSpace, toLower)
import MapReduce
import System.Directory
import qualified Data.Map as M
import System.IO.Error (catchIOError)

mapper :: String -> M.Map String Int
mapper = getWordFreqMap . getWords

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

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

```
reducer :: [M.Map String Int] -> [(String, Int)]
reducer = M.toList . foldl (M.unionWith (+)) M.empty

wordCount :: FilePath -> FilePath -> IO ()
wordCount filePath output = do
    files <- listDirectory filePath
    let myFiles = filter (\x -> x `notElem` [".", ".."]) (map
(filePath ++) files)
    parsedFiles <- mapM readFile myFiles
    let freqs = mapReduce mapper reducer parsedFiles
    printFreqs freqs output


printFreqs :: [(String, Int)] -> FilePath -> IO ()
printFreqs freqs output = do
    let outputText = unlines $ map (\(word, count) -> word ++ "," ++
show count) freqs
    if output == "-"
        then putStrLn outputText
        else catchIOError (writeFile output outputText) (\_ ->
putStrLn "Error: could not write to file")
```