COMS W4995 Parallel Functional Programming Project Report
MapReduce Word Frequency Search (WFS)

Sarah Yang (sy3038)
Arush Sarda (as6785)
Patricia Luc (pbl2116)

1. Introduction

In this project, we aimed to modify and extend on the word frequency program we completed as homework. In addition to the word counter, we implemented a word search functionality which outputs the count of a user-inputted search word. MapReduce Word Frequency Search (WFS) works as follows: first, it reads in the input file specified by the program arguments. Then, it calculates word frequencies of the input file using MapReduce. Finally, it repeatedly asks the user for a search word, and if the word exists in the file, then the program prints the word and its count, otherwise, the program prints the most similar word in the input file and its corresponding count.

2. Implementation

To calculate the word frequencies, we used the MapReduce programming model[1]. MapReduce has a unique advantage in parallelism because it was designed for large amounts of data and distributed systems, where multiple machines work in parallel to analyze data. The MapReduce model is split into two phases: map and reduce. Before entering the map phase, the input data is split into chunks and assigned to each machine. In the map phase, mapper machines extract relevant data from their assigned chunk. In the reduce phase, reducer machines take their assigned map stage output and aggregates the information. In our application of MapReduce, mappers emit a list of tuples (<tokenized word>, 1), and reducers aggregate the list of tuples into a map of unique words to their count.

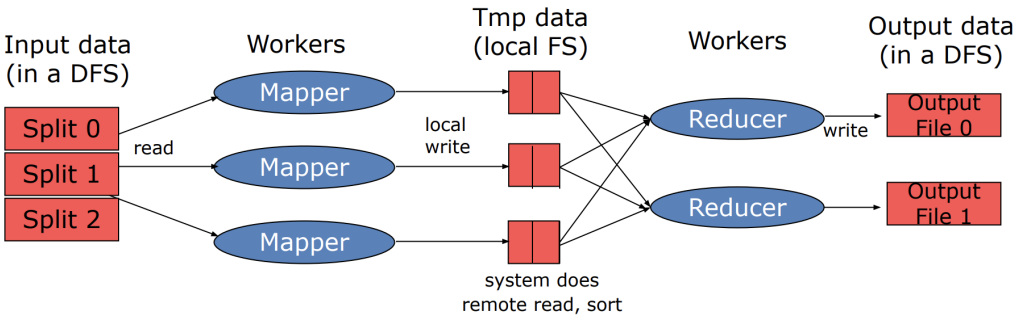

Figure 1. Workflow of MapReduce [2]

Regarding the word search feature, if the search word exists in the input file, the program finds its count by consulting the word count map. In the case that the search word does not exist in the input file, a fuzzy search algorithm using Levenshtein distance is used to find the closest matching existing word. The Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into another. In our program, the closest matching word would be the one that has the smallest distance from the user's search word.

3. Parallelization

As for how we then approached parallelizing our program, we will discuss how we first examined parallelism options in the MapReduce step and then techniques explored in the fuzzy word search step.

3.1 Parallelizing Map Reduce

For parallelizing Map Reduce, we considered two different designs: parallelizing each step of Map Reduce or parallelizing Map Reduce across chunks. To parallelize each step of Map Reduce, the file contents are split into k number of chunks in which k is the number of cores being run. All cores would then "clean" their respect chunks to find valid words in the file. To find words, all non-alphabetic characters aside from whitespace are discarded and whatever is left is treated as lowercase. Upon completion

of this step, all cores proceed to do map operations on their respective chunks. Upon completion of this step, all cores proceed to do reduce operations on their assigned chunks. Finally, one core is tasked to put the output of each reduce together into a single map. To parallelize across chunks, the file contents are split into k number of chunks as well. Each core then performs clean, map, and reduce operations on their respective chunks. Likewise, one core takes the output of each reduce from each core and aggregates them together into a complete frequency map.

We decided the latter parallelization design would be the most optimal as we did not want stages of parallelization to have to wait upon one another and run into potential straining issues with synchronization after each parallelization step. We did implement both designs and found that parallelizing each chunk rather than each step did run faster ultimately supporting our initial design choice. This is because separating the parallelization into separate steps inevitably adds unnecessary and additional overhead therefore making it perform as if it were purely sequential.

3.2 Parallelizing Fuzzy Search

We examined similar parallelization designs for the fuzzy word search portion of our program. Initially, we designed the parallel fuzzy search implementation such that the map was split into k chunks. Following, each core computed the Levenshtein distances between the target word and each word within its chunk. Upon completion, the cores would then find the minimum distance of their respective chunks. This ultimately was a design based on parallelizing each step and thus when testing later on, it wasn't leading to the speedup we'd expected. This further supported our finding that parallelizing each step separately is inefficient and potentially adds overhead. In the final version, each core calculates the distances of the words in its chunk against the target word, all the while keeping track of the minimum distance that it sees and then returns the minimum Levenshtein distance of its chunk. One core then gathers up the minimum distances of each chunk and finds the overall minimum across them.

We also considered parallelizing the Levenshtein distance calculation itself however realized that it is already very optimal. Using dynamic programming and memoization, it runs in O(nm) where n and m are the length of the strings. The longest word in the English dictionary is 45 letters long so it runs in constant time for all real-world scenarios. So parallelizing this part of the program would not make a significant difference to the end speedup.

4. Sequential Haskell Implementation

This Haskell code is a simple implementation of the MapReduce paradigm for word counting, with fuzzy searching to find the most similar word to a user-inputted search word using the Levenshtein distance. We detail the implementation and its functions below.

4.1 Data Cleaning and Splitting

```
Unset
cleanAndSplit :: String -> [String]
cleanAndSplit s = words $ Prelude.map toLower $ Prelude.filter (\x
-> isAlpha x || isSpace x) s
```

The ‘cleanAndSplit’ function is used to preprocess the input text. It uses the ‘map’ and ‘filter’ functions from ‘Prelude’ to transform and filter the input string. The lambda function ‘(\x -> isAlpha x || isSpace x)’ filters out non-alphabetic characters except for spaces, and then ‘words’ is used to split the string into a list of words.

4.2 Map Stage

```
Unset
doMap :: [String] -> [(String, Int)]
doMap xs = [(w,1) | w <- xs]
```

The ‘doMap’ function performs the map phase of MapReduce. It uses Haskell’s list comprehension to create a list of key-value pairs, such that each word in the input list ‘xs’ is paired with the integer ‘1’.

4.3 Reduce Stage

```
Unset
doReduce :: [(String, Int)] -> Map String Int
doReduce xs = Map.fromListWith (+) xs
```

The ‘doReduce’ function performs the reduce phase of MapReduce. It uses the ‘fromListWith’ function from the ‘Data.Map’ module alongside the ‘(+)’ function to merge the list of tuples ‘xs’ into a map, summing the values of duplicate keys.

4.4 Levenshtein Distance Calculation

```
Unset
calcLevenshteinDist :: String -> String -> (Int, String)
calcLevenshteinDist w1 w2 = (last $ Prelude.foldl transform
[0..length w1] w2, w2)
  where
       transform xs@(x:xs') c = scanl compute (x + 1) (zip3 w1 xs
xs')
       where
       compute z (c', x', y) = minimum [y + 1, z + 1, x' +
fromEnum (c /= c')]
```

The ‘calcLevenshteinDist’ function calculates the Levenshtein distance between two strings ‘w1’ and ‘w2’. It uses dynamic programming to achieve this by using ‘foldl’ and ‘scanl’ to build up a list of minimum edit distances between the two strings. The ‘zip3’

function then combines the characters of the two strings and the intermediate results to calculate the minimum number of edits required.

4.5 Fuzzy Search

```
Unset
getClosestWord :: Map String Int -> String -> (String, Int)
getClosestWord wordFreq target = (closestWord, resultInt $
Map.lookup closestWord wordFreq)
       where
       closestWord = getClosestWord' calcLevenshteinDist (Map.keys
wordFreq) target
```

```
Unset
getClosestWord' :: (String -> String -> (Int, String)) -> [String] ->
String -> String
getClosestWord' distFunction listWords target' = snd $ Map.findMin $
Map.fromList $ Prelude.map (distFunction target') listWords
```

```
Unset
resultInt :: Maybe Int -> Int
resultInt r = case r of
       Just x  -> x
       Nothing -> -1
```

The ‘getClosestWord’ function finds the word in the ‘wordFreq’ map that is most similar to the ‘target’ word. It uses a helper function ‘getClosestWord’’ to iterate over the keys of the ‘wordFreq’ map and calculate their Levenshtein distances to the target word. The ‘resultInt’ helper function extracts frequencies from the ‘Map.lookup’ function.

4.6 User Interaction

```
Unset
getUserInput :: Map String Int -> IO ()
getUserInput wordFreq = do
  putStr "Enter a search word (or 'exit' to quit): "
  hFlush stdout
  target <- getLine

  if target == "exit"
       then putStrLn "Exiting..."
       else do
       putStrLn ("  You entered: \"" ++ target ++ "\"")
       let value = resultInt $ Map.lookup target wordFreq
       if value /= -1
       then putStrLn $ "  count: " ++ show value
       else do
       let (closest, count) = getClosestWord wordFreq target
       putStrLn $ "  search word not found:"
       putStrLn $ "  closest word: \"" ++ closest ++ "\"\n  count:
" ++ (show count)
       getUserInput wordFreq
```

The ‘getUserInput’ function repeatedly prompts the user for a search word and provides feedback. It uses Haskell's IO monad to handle side effects, such as reading from and writing to the console. The ‘putStr’, ‘hFlush’, and ‘getLine’ functions are used for user interaction, while the if expression and do blocks handle the control flow.

4.7 Main Function

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

```
        [filename] -> do
        content <- readFile filename
        putStrLn $ "Starting MapReduce word counting..."
        let wordFreq = doReduce $ doMap $ cleanAndSplit content
        wordFreq `deepseq` return ()     -- force computation
        putStrLn $ "MapReduce completed..."
        getUserInput $ wordFreq
        _ -> do
        pn <- getProgName
        die $ "Usage: " ++ pn ++ " <filename>"
```

The 'main' function is the entry point of the Haskell program. It parses command-line arguments, reads the content of a file, and applies the 'cleanAndSplit', 'doMap', and 'doReduce' functions to count word frequencies. The 'deepseq' function is used to force the evaluation of the wordFreq map before proceeding to user interaction. The program uses pattern matching to handle different cases of command-line arguments and provides usage information if the arguments are incorrect.

5. Parallel Haskell Implementation

To optimize the sequential implementation and take advantage of Haskell's parallelism, we parallelized the two key components of our implementation: MapReduce and Fuzzy Search. We describe our process for parallelizing each of these algorithms below.

5.1 Parallel Map Reduce

The parallelization of the MapReduce process is achieved by splitting the input data into chunks, processing each chunk in parallel, and then combining the results.

```
Unset
doMapReduce :: String -> Map.Map String Int
doMapReduce = Map.fromListWith (+) . flip zip (repeat 1) . cleanAndSplit
```

The ‘doMapReduce’ function is similar to the sequential version, but it operates on a chunk of the input data rather than the whole data. It cleans and splits the input string into words, maps each word to a tuple of the word and 1, and reduces the tuples by summing the counts of the same words.

```
Unset
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs = let (ys, zs) = splitAt n xs in ys : chunk n zs
```

The ‘chunk’ function is used to split the input data into chunks of approximately equal size. The number of chunks is determined by the number of available cores (‘numCapabilities’), which allows the program to fully utilize the processing power of the machine.

```
Unset
main = do
 args <- getArgs
 case args of
       [filename] -> do
       content <- readFile filename
       putStrLn "Starting MapReduce word counting..."
       let chunks = chunk (length content `div` numCapabilities)
content
       mappedReduced = map doMapReduce chunks `using` parList rdeepseq
       wordFreq = Map.unionsWith (+) mappedReduced
       wordFreq `deepseq` return () -- force computation
       putStrLn "MapReduce completed..."
       getUserInput wordFreq
```

```
        _ -> do
        pn <- getProgName
        die $ "Usage: " ++ pn ++ " <filename>"
```

In the ‘main’ function, we parallelize the MapReduce process as follows:

1. The first step in the parallelization process is to divide the input data into chunks. The chunk function is used to split the input string content into non-overlapping substrings. The size of each chunk is determined by dividing the total length of the input string by the number of available cores (numCapabilities). This division ensures that the workload is evenly distributed across all cores, maximizing the utilization of the machine's processing power.
2. The next step is to apply the ‘doMapReduce’ function to each chunk in parallel. This is achieved by using the ‘map’ function in combination with the ‘using’ function and the ‘parList rdeepseq’ strategy. The map function applies the ‘doMapReduce’ function to each chunk, resulting in a list of maps where the keys are words and the values are their frequencies in each chunk. The ‘using’ function is used to apply a parallel strategy to a value. In this case, the value is the list of maps, and the strategy is ‘parList rdeepseq’. The parList strategy applies a strategy to each element of a list in parallel, and the strategy used here is ‘rdeepseq’, which forces the complete evaluation of each map, ensuring that the computation of each chunk is fully evaluated before proceeding.
3. The final step is to combine the results of the parallel computations into a single map. The ‘Map.unionsWith (+)’ function is used to merge a list of maps into one map. If a key appears in more than one map, the function combines the corresponding values using the (+) function, effectively summing the frequencies of each word across all chunks and creating the final word frequency map.

5.2 Parallel Fuzzy Search

The parallelization of the fuzzy word search process is achieved by splitting the frequency map into chunks, calculating the Levenshtein distances between the target word and each word within the chunk in parallel while simultaneously keeping track of the minimum distance seen, and then finding the minimum distance across each chunks' minimums.

```
Unset
getClosestWord :: Map.Map String Int -> String -> (String, Int)
getClosestWord wordFreq target = (closestWord, resultInt $ Map.lookup
closestWord wordFreq)
  where
  closestWord = getClosestWord' (Map.keys wordFreq) target
```

The ‘getClosestWord’ function is similar to the sequential version, but no longer passes the ‘calcLevenshteinDist’ function as a parameter to its helper function.

```
Unset
getClosestWord' :: [String] -> String -> String
getClosestWord' listWords target' = snd $ minimum $ parMap rdeepseq
(findMinDist startMin startWord target') chunks
 where
  chunks = chunk (length listWords `div` numCapabilities) listWords
  startMin = -1
  startWord = ""
```

The ‘getClosestWord’ helper function breaks the map into chunks using the ‘chunk’ function and applies the ‘findMinDist’ function to each chunk. This is achieved by using the ‘parMap rdeepseq’ strategy. Then it collects the results of this strategy which is a list of calculated minimum Levenshtein distance pairs across the chunks, finds the minimum of them sequentially, and returns the word from the overall minimum distance pair.

```
Unset
findMinDist :: Int -> String -> String -> [String] -> (Int, String)
findMinDist minDist minWord _ [] = (minDist, minWord)
findMinDist minDist minWord searchWord (x:xs)
 | minDist == -1 = findMinDist dist word searchWord xs
 | otherwise   = findMinDist (min minDist dist) findMinWord searchWord
xs
  where
   distPair = calcLevenshteinDist searchWord x
   dist = fst distPair
   word = snd distPair
   findMinWord | minDist <= dist = minWord
         | otherwise    = word
```

The ‘findMinDist’ function is what each core applies to their chunk in parallel. It keeps track of the minimum distance seen, and the word that corresponds to that distance, while calculating the Levenshtein distances between the target word and the words in its chunk. The ‘calcLevenshteinDist’ is the same as we did not parallelize this due to insignificant potential for speedup.

6. Evaluation and Performance

To compare the sequential implementation with the parallel implementation, we performed the following for both the sequential and parallel implementations:

1. Compiled with Full Threading and Optimization
    a. stack ghc -- -O2 -Wall -threaded -rtsopts -eventlog .\wfsSequential.hs
    b. stack ghc -- -O2 -Wall -threaded -rtsopts -eventlog .\wfsParallel.hs
2. Implemented File Input Functionality
    a. We implemented file input functionality to avoid issues with user input when testing our code. Thus, we adjusted the programs to take in a second command line argument, specifying an input text file that should contain a list of queries to the program after the Map Reduce stage has been completed. Below is the modified code, which includes a

function for running the tests within the input query file, and the modified main function to take in the additional command line argument for the file.

```
Unset
runTests :: Map.Map String Int -> [String] -> IO ()
runTests wordFreq testWords = do
 putStrLn "Testing..."
 let results = [getValueOrClosest wordFreq t | t <- testWords]
       printTuple (str, num) = putStrLn $ "(" ++ str ++ ", " ++ show num ++
")"
 mapM_ printTuple results
 putStrLn "Done."
```

```
Unset
       main :: IO ()
       main = do
        args <- getArgs
        case args of
              [filename, testfile] -> do
              content <- readFile filename
              tests <- readFile testfile
       ...
       runTests wordFreq (cleanAndSplit tests)
       _ -> do
       pn <- getProgName
       die $ "Usage: " ++ pn ++ " <filename>"
```

3. Tested with Three Input Query Files
    a. testEasy.txt
        i. This file contains short words already in the map. This serves to test the performance of MapReduce, as we are not calling the fuzzy search algorithm at all.
    b. testHardShort.txt

i. This file contains short words that are not in the map. This serves to test the performance of Fuzzy Search without the overhead of Levenshtein distance computation.

c. testHardLong.txt

i. This file contains long words that are not in the map. This serves to test the performance of both the Fuzzy Search algorithm and the Levenshtein distance computation.

6.1 Sequential Testing

For the sequential implementation, we ran it with one core on all three test files. Specifically, we ran it as (100-0.txt is the full works of William Shakespeare, around 1 million words):

```
.\wfsSequentialTest .\100-0.txt test\(queryFile).txt +RTS -N1 -ls -s
```

On each of the test files, these were the evaluation diagrams from Threadscope:

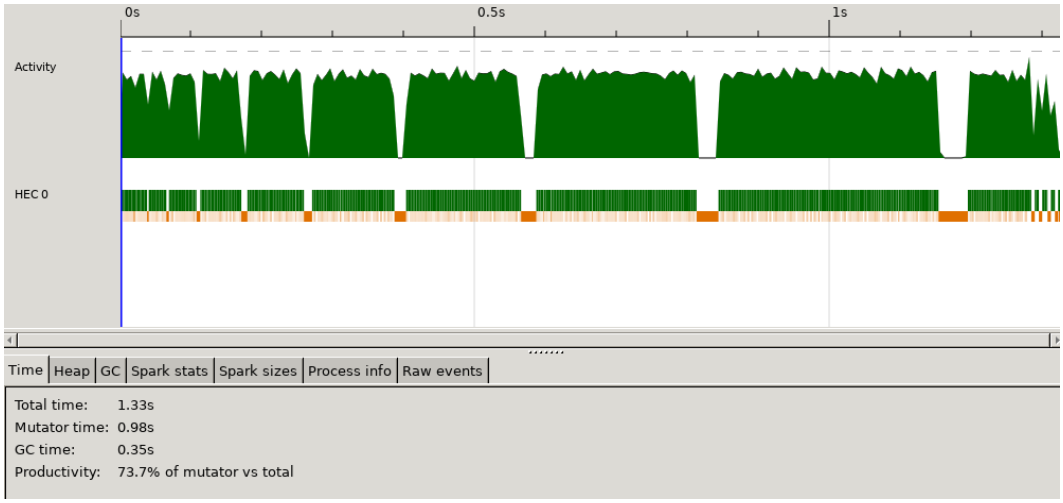

Figure 2. ThreadScope diagram for testEasy.txt running sequentially

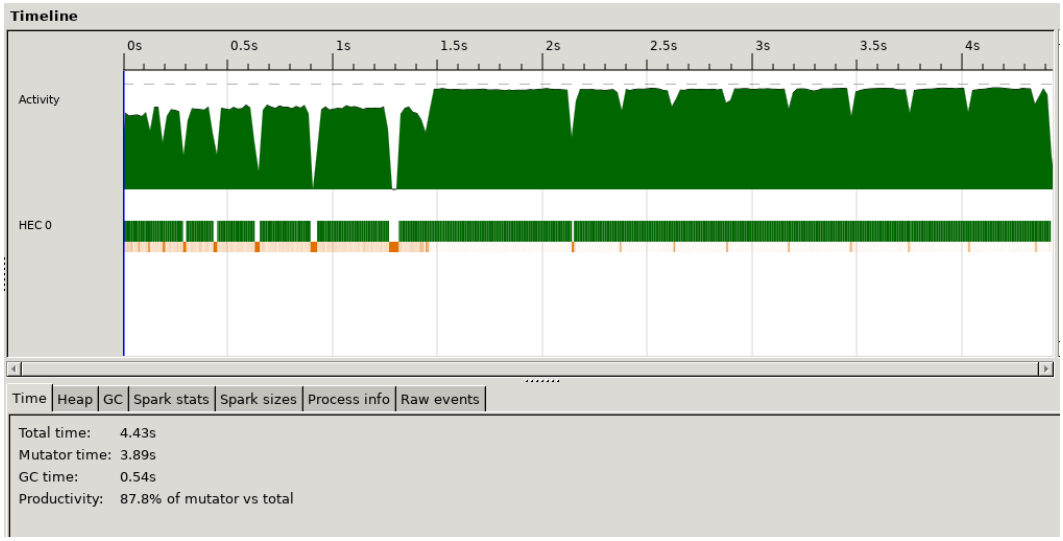

Figure 3. ThreadScope diagram for testHardShort.txt running sequentially

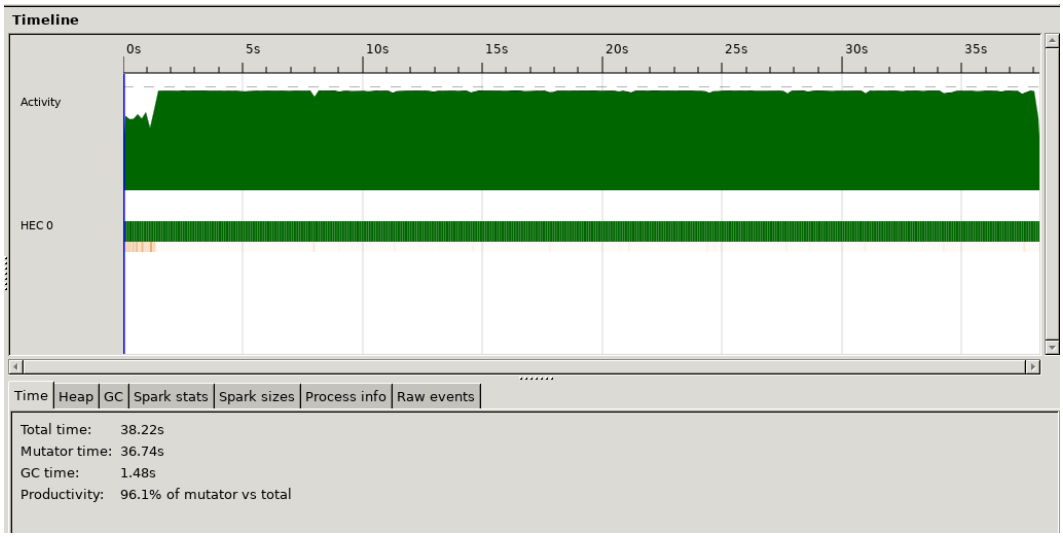

Figure 4. ThreadScope diagram for testHardLong.txt running sequentially

6.2 Parallel Testing

For the parallel implementation, we ran it on C = 2 - 16 cores on all three test files. Specifically, we ran it as (100-0.txt is the full works of William Shakespeare, around 1 million words):

.\wfsParallelTest .\100-0.txt test\(queryFile).txt +RTS -N(C) -ls -s

Here is a graph of the number of cores vs. the speedup over the sequential algorithm:

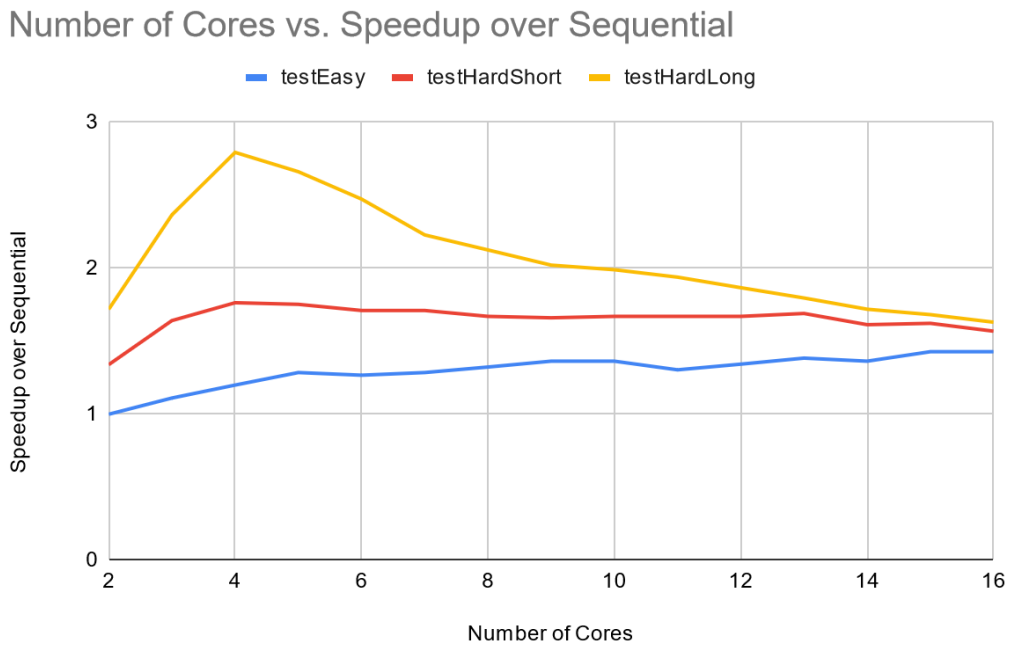

Figure 5. Parallel speedup over sequential as number of cores increases

The parallel MapReduce algorithm benefits linearly with the number of cores, as the computation time continually decreases as the core count increases. This owes largely to the size of the input, which is roughly a million words. This allows for the chunks to be very large for each core to work on, so there is a better distribution of work for each core, and the task is not too fine-grained. For the parallel closest word algorithm, on the other hand, the search space is quite a bit smaller, being roughly 35,000 words, so the amount of work for each core is quite a bit smaller, and it becomes a bit fine-grained. This is why the performance peaks at 4 cores, as it seems that 4 cores is where the best load balancing occurs and each core has a good distribution of work. As the number of cores increases, however, the tasks become too fine-grained and it becomes slower.

For C = 4 cores, on each of the test files, these were the evaluation diagrams from Threadscope:

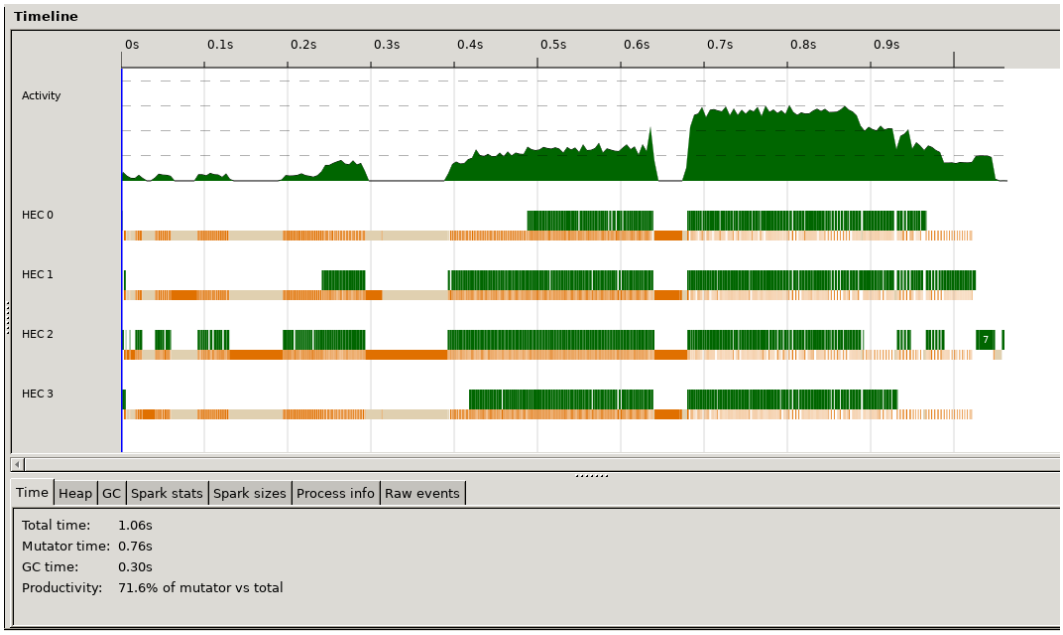

Figure 6. ThreadScope diagram for testEasy.txt running parallel on 4 cores

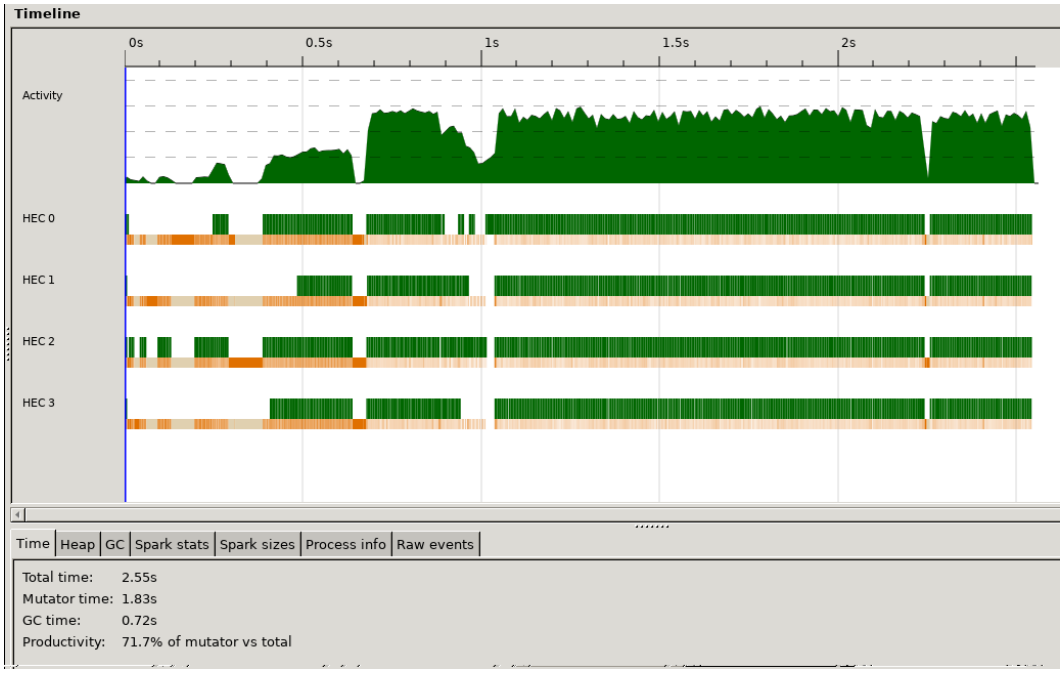

Figure 7. ThreadScope diagram for testHardShort.txt running parallel on 4 cores

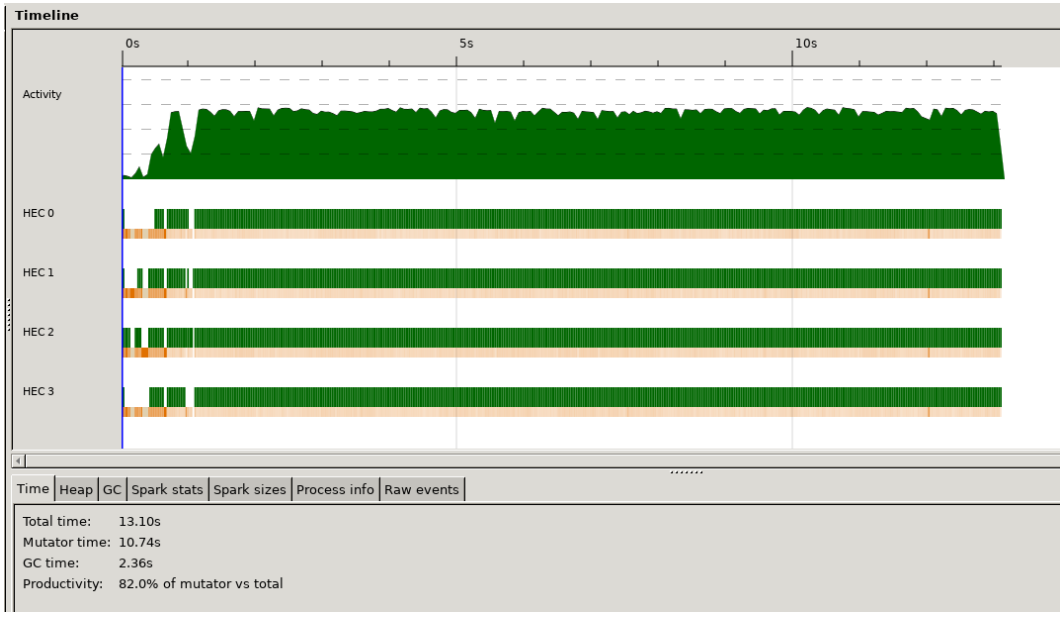

Figure 8. ThreadScope diagram for testHardLong.txt running parallel on 4 cores

7. Reflection and Conclusion

Haskell's high-level abstractions and powerful type system make it an excellent language for parallel programming. The transition from the sequential version of the code to the parallel version was quite easy to reason about, thanks to Haskell's support for parallel strategies and its purely functional nature.

7.1 Effectiveness of the Algorithm, and Possible Bounds

7.1.1 Parallel Map Reduce

In the parallel Map Reduce algorithm, the input data is divided into chunks, and the map and reduce operations are performed on each chunk in parallel. This is achieved using the parList rdeepseq strategy, which applies a function to each element of a list in parallel and ensures that the computation of each element is fully evaluated.

This approach takes full advantage of the available cores and scales well with the size of the input data. As the size of the input data increases, the workload for each core also increases, ensuring that all cores have plenty of work to do. This is particularly beneficial for I/O-bound tasks like MapReduce, as it allows the CPU to continue processing while waiting for I/O operations to complete.

However, it's worth noting that the effectiveness of this approach depends on the size of the input data and the number of available cores. If the input data is too small or the number of cores is too large, the overhead of managing the parallel computations may outweigh the benefits.

7.1.2 Parallel Fuzzy Search

The process of finding the closest word was also parallelized by dividing the list of words into chunks and processing each chunk in parallel. This is similar to the parallelization of the MapReduce process and is achieved using the same parList rdeepseq strategy.

However, the parallelism of this process is somewhat limited by the size of the search space, which is the number of keys in the map. The maximum number of keys is the number of unique words in the input data, which is bounded by the number of words in the English language (around 500,000). As a result, if the number of cores increases to be too large, the workload for each core becomes too fine-grained, and the overhead of managing the parallel computations may outweigh the benefits.

7.2 Future Optimizations and Known Issues

7.2.1 Overall

1. Core Utilization: Currently, the program uses the maximum number of cores available on the machine (numCapabilities). While this approach maximizes the utilization of the machine's processing power, it may not be optimal in all situations. For example, it may make the work for each core too fine-grained for the input data. A potential optimization could be to perform adaptive core allocation in the program based on the input and resource contention in the computer.
2. File Reading: The program currently reads the entire input file into memory before processing it. This approach can be inefficient and potentially problematic when dealing with large files, as it can consume a significant amount of memory. A potential optimization could be to read the file in chunks and process each chunk as it is read, reducing memory usage and potentially improving performance by overlapping I/O and computation.

7.2.2 Fuzzy Search

1. Data Structure: The program currently uses a list to store the words for the fuzzy search algorithm. Searching a list has a linear time complexity, which can be inefficient when dealing with a large number of words. A potential optimization could be to use a more efficient data structure, such as a trie. A trie can significantly prune the search space and speed up the search process, especially when the words share common prefixes.

2. Output Consistency: The output of the fuzzy search algorithm in the parallel version is not necessarily the same as in the sequential version. This is because when there are ties in the minimum Levenshtein distance, the parallel version may return a different word than the sequential version due to the non-deterministic order of parallel computations. While this is not necessarily a problem (as all words with the minimum distance are equally valid), it could be confusing to users. A potential solution could be to sort the words and always return the lexicographically smallest word in case of ties.
3. Minimum Distance Tracking: The program currently keeps track of the minimum distance seen by each core as it calculates the distances. This is done iteratively in theory but enacted recursively due to the recursive nature of the findMinDist function. While this approach works, it may not be the most efficient way to track the minimum distance. A potential optimization could be to use a more efficient data structure or algorithm to track the minimum distance, reducing the computational overhead and potentially improving performance.

8. References

---

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008. doi:10.1145/1327452.1327492

[2] Professor Roxana Geambasu, COMS W4113 Distributed Systems lecture slides. https://systems.cs.columbia.edu/ds1-class/lectures/02-map-reduce.pdf