COMS 4995 Parallel Functional Programming
Burrows-Wheeler transform

David Winograd (dmw2181), Zulal Ozyer (zo2146)

December 22, 2021

1. Introduction

The aim of this project is to compare Haskell's sequential programming performance to its parallel programming performance. We used Burrows-Wheeler transform algorithm to compare the two programming performances.

2. Burrows-Wheeler Transform

Burrows-Wheeler transform takes a character string (either a word or sentence) and alters it to make it easier for compression. There are many applications of the BTW algorithm, such as in the field of biology, when dealing with lettered genome sequences (A, C, T, G). The general idea consists of constructing a list in which the rows are all rearranged systematically, typically in “dictionary order”. Once elements in the list are clustered together, they are ordered in such a way that the string itself becomes more compressible due to similar letters being close to one another.

3. Implementation

a. Sequential Implementation

There are 3 steps to perform Burrows-Wheeler Transform on a string:

1. In the first step, we form all the cyclic rotations of a given string. If we use a sample word such as “tree”, the rotations would consist of “tree$”, “$tree”, “e$tre”, “ee$tr”, and “ree$t”.
2. In the second step, we order the formed rotations lexicographically. In our example, this would be sorted as “$tree”, “e$tre”, “ee$tr”, “ree$t”, “tree$”, so our last column would be “eert$”.
3. The last step is outputting the last column of the strings. We do this because the last column will have the best clustering of symbols when compared to all of the other columns. Also, with this BWT output, the remaining rotations of the original word can be recovered. None of the other columns have this unique characteristic.

We applied step 1, step 2 and step 3 for all the strings in a given text and constructed a list from the output of step 3.

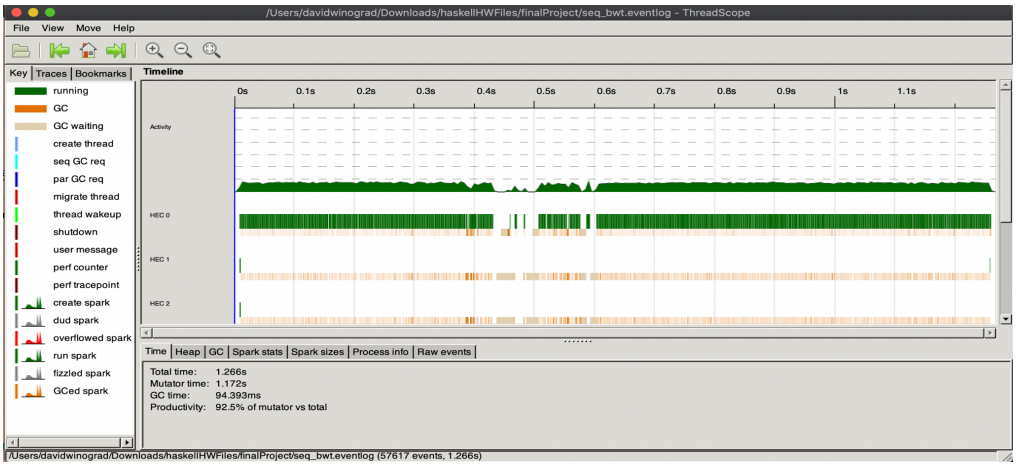
b. Parallel Implementation

We tried two approaches for the parallel implementation. In the first approach, we first divide the given text into two. Then we performed the 3 steps from sequential implementation on each of the strings of two separated chunks of the text. We used two rpar and two rseq functions to apply 3 steps simultaneously. However, the first approach did not work as it was not executed in a fully parallel manner. In the second approach, we used parListChunk to run the functions on separate chunks of words. We tried many numbers to find the optimal chunks for the parListChunk function. We used 6 cores because our computer’s processor has 6 intel cores.

4. Results

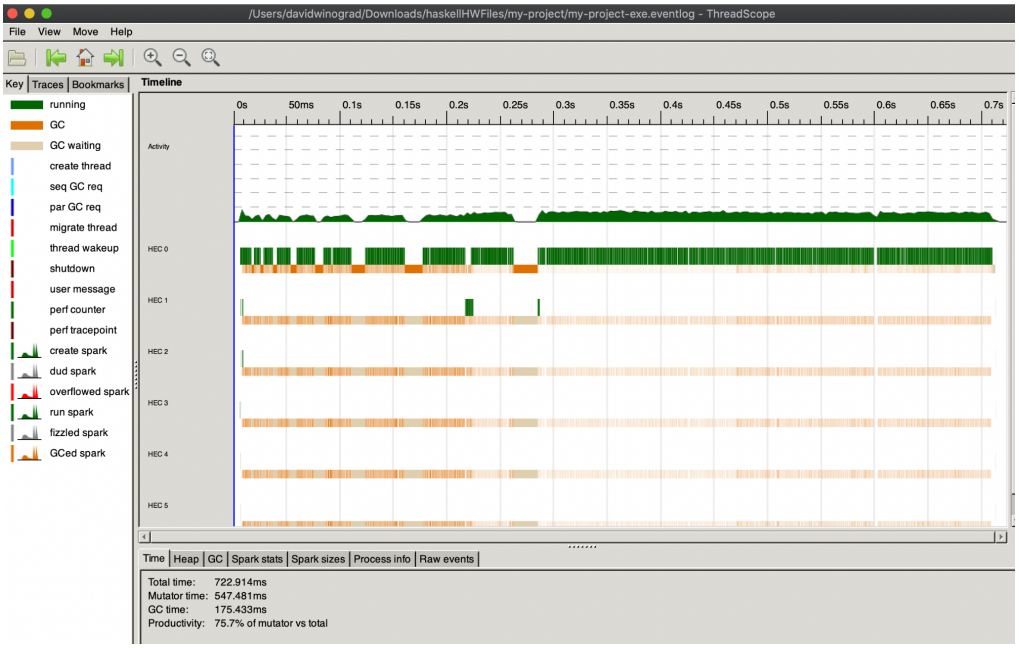
a. Sequential Performance

i. Using Threadscope, we were able to run our program sequentially with a total time of 1.266 seconds. This varies depending on the number of words in our sample text file, but we will be using a file with roughly 100,000 words across the board. Towards the end of the threadscope performance, there is some time accounted for printing out the result, which takes roughly half the time as the computation, given that 100,000 words are printed out.

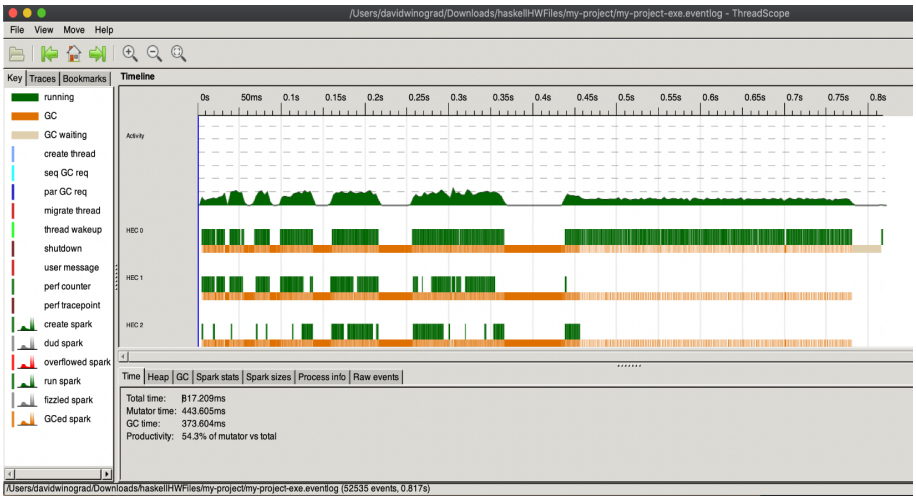
ii. 

b. Parallel Performance

i. For the first approach, although we could get the parallel implementation run faster than the sequential implementation, the execution itself was not fully parallel, and therefore, was not a valid result.

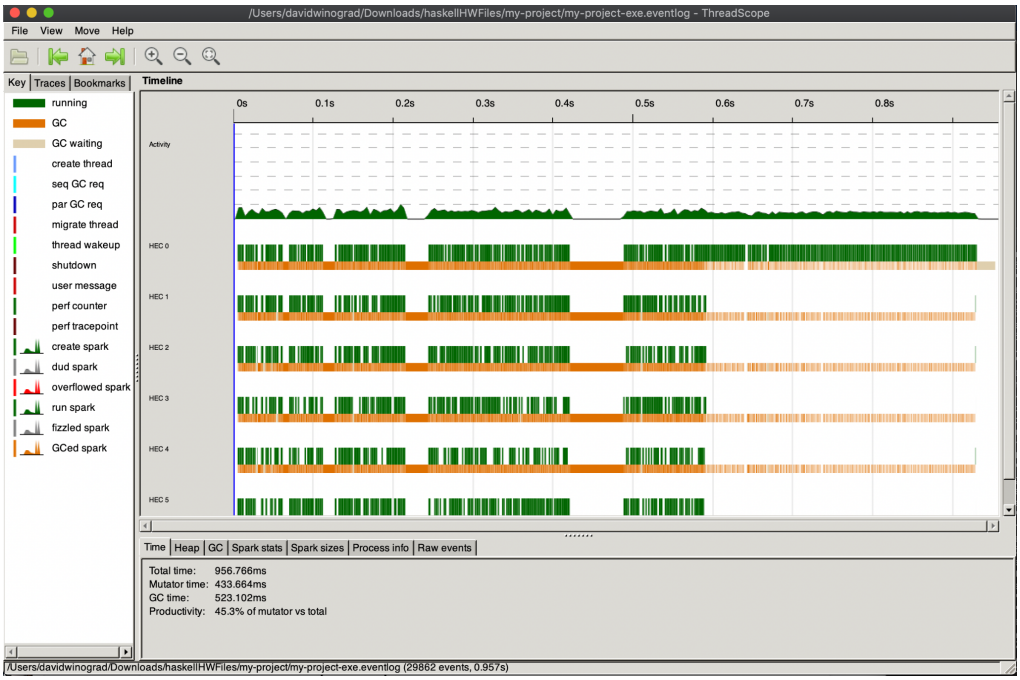
ii. 

iii. For the second approach, we were able to run our program with a total time of approximately 0.817 seconds. There is also some garbage collection time taking place in the initial stages of the performance, which appears to be standard practice and cannot be removed. Below is the figure for a chunk size of 5000.

iv. 

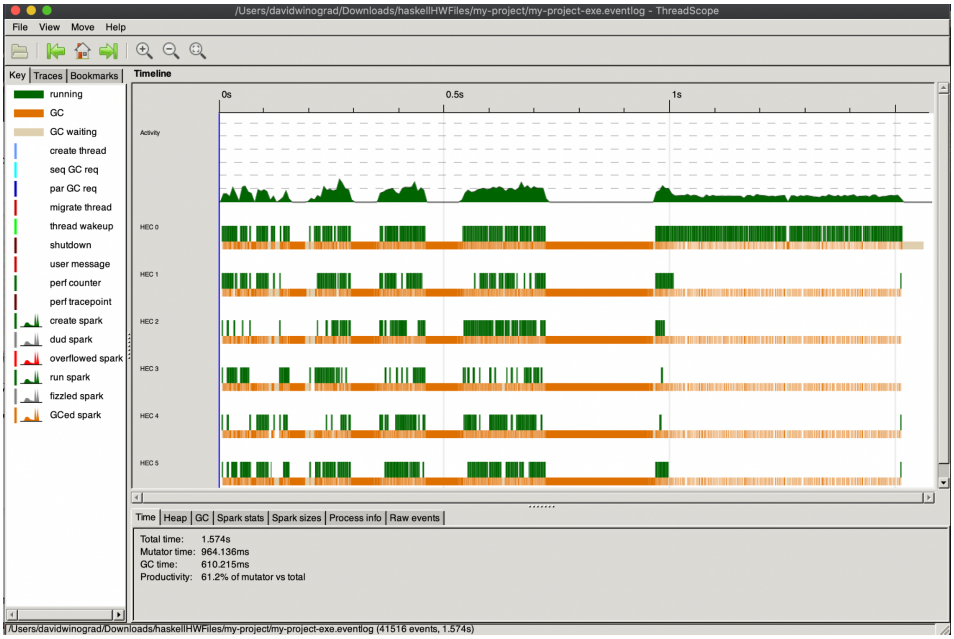
v. With this new approach, we had to try various chunk sizes to see which was the most optimal for performance and efficiency. Below is the parallel

implementation with a chunk size of 100.



vi. Below is the parallel implementation with a chunk size of 3000.

vii.



5. Conclusion

a. All in all, this was a very rewarding and educational project. There were several challenges and lessons learned. Challenges included determining which parallelization function is most suitable (Rseq vs Rpar vs parList vs parListChunk) and customizing the chunk number to find optimal performance. Since our code was not too incredibly complicated, the benefits of parallelization were not always obvious, which is why we needed to include a very long text file of 100,000 words to see the results. In addition, we learned a lot about parallelism of algorithms, string compression in Haskell, and building Haskell projects from scratch, including programs like Stack, Cabal, and Threadscope.

6. References

a. https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/

7. Code

a. Sequential Implementation

```haskell
--sequential implementation
import System.Environment(getArgs, getProgName)
import System.IO()
import System.Exit(die)
import Data.Char(toLower)
import Data.List

main :: IO ()
main = do
 args <- getArgs
 case args of
  [filename] -> do
   contents <- readFile filename
   let lwords = cleaningWords contents
   print (buildFinalList lwords)

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

--clean words for the operation
cleaningWords :: String -> [String]
cleaningWords s = words (map toLower s)

--construct a list for all burrows-wheeler transformation of the words
buildFinalList :: [String] -> [String]
buildFinalList [] = []
buildFinalList (x:xs) = [getFinalColumn (sortOn (map toLower) (createAllRotations (length x) x))] ++ (buildFinalList xs)

--form all the cyclic rotations of a given word
createAllRotations :: Int -> String -> [String]
createAllRotations 0 _ = []
createAllRotations n word = [w] ++ (createAllRotations (n-1) w)
 where w = rotateWord word

--outputting the last column of the string
getFinalColumn :: [String] -> String
getFinalColumn [] = []
getFinalColumn (x:xs) = [last x] ++ (getFinalColumn xs)

--single rotation for a given word
rotateWord :: [Char] -> [Char] --rotate "hello" --> "ohell"
rotateWord word = take len $ drop (negate 1 `mod` len) $ cycle word
 where len = length word

```

b. Parallel Implementation

Users > davidwinograd > Downloads > haskellHWFiles > my-project > app > Main.hs

```haskell
module Main where

import Lib

import System.Environment(getArgs, getProgName)
import System.IO()
import System.Exit(die)
import Data.Char(toLower)
import Data.List
import Data.Maybe
import Control.Parallel.Strategies

main :: IO ()
main = do
 args <- getArgs
 case args of
  [filename] -> do
   contents <- readFile filename
   let lwords = cleaningWords contents
   let solutions = buildFinalList lwords `using` parListChunk 5000 rseq
   print (solutions)

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


cleaningWords :: String -> [String]
cleaningWords s = words (map toLower s)

buildFinalList :: [String] -> [String]
buildFinalList [] = []
buildFinalList (x:xs) = [getFinalColumn (sortOn (map toLower) (createAllRotations (length x) x))] ++ (buildFinalList xs)

createAllRotations :: Int -> String -> [String]
createAllRotations 0 _ = []
createAllRotations n word = [w] ++ (createAllRotations (n-1) w)
 where w = rotateWord word

getFinalColumn :: [String] -> String
getFinalColumn [] = []
getFinalColumn (x:xs) = [last x] ++ (getFinalColumn xs)

rotateWord :: [Char] -> [Char] --rotate "hello" --> "ohell"
rotateWord word = take len $ drop (negate 1 `mod` len) $ cycle word
 where len = length word
```