COMS W4995 Parallel Functional Programming
Project Report: Palindrome Partitioning

Jesse Chong
jlc2332@columbia.edu
December 22 2021 (Fall 2021)

I. Introduction

Palindrome partitioning[1] is an optimization problem where we must split a string into palindromic substrings and then return the minimum number of "cuts" needed to complete the partition. To clarify what "cuts" are, take this for example: the string "abracecar" would require 2 cuts to be minimally partitioned into the substrings "a", "b", and "racecar". For this project, only files with input strings made up of lowercase alphabetical characters are processed.

This problem is a variation of the more popularly known matrix chain multiplication problem[2]. Both of these problems involve finding the most efficient solution. Thus, we can apply the learnings we find from this experiment to other optimization problems as well since this is a generic problem.

In regards to the palindrome partitioning algorithm, there were two sequential solutions that were referenced[3] for this project: a naive brute force algorithm with a running time complexity of $O(n^3)$ and a dynamic programming (DP) algorithm with a running time complexity of $O(n^2)$.

With this in mind comes the problem statement: is it possible to parallelize the naive $O(n^3)$ solution to match or exceed $O(n^2)$? This report seeks to explore this question by explaining how the two algorithms work in Haskell, what experimentation was done to find a parallelized naive algorithm, what results came from the data, and the conclusion.

[1] Taken from this Leetcode problem: https://leetcode.com/problems/palindrome-partitioning-ii/
[2] https://en.wikipedia.org/wiki/Matrix_chain_multiplication
[3] Referenced GeeksForGeeks Python and C++ solutions: https://www.geeksforgeeks.org/palindrome-partitioning-dp-17

II. Sequential Algorithms

A. Naive

The sequential naive palindrome partitioning algorithm runs at $O(n^3)$ and will recurse over the given string to search for valid partitions which in this case would be any longest possible palindromic substring. For example, in the "abracecar" example, partitioning on the substring "racecar" is the best choice while partitioning every character by itself in "racecar" (e.g., "r", "a", "c".. etc.) would be the worst possible partition. In Haskell, we implemented this using the *map* function in a sliding window pattern. The recursive loop inside the *map* will reference a left, middle, and right index in order to process every possible substring.

```
palParSequentialNaive :: String -> Int -> Int -> Int
palParSequentialNaive word l r
  | l >= r                      = 0
  | (isPalindrome word l r) == True = 0
  | otherwise                   = minimum (map (\m -> palParInnerLoop' word l m r) [l..(r - 1)])
  where
    -- Helper function to help with the recursion
    palParInnerLoop' :: String -> Int -> Int -> Int -> Int
    palParInnerLoop' word' l' m' r' = 1 + (palParSequentialNaive word' l' m') + (palParSequentialNaive word' (m' + 1) r')
```

Figure 1. Code snippet of the sequential naive algorithm.

Besides that, there is also the palindrome check itself. This algorithm simply validates a string by comparing characters from the outer bounds of the string (i.e., starting from the *head* and *last* of the string) to the middle.

```
-- Helper function to check if a given String is a palindrome.
-- Starts from the ends of the String until it reaches the middle.
isPalindrome :: String -> Int -> Int -> Bool
isPalindrome str l r
  | l >= r                  = True
  | (str !! l) /= (str !! r) = False
  | otherwise               = isPalindrome str (l + 1) (r - 1)
```

Figure 2. Code snippet of the palindrome check.

In regards to parallelization the *map* itself and the function[4] executed inside the *map* are good candidates. In the section regarding the experiments done, the different methods in which these two sub-algorithms were parallelized will be explored.

B. Dynamic Programming (DP)

The sequential DP algorithm runs at $O(n^2)$. Instead of recursing for the substrings and calculating the minimum number of cuts at the same time, this algorithm instead processes them separately.

Generally speaking, the algorithm starts off by initializing a 1-dimensional array to store the minimum number of cuts needed for each length of the string (from 0 to its length) and a 2-dimensional array to store whether or not a given sequence of the string is a palindrome or not (all possible substrings).

The former is implemented in Haskell using a *list* of *Int*'s where indexing the list by *(length string) -1* will return the minimum number of cuts needed for the entire string itself which is exactly the result the algorithm works towards.

For the latter, it was implemented by using the *Matrix* type found in the *Data.Matrix* library. Since we know substrings of length 1 are palindromes, this 2-dimensional array can be initialized as an identity matrix. The matrix's rows represent the left index of a substring and the matrix's columns represent the right index of a substring.

```
-- palParSequentialDP, the O(n^2) solution.
palParSequentialDP :: String -> Int -> Int
palParSequentialDP word len = do
  {- c[i]    = A list holding minimum number of cuts needed for each substring for the input string from 0 to i
     p[i][j] = A matrix storing whether or not the substring from i to j is a palindrome.
               If 1, then substring is a palindrome. Else, then not. -}
  let c  = replicate (len) 0 -- Initiate list of cuts
      p  = identity (len)    -- Initiate 2x2 identity matrix of substring lengths
      p0 = foldl (palSubStringLoop' word len) p [2..(len)]
      c0 = foldl (palCutLoop' p0) c [0..(len - 1)]
  c0 !! (len - 1) -- Returns the minimum cut value for the string
```

Figure 3. Code snippet of the sequential DP algorithm.

While processing both of these data structures, the algorithm relies on the previous state to calculate the next. Due to this tight coupling, the sequential DP algorithm does not lend itself well to parallelization. Although the logic itself is more complicated to program, the DP algorithm has the benefit of being extremely fast. A later section will compare the performance of the parallelized naive algorithm to this sequential DP algorithm.

III. Experimenting with Parallelization

To preface, the following experiments were completed within a Linux virtual machine. The virtual machine is running *Ubuntu 20.04* and was allocated ~5 GB of RAM, 4 CPU cores, and 20 GB of storage. The host machine is running *Windows 10* with 8 GB of RAM, 8 CPU cores[5], and 512 GB of SSD storage.

A. Program Summary

The written Haskell program works by reading 3 arguments from the command line: a path to an *input file* which should be one or more lines of strings made from lowercase alphabetical characters, a *mode* which can either be *s, p1,* or *p2*[6]*,* and a *version* which determines what type of algorithm will be used during the processing[7]. With these three inputs the program will calculate the palindrome partition for each inputted string and print out the minimum number of cuts required for each string in order to the console.

Additionally, there is input validation to check for the correct program usage (e.g., that the user does not provide any more or less inputs) and to check if the given string is valid for processing. Besides that, there is also a test file built with *HUnit* that runs all of the algorithms against various test cases.

B. Methods Used

Overall there were 14 different parallelization attempts made on the sequential naive algorithm. The first attempt is referred to as V1, the second as V2, and so forth until V14.

V1 parallelizes the *map* sub-algorithm by using *rpar* to generate a spark per each element of the list.

[4] Referring to the function *palParInnerLoop'*.

[5] Specifically a Intel Core i5-10300H quad core processor with hyperthreading enabled which simulates eight cores.

[6] "s" is sequential, "p1" is the parallelized naive algorithm, "p2", is the same as "p1" but also parallelizes the line-by-line file read operation.

[7] See the source files for more information on the version.

V2 is identical to V1 except it adds another layer of parallelization to the function that finds the minimum cuts needed.

V3 is identical to V1 except instead of generating sparks per element it does it per chunk.

V4 is identical to V2 except instead of generating sparks per element it does it per chunk.

V5 is identical to V1 except it only parallelizes up to a certain depth.

V6 is identical to V2 except it only parallelizes up to a certain depth.

V7 is identical to V1 except instead of *rpar* it uses *rdeepseq* which will evaluate a spark's result completely after generation.

V8 is identical to V2 except instead of *rpar* it uses *rdeepseq* which will evaluate a spark's result completely after generation.

V9 is identical to V7 except it will also use buffering.

V10 is identical to V8 except it will also use buffering.

V11 is identical to V7 except it only parallelizes up to a certain depth.

V12 is identical to V8 except it only parallelizes up to a certain depth.

V13 is identical to V11 except it will also use buffering.

V14 is identical to V12 except it will also use buffering.

These 14 versions were benchmarked against a non-palindromic string of 20 characters. Due to the $O(n^3)$ nature of the sequential naive algorithm, anything more than that would take too long[8] for the purposes of this experiment. To further clarify, the benchmark was performed against a non-palindromic string to force the algorithm to take the longest amount of time possible (i.e., the worst case). The following table shows the results of the experiment.

Algorithm	Avg. Runtime (s)	Algorithm	Avg. Runtime (s)
Naive	21.195	V8	23.8696
V1	21.2012	V9	21.7052
V2	46.8322	V10	25.351
V3	22.9644	**V11**	**11.9704**
V4	46.9568	V12	22.4668
V5	**9.193**	**V13**	**12.7192**
V6	19.7866	V14	20.2772
V7	18.9756		

Table 1. Summary table showing performance of each of the parallel algorithms against the sequential naive. V5, V11, and V13 were the top performers.

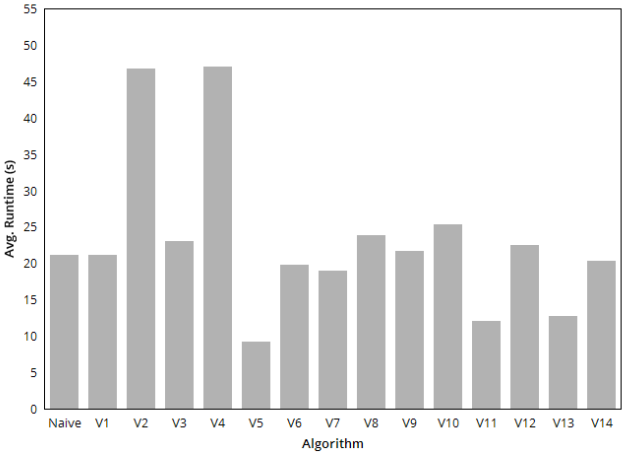

Figure 4. Bar chart of algorithm average running times.

The full table[9] tracks various metrics such as memory used in bytes, total time elapsed in seconds, and information regarding the sparks. These were recorded by hand after running the program against the *alphabet20.txt*[10] file 5 times with the options *+RTS -N4*[11] *-s*. From these results, the top 3 performing algorithms were V5, V11, and V13. Common features among these 3 algorithms include using only one layer of parallelization on the *map* sub-algorithm and parallelizing only to a certain depth. To note, these 3 algorithms also have

[8] The sequential naive algorithm ran for over 10+ minutes against a non-palindromic string of 24 characters and was not able to complete.

[9] To see all of the data for this table and others in this report, see the attached spreadsheets at the end of the file.

[10] *alphabet20.txt* is a single-line file with 20 lowercase characters of the English alphabet from *a* to *t*.

[11] The sequential algorithms were run with only 1 core.

the lowest amount of sparks garbage collected, sparks fizzled, and total memory used compared to the other versions. This has to do with the *depth* controlling how far the algorithms would parallelize as it seems beyond a certain *depth* value (such as in V1 which has no restriction) a lot of time is wasted on garbage collecting sparks.

Algorithm	Avg. Sparks GC'd	Algorithm	Avg. Sparks GC'd
V1	1157416937	V8	1158977483
V2	2052235732	V9	578672116.4
V3	1155358194	V10	1159493536
V4	1917620651	**V11**	**18219.8**
V5	**43834.8**	V12	572426863
V6	573152935.6	**V13**	**20069**
V7	578487687.6	V14	576922111.6

Table 2. Summary table showing average sparks garbage collected for each of the parallel algorithms.

For the purposes of this project, only V5 was selected to compare with the sequential algorithms since it performed the fastest.

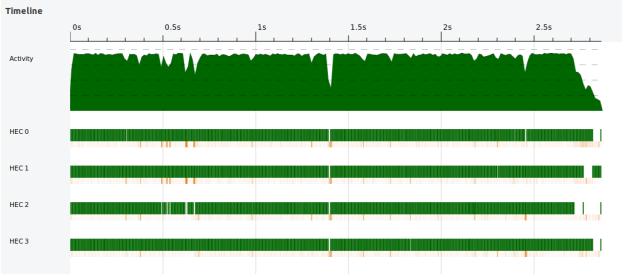

Figure 5. Threadscope of the V5 algorithm running against *alphabet19.txt* since Threadscope kept crashing with the *alphabet20.txt* event log.

Continuing the experiments with V5, the next step is to decide how many cores should be used while running the program. If the amount of cores is not specified, the program will use the option *-N4* hence why the previous benchmark ran against 4 cores. As shown in the following table, *-N4* ended up being the most efficient amount of cores to use.

Cores Run	Avg. Runtime (s)	Algorithm	Avg. Runtime (s)
1	23.65433333	8	14.781
2	12.74833333	12	20.36433333
4	**8.931**	16	26.246
6	14.74166667		

Table 3. Summary table showing the performance of V5 against different numbers of cores.

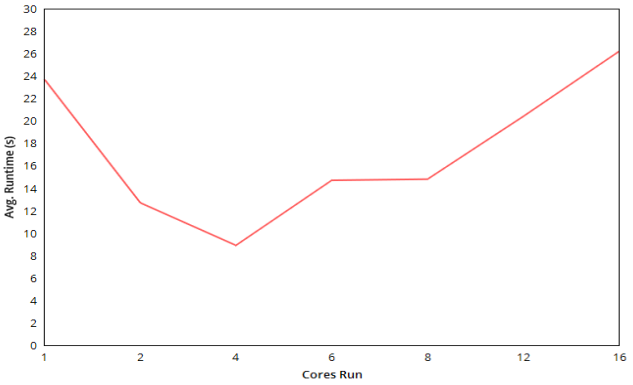

Figure 6. Line chart showing how the number of cores influences the performance of V5.

Going below 4 cores means parallelization is not being leveraged enough while going beyond 4 cores means the program spends too much time cleaning up unused sparks.

Based on the data, the V5 algorithm running on 4 cores is a winner. Now this algorithm will be compared against the sequential naive and sequential DP algorithms.

IV. Parallelization Results

For the results below, each algorithm ran against various test cases for both single-line and multi-line files.

For single-line files: *small1.txt, medium1.txt,* and *medium2.txt* are random[12] lowercase strings while *alphabet1.txt* to *alphabet21.txt* are the lowercase English alphabet with lengths 1 to 21.

For multi-line files: *multiline1.txt* and *multiline2.txt* hold 210 and 18 lines respectively of random valid strings. *slowMultiline1.txt, slowMultiline2.txt,* and *slowMultiline3.txt* hold 11,

[12] *small1.txt* stores "ababb", *medium1.txt* stores "ababbbabbababa", and *medium2.txt* stores "columbia".

6, and 3 lines of 20 lowercase characters of the English alphabet respectively.

Another algorithm called V5' was added for testing purposes. It's identical to V5 except the file reading operation was parallelized. So instead of sequentially reading a file line-by-line, V5' will process this work in parallel. This V5' function was only tested for multi-line cases.

A. Versus Sequential Naive

V5 performs well against the sequential naive algorithm. For single-line files, there was an average speedup of about 2.596 times for V5 compared to the sequential. However, memory usage versus the sequential downscaled by 0.388 times. For multi-line files, there was an average speedup of about 1.850 times. However, memory usage downscaled by about 0.269 times.

V5' performs similarly but suffers as the file has more lines. For multi-line files, there was an average speedup of about 1.683 times. Again, memory usage downscaled by about 0.260 times.

Algorithm	Avg. Runtime (s)	Mem. Usage (bytes)	Runtime Multiplier	Memory Multiplier
Naive (SL)	7.0582857	75783.42857	1	1
V5 (SL)	2.7188571	195401.7143	2.596	0.388
Naive (ML)	87.8194	95758.4	1	1
V5 (ML)	47.4708	355832	1.850	0.269
V5' (ML)	52.1706	368616	1.683	0.260

Table 4. Summary table showing the performance of V5 compared to the naive algorithm.

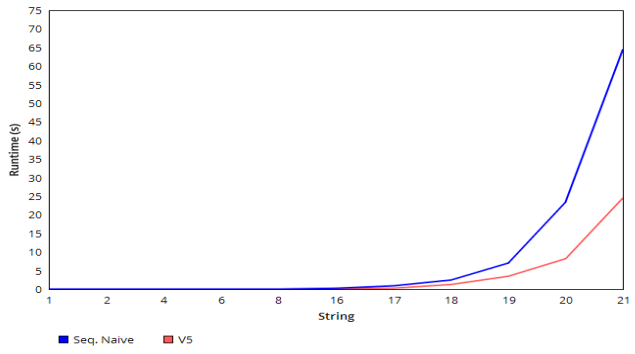

Figure 7. Line chart showing the performance of V5 against the sequential naive algorithm on single-line alphabet strings.

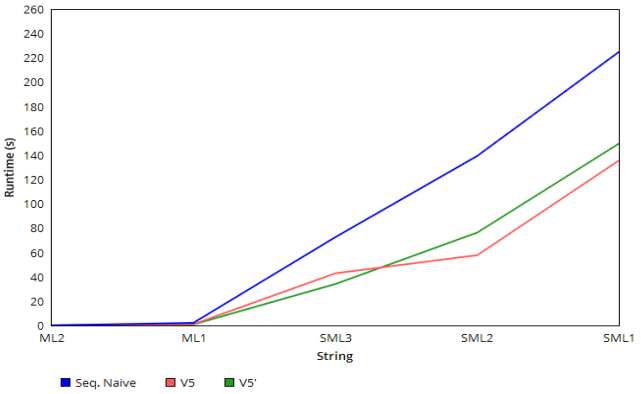

Figure 8. Line chart showing the performance of V5 and V5' against the sequential naive algorithm on multi-line strings.

Overall, the improvements are not bad in comparison to the sequential naive algorithm.

B. Versus Sequential DP

The performance of V5 against the sequential DP algorithm is unfortunately a different story. The V5 comes nowhere close to performing as well as the DP one. For single-line files, the V5 algorithm compared to the sequential DP was 0.004 times slower. Similarly, the V5 algorithm's memory usage was worse by 0.341 times. For multi-line files, V5 was 2.317×10^4 times slower. The V5 algorithm's memory usage downscaled by 0.582 times.

V5' falls short as well, having a running time that was 2.108×10^4 times slower. Memory usage downscaled by 0.562 times.

Algorithm	Avg. Runtime (s)	Mem. Usage (bytes)	Runtime Multiplier	Memory Multiplier
DP (SL)	0.011428571433	66574.28571	1	1
V5 (SL)	2.7188571	195401.7143	0.004	0.341
DP (ML)	0.011	207009.6	1	1
V5 (ML)	47.4708	355832	2.317×10^4	0.582
V5' (ML)	52.1706	368616	2.108×10^4	0.562

Table 5. Summary table showing the performance of V5 and V5' compared to the DP algorithm.

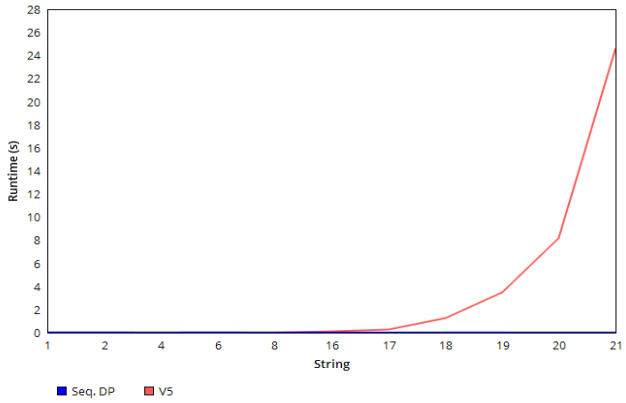

Figure 9. Line chart showing the performance of V5 against the sequential DP algorithm on single-line strings.

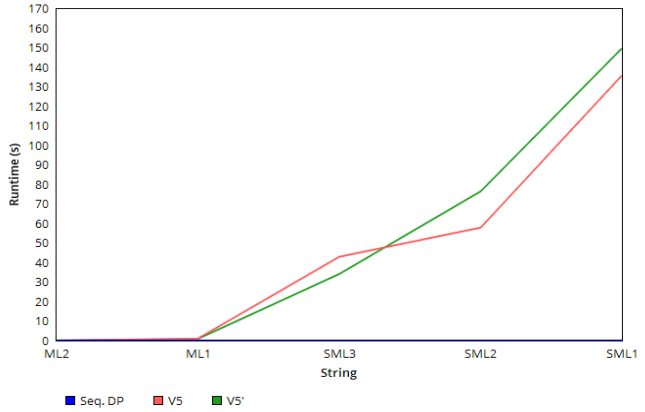

Figure 10. Line chart showing the performance of V5 and V5' against the sequential DP algorithm on multi-line strings.

From the data, it looks to be that the DP algorithm is performing at constant time. However, by increasing the input string size it is clear that DP is not. For large strings, DP still performs quickly at the cost of consuming excess amounts of memory.

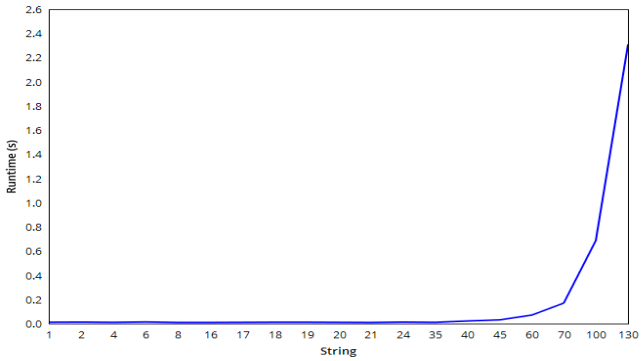

Figure 11. Line chart showing the running time of the sequential DP algorithm for larger strings.

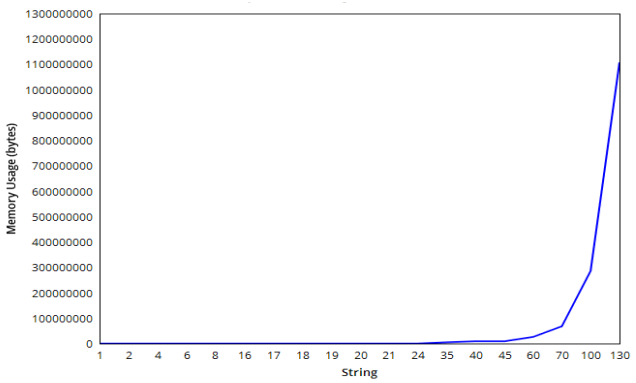

Figure 12. Line chart showing the memory usage of the sequential DP algorithm. Inputting a non-palindromic string of around length 130 will lead to around 1 GB of memory usage.

V. Conclusion

A. Result Summary

Revisiting the problem statement from the beginning of this report, it's clear that parallelizing the sequential naive algorithm is still not enough to match or exceed the $O(n^2)$ DP algorithm. Despite this, parallelization still showed solid performance gains against the slower $O(n^3)$ naive algorithm. On average, we saw a ~2.596 speed up comparing the parallel algorithm to the sequential naive one for single-lined files.

In general, the Palindrome Partitioning problem is not very scalable. For the sequential naive–and in turn the parallel algorithm as well–going beyond 20 characters for a non-palindromic string causes the program to run for a very long period of time (up to 10+ minutes). For the sequential DP algorithm, we see this effect on memory usage with a non-palindromic string of 130 characters taking over 1 GB of memory. The program will begin crashing past that length.

B. Final Thoughts

In terms of being the pinnacle of software speed ups, parallelization is not a "silver bullet"[13]. Although the parallelized solution was outclassed by the DP solution, it's not to say that there wasn't a cost. In terms of code maintenance, the DP algorithm is harder to follow and requires a lot of

[13] https://en.wikipedia.org/wiki/No_Silver_Bullet

developer overhead to be composed at all. On the other hand, parallelization can easily speed up a program with little development overhead and with the benefit of being easier to read and maintain. In conclusion, the findings of this report shows that parallelization is a beneficial method to leverage.

C. Acknowledgements

Special thanks to Professor Stephen Edwards for introducing me to the very interesting (and sometimes scary) world of Parallel Functional Programming and to Max Helman for being an awesome TA!

```
{-

 Name: Jesse Chong
 Uni: jlc2332
 Final Project: Palindrome Partitioning (PalPar)
 File Name: PalParMain.hs

-}

module Main where

import PalParSequential(palParSequential)
import PalParParallel(palParParallel)
import Control.Parallel.Strategies(NFData, parList, rdeepseq, withStrategy)
import Data.Char(isLower)
import System.Environment(getArgs, getProgName)
import System.Exit(die)

{- This program must be given:
1. a path to a file of all lower case Strings
2. a mode, "s", "p1", "p2"
3. and an algorithm version (see src .hs files for what versions there are to use) -}
main :: IO ()
main = do args <- getArgs
          case args of
            [filename, mode, version] -> do
              content <- readFile filename
              let ls = lines content
              palParWrapper' ls mode version
            _ -> do
              pn <- getProgName
              die $ "Usage: " ++ pn ++ " <filename> <mode> <version>"
  where
    {- palParWrapper' is the function that handles input validation
    before it passes the given args over to the actual processing functions
    such as palParSequential or palParParallel. -}
    palParWrapper' :: [String] -> String -> String -> IO ()
    palParWrapper' ls mode version
      | any (== True) [ isInvalid' word | word <- ls] = do
        die $ "Input words must consist of all lowercase alphabetical characters"
      | (isValidVersion' mode version) == False = do
        die $ "Version must be within the valid range (check the Sequential and Parallel .hs files)"
      | mode == "s" = do
        mapM_ (\word -> putStrLn (show $ palParSequential word version)) ls
      | mode == "p1" = do
        mapM_ (\word -> putStrLn (show $ palParParallel word version)) ls
      | mode == "p2" = do
        let p2result = parMapDeepSeq' (\word -> show $ palParParallel word version) ls
        mapM_ (\n -> putStrLn n) p2result
      | otherwise = do
        die $ "Mode must either be sequential 's' or parallel 'p'"

    -- Helper function for checking if a word is all lower case
    isInvalid' :: String -> Bool
    isInvalid' word = any (\w -> not (isLower w)) word
```

```
    -- Helper function for checking if the version is valid
    isValidVersion' :: String -> String -> Bool
    isValidVersion' mode version
      | mode == "s" = elem (read version :: Int) [1..2 ]
      | otherwise   = elem (read version :: Int) [1..14]

    -- Helper function to make the palParParallel calls also parallel
    parMapDeepSeq' :: (NFData y) => (x -> y) -> [x] -> [y]
    parMapDeepSeq' f' = withStrategy (parList rdeepseq) . map f'
```

```
{-

 Name: Jesse Chong
 Uni: jlc2332
 Final Project: Palindrome Partitioning (PalPar)
 File Name: PalParCommon.hs

-}

module PalParCommon(isPalindrome) where

-- Helper function to check if a given String is a palindrome.
-- Starts from the ends of the String until it reaches the middle.
isPalindrome :: String -> Int -> Int -> Bool
isPalindrome str l r
  | l >= r                = True
  | (str !! l) /= (str !! r) = False
  | otherwise             = isPalindrome str (l + 1) (r - 1)
```

```
{-

 Name: Jesse Chong
 Uni: jlc2332
 Final Project: Palindrome Partitioning (PalPar)
 File Name: PalParSequential.hs

-}

module PalParSequential(palParSequential) where

import PalParCommon(isPalindrome)
import Data.Matrix(Matrix, getElem, identity, setElem)

{- Set "version" to a different integer to choose what sequential algorithm gets used.
Version 1 is an O(n^3) naive solution.
Version 2 is an O(n^2) dynamic programming solution.

The below Haskell solutions were written after referencing the Python and C++ implementations on GeeksForGeeks
```

```
:
  - https://www.geeksforgeeks.org/palindrome-partitioning-dp-17/
-}
palParSequential :: String -> String -> Int
palParSequential word version
  | version == "1"  = palParSequentialNaive word 0 ((length word) - 1)
  | version == "2"  = palParSequentialDP word (length word)
  | otherwise       = palParSequentialNaive word 0 ((length word) - 1)

{- palParSequentialNaive, the O(n^3) solution.
Past about 16 characters, this algorithm experiences immense slow down. -}
palParSequentialNaive :: String -> Int -> Int -> Int
palParSequentialNaive word l r
  | l >= r                      = 0
  | (isPalindrome word l r) == True = 0
  | otherwise                   = minimum (map (\m -> palParInnerLoop' word l m r) [l..(r - 1)])
  where
    -- Helper function to help with the recursion
    palParInnerLoop' :: String -> Int -> Int -> Int -> Int
    palParInnerLoop' word' l' m' r' = 1 + (palParSequentialNaive word' l' m') + (palParSequentialNaive word' (m' + 1)
r')

-- palParSequentialDP, the O(n^2) solution.
palParSequentialDP :: String -> Int -> Int
palParSequentialDP word len = do
  {- c[i]    = A list holding minimum number of cuts needed for each substring for the input string from 0 to i
     p[i][j] = A matrix storing whether or not the substring from i to j is a palindrome.
               If 1, then substring is a palindrome. Else, then not. -}
  let c  = replicate (len) 0 -- Initiate list of cuts
      p  = identity (len)    -- Initiate 2x2 identity matrix of substring lengths
      p0 = foldl (palSubStringLoop' word len) p [2..(len)]
      c0 = foldl (palCutLoop' p0) c [0..(len - 1)]
  c0 !! (len - 1) -- Returns the minimum cut value for the string

  where
    {- Builds the matrix of substring palindrome conditions.
       p'   = palindrome matrix
       len' = length of the input string
       l'   = length of the substring     -}
    palSubStringLoop' :: String -> Int -> Matrix Int -> Int -> Matrix Int
    palSubStringLoop' word' len' p' l' = foldl (palSubStringInner' word' l') p' [0..(len' - l')]

    -- Checks whether the substring is a palindrome or not.
    palSubStringInner' :: String -> Int -> Matrix Int -> Int -> Matrix Int
    palSubStringInner' word' l' p' i' = do
      let j' = i' + l' - 1 -- Ending index for the substring
      if l' == 2 then do -- Length of the substring was 2, so just compare two characters
        setElem' (fromEnum $ ((word' !! i') == (word' !! j'))) (i', j') p'
      else do
        setElem' (fromEnum $ ((word' !! i') == (word' !! j') && ((getElem' (i' + 1) (j' - 1) p') == 1))) (i', j') p'

    {- Builds the list of minimum cuts needed for the substrings.
       p0' = palindrome matrix
       c'  = palindrome cut list
       l'  = length of the substring -}
```

```
  palCutLoop' :: Matrix Int -> [Int] -> Int -> [Int]
  palCutLoop' p0' c' l'
    | getElem' 0 l' p0' == 1 = update' l' 0 c' -- The substring was a palindrome. No cuts needed; set to 0
    | otherwise              = foldl (palCutInner' p0' l') (update' l' (maxBound :: Int) c') [0..(l' - 1)]

  -- Counts the number of cuts for each substring
  palCutInner' :: Matrix Int -> Int -> [Int] -> Int -> [Int]
  palCutInner' p0' l' c' j' = do
    if ((getElem' (j' + 1) l' p0') == 1) && ((1 + (c' !! j')) < (c' !! l')) then do
      update' l' (1 + (c' !! j')) c'
    else do
      c'

  {- Modification on the Matrix getElem function to use zero based indexing.
     https://hackage.haskell.org/package/matrix-0.3.6.1/docs/Data-Matrix.html#v:getElem -}
  getElem' :: Int -> Int -> Matrix a -> a
  getElem' x y m = getElem (x + 1) (y + 1) m

  {- Modification on the Matrix setElem function to use zero based indexing.
     https://hackage.haskell.org/package/matrix-0.3.6.1/docs/Data-Matrix.html#v:setElem -}
  setElem' :: a -> (Int, Int) -> Matrix a -> Matrix a
  setElem' a (x, y) m = setElem a (x + 1, y + 1) m

  -- Helper function that updates the given list with the given element at the given index
  update' :: Int -> a -> [a] -> [a]
  update' i v ls = (take i ls) ++ [v] ++ (tail $ drop i ls)



-----------------------------------------

{-

 Name: Jesse Chong
 Uni: jlc2332
 Final Project: Palindrome Partitioning (PalPar)
 File Name: PalParParallel.hs

-}

module PalParParallel(palParParallel) where

import PalParCommon(isPalindrome)
import Control.Parallel(par, pseq)
import Control.Parallel.Strategies(NFData, Strategy, parBuffer, parList, parListChunk,
                                   parMap, rdeepseq, rpar, rseq, withStrategy)

-- Set version to a different integer to change what algorithm gets used.
palParParallel :: String -> String -> Int
palParParallel word version
  | version == "1"  = palParParallel1  word 0 ((length word) - 1)
  | version == "2"  = palParParallel2  word 0 ((length word) - 1)
  | version == "3"  = palParParallel3  word 0 ((length word) - 1)
  | version == "4"  = palParParallel4  word 0 ((length word) - 1)
  | version == "5"  = palParParallel5  word 0 ((length word) - 1) 4 -- Change this last number to change depth. Defaul
```

```
t = 4
  | version == "6"  = palParParallel6  word 0 ((length word) - 1) 4 -- Change this last number to change depth. Defaul
t = 4
  | version == "7"  = palParParallel7  word 0 ((length word) - 1)
  | version == "8"  = palParParallel8  word 0 ((length word) - 1)
  | version == "9"  = palParParallel9  word 0 ((length word) - 1)
  | version == "10" = palParParallel10 word 0 ((length word) - 1)
  | version == "11" = palParParallel11 word 0 ((length word) - 1) 4 -- Change this last number to change depth. Defa
ult = 4
  | version == "12" = palParParallel12 word 0 ((length word) - 1) 4 -- Change this last number to change depth. Defa
ult = 4
  | version == "13" = palParParallel13 word 0 ((length word) - 1) 4 -- Change this last number to change depth. Defa
ult = 4
  | version == "14" = palParParallel14 word 0 ((length word) - 1) 4 -- Change this last number to change depth. Defa
ult = 4
  | otherwise       = palParParallel1  word 0 ((length word) - 1)

-- Helper function for parListChunk
withChunk' :: Int -> Strategy y -> (x -> y) -> [x] -> [y]
withChunk' c' s' f' = withStrategy (parListChunk c' s') . map f'

-- Helper function for parList rdeepseq
parMapDeepSeq' :: (NFData y) => (x -> y) -> [x] -> [y]
parMapDeepSeq' f' = withStrategy (parList rdeepseq) . map f'

-- Helper function for parBuffer rdeepseq
parBufferMapDeepSeq' :: (NFData y) => (x -> y) -> [x] -> [y]
parBufferMapDeepSeq' f' = withStrategy (parBuffer 500 rdeepseq) . map f'

-- Version 1
-- One layer of parallelization
palParParallel1 :: String -> Int -> Int -> Int
palParParallel1 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (parMap rpar (\m -> palParLoop1' word l m r) [l..(r - 1)])

palParLoop1' :: String -> Int -> Int -> Int -> Int
palParLoop1' word l m r = 1 + (palParParallel1 word l m) + (palParParallel1 word (m + 1) r)

-- Version 2
-- Two layers of parallelization
palParParallel2 :: String -> Int -> Int -> Int
palParParallel2 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (parMap rpar (\m -> palParLoop2' word l m r) [l..(r - 1)])

palParLoop2' :: String -> Int -> Int -> Int -> Int
palParLoop2' word l m r = parCall1 `par` parCall2 `pseq` parCall1 + parCall2 + 1
  where
    parCall1 = palParParallel2 word l m
    parCall2 = palParParallel2 word (m + 1) r

-- Version 3
```

```
-- One layer of parallelization and chunking based on bottleneck size (which starts around 16 characters)
palParParallel3 :: String -> Int -> Int -> Int
palParParallel3 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (withChunk' 4 rseq (\m -> palParLoop1' word l m r) [l..(r - 1)])

-- Version 4
-- Two layers of parallelization and chunking based on bottleneck size (which starts around 16 characters)
palParParallel4 :: String -> Int -> Int -> Int
palParParallel4 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise  = minimum (withChunk' 4 rseq (\m -> palParLoop2' word l m r) [l..(r - 1)])

-- Version 5
-- One layer of parallelization with depth
palParParallel5 :: String -> Int -> Int -> Int -> Int
palParParallel5 word l r d
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | d <= 0    = minimum (map (\m -> palParLoop5' word l m r d) [l..(r - 1)])
  | otherwise = minimum (parMap rpar (\m -> palParLoop5' word l m r (d - 1)) [l..(r - 1)])

palParLoop5' :: String -> Int -> Int -> Int -> Int -> Int
palParLoop5' word l m r d = 1 + (palParParallel5 word l m d) + (palParParallel5 word (m + 1) r d)

-- Version 6
-- Two layers of parallelization with depth
palParParallel6 :: String -> Int -> Int -> Int -> Int
palParParallel6 word l r d
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | d <= 0    = minimum (map (\m -> palParLoop6' word l m r d) [l..(r - 1)])
  | otherwise = minimum (parMap rpar (\m -> palParLoop6' word l m r (d - 1)) [l..(r - 1)])

palParLoop6' :: String -> Int -> Int -> Int -> Int -> Int
palParLoop6' word l m r d = parCall1 `par` parCall2 `pseq` parCall1 + parCall2 + 1
    where
      parCall1 = palParParallel6 word l m d
      parCall2 = palParParallel6 word (m + 1) r d

-- Version 7
-- One layer of parallelization using rdeepseq strategy
palParParallel7 :: String -> Int -> Int -> Int
palParParallel7 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (parMapDeepSeq' (\m -> palParLoop7' word l m r) [l..(r - 1)])

palParLoop7' :: String -> Int -> Int -> Int -> Int
palParLoop7' word l m r = 1 + (palParParallel7 word l m) + (palParParallel7 word (m + 1) r)

-- Version 8
-- Two layers of parallelization using rdeepseq strategy
```

```
palParParallel8 :: String -> Int -> Int -> Int
palParParallel8 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (parMapDeepSeq' (\m -> palParLoop8' word l m r) [l..(r - 1)])

palParLoop8' :: String -> Int -> Int -> Int -> Int
palParLoop8' word l m r = parCall1 `par` parCall2 `pseq` parCall1 + parCall2 + 1
  where
    parCall1 = palParParallel8 word l m
    parCall2 = palParParallel8 word (m + 1) r

-- Version 9
-- One layer of parallelization using rdeepseq strategy and buffering
palParParallel9 :: String -> Int -> Int -> Int
palParParallel9 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (parBufferMapDeepSeq' (\m -> palParLoop9' word l m r) [l..(r - 1)])

palParLoop9' :: String -> Int -> Int -> Int -> Int
palParLoop9' word l m r = 1 + (palParParallel9 word l m) + (palParParallel9 word (m + 1) r)

-- Version 10
-- Two layers of parallelization using rdeepseq strategy and buffering
palParParallel10 :: String -> Int -> Int -> Int
palParParallel10 word l r
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | otherwise = minimum (parBufferMapDeepSeq' (\m -> palParLoop10' word l m r) [l..(r - 1)])

palParLoop10' :: String -> Int -> Int -> Int -> Int
palParLoop10' word l m r = parCall1 `par` parCall2 `pseq` parCall1 + parCall2 + 1
  where
    parCall1 = palParParallel10 word l m
    parCall2 = palParParallel10 word (m + 1) r

-- Version 11
-- One layer of parallelization using rdeepseq strategy and depth
palParParallel11 :: String -> Int -> Int -> Int -> Int
palParParallel11 word l r d
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | d <= 0    = minimum (map (\m -> palParLoop11' word l m r d) [l..(r - 1)])
  | otherwise = minimum (parMapDeepSeq' (\m -> palParLoop11' word l m r (d - 1)) [l..(r - 1)])

palParLoop11' :: String -> Int -> Int -> Int -> Int -> Int
palParLoop11' word l m r d = 1 + (palParParallel11 word l m d) + (palParParallel11 word (m + 1) r d)

-- Version 12
-- Two layers of parallelization using rdeepseq strategy and depth
palParParallel12 :: String -> Int -> Int -> Int -> Int
palParParallel12 word l r d
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
```

```
  | d <= 0    = minimum (map (\m -> palParLoop12' word l m r d) [l..(r - 1)])
  | otherwise = minimum (parMapDeepSeq' (\m -> palParLoop12' word l m r (d - 1)) [l..(r - 1)])

palParLoop12' :: String -> Int -> Int -> Int -> Int
palParLoop12' word l m r d = parCall1 `par` parCall2 `pseq` parCall1 + parCall2 + 1
  where
    parCall1 = palParParallel12 word l m d
    parCall2 = palParParallel12 word (m + 1) r d

-- Version 13
-- One layer of parallelization using rdeepseq strategy, depth, and buffering
palParParallel13 :: String -> Int -> Int -> Int -> Int
palParParallel13 word l r d
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | d <= 0    = minimum (map (\m -> palParLoop13' word l m r d) [l..(r - 1)])
  | otherwise = minimum (parBufferMapDeepSeq' (\m -> palParLoop13' word l m r (d - 1)) [l..(r - 1)])

palParLoop13' :: String -> Int -> Int -> Int -> Int -> Int
palParLoop13' word l m r d = 1 + (palParParallel13 word l m d) + (palParParallel13 word (m + 1) r d)

-- Version 14
-- Two layers of parallelization using rdeepseq strategy, depth, and buffering
palParParallel14 :: String -> Int -> Int -> Int -> Int
palParParallel14 word l r d
  | l >= r = 0
  | (isPalindrome word l r) == True = 0
  | d <= 0    = minimum (map (\m -> palParLoop14' word l m r d) [l..(r - 1)])
  | otherwise = minimum (parBufferMapDeepSeq' (\m -> palParLoop14' word l m r (d - 1)) [l..(r - 1)])

palParLoop14' :: String -> Int -> Int -> Int -> Int -> Int
palParLoop14' word l m r d = parCall1 `par` parCall2 `pseq` parCall1 + parCall2 + 1
  where
    parCall1 = palParParallel14 word l m d
    parCall2 = palParParallel14 word (m + 1) r d


-----------------------------------------


{-

 Name: Jesse Chong
 Uni: jlc2332
 Final Project: Palindrome Partitioning
 File Name: PalParSpec.hs

-}

import Test.HUnit(Test, Test(TestCase, TestLabel, TestList),
                  assertEqual, runTestTT)

import PalParSequential(palParSequential)
import PalParParallel(palParParallel)
```

```
-- Helper function to read a one word file for testing purposes
readFileHelper' :: String -> String -> String -> IO (Int)
readFileHelper' filename mode version
  | mode == "s" = do
    word <- readFile filename
    let result = palParSequential word version
    return result
  | otherwise = do
    word <- readFile filename
    let result = palParParallel word version
    return result

testSequentialNaive1 :: Test
testSequentialNaive1 = TestCase $ do
  let expected = 0
  actual <- (readFileHelper' "./test/ValidCases/alphabet1.txt" "s" "1")
  assertEqual "testSequentialNaive1" expected actual

testSequentialNaive2 :: Test
testSequentialNaive2 = TestCase $ do
  let expected = 3
  actual <- (readFileHelper' "./test/ValidCases/medium1.txt" "s" "1")
  assertEqual "testSequentialNaive2" expected actual

testSequentialNaive3 :: Test
testSequentialNaive3 = TestCase $ do
  let expected = 0
  actual <- (readFileHelper' "./test/ValidCases/palindrome1.txt" "s" "1")
  assertEqual "testSequentialNaive3" expected actual

testSequentialNaive4 :: Test
testSequentialNaive4 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "s" "1")
  assertEqual "testSequentialNaive4" expected actual

testSequentialNaive5 :: Test
testSequentialNaive5 = TestCase $ do
  let expected = 15
  actual <- (readFileHelper' "./test/ValidCases/alphabet16.txt" "s" "1")
  assertEqual "testSequentialNaive5" expected actual

testSequentialDP1 :: Test
testSequentialDP1 = TestCase $ do
  let expected = 0
  actual <- (readFileHelper' "./test/ValidCases/alphabet1.txt" "s" "2")
  assertEqual "testSequentialDP1" expected actual

testSequentialDP2 :: Test
testSequentialDP2 = TestCase $ do
  let expected = 3
  actual <- (readFileHelper' "./test/ValidCases/medium1.txt" "s" "2")
  assertEqual "testSequentialDP2" expected actual

testSequentialDP3 :: Test
```

```
testSequentialDP3 = TestCase $ do
  let expected = 0
  actual <- (readFileHelper' "./test/ValidCases/palindrome1.txt" "s" "2")
  assertEqual "testSequentialDP3" expected actual

testSequentialDP4 :: Test
testSequentialDP4 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "s" "2")
  assertEqual "testSequentialDP4" expected actual

testSequentialDP5 :: Test
testSequentialDP5 = TestCase $ do
  let expected = 15
  actual <- (readFileHelper' "./test/ValidCases/alphabet16.txt" "s" "2")
  assertEqual "testSequentialDP5" expected actual

testSequentialList :: Test
testSequentialList = TestList [TestLabel "testSequentialNaive1" testSequentialNaive1,
                               TestLabel "testSequentialNaive2" testSequentialNaive2,
                               TestLabel "testSequentialNaive3" testSequentialNaive3,
                               TestLabel "testSequentialNaive4" testSequentialNaive4,
                               TestLabel "testSequentialNaive5" testSequentialNaive5,
                               TestLabel "testSequentialDP1"    testSequentialDP1,
                               TestLabel "testSequentialDP2"    testSequentialDP2,
                               TestLabel "testSequentialDP3"    testSequentialDP3,
                               TestLabel "testSequentialDP4"    testSequentialDP4,
                               TestLabel "testSequentialDP5"    testSequentialDP5]

testParallelVOne1 :: Test
testParallelVOne1 = TestCase $ do
  let expected = 0
  actual <- (readFileHelper' "./test/ValidCases/alphabet1.txt" "p" "1")
  assertEqual "testParallelVOne1" expected actual

testParallelVOne2 :: Test
testParallelVOne2 = TestCase $ do
  let expected = 3
  actual <- (readFileHelper' "./test/ValidCases/medium1.txt" "p" "1")
  assertEqual "testParallelVOne2" expected actual

testParallelVOne3 :: Test
testParallelVOne3 = TestCase $ do
  let expected = 0
  actual <- (readFileHelper' "./test/ValidCases/palindrome1.txt" "p" "1")
  assertEqual "testParallelVOne3" expected actual

testParallelVOne4 :: Test
testParallelVOne4 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "1")
  assertEqual "testParallelVOne4" expected actual

testParallelVOne5 :: Test
testParallelVOne5 = TestCase $ do
```

```
  let expected = 15
  actual <- (readFileHelper' "./test/ValidCases/alphabet16.txt" "p" "1")
  assertEqual "testParallelVOne5" expected actual

testParallelVTwo1 :: Test
testParallelVTwo1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "2")
  assertEqual "testParallelVTwo1" expected actual

testParallelVTwo2 :: Test
testParallelVTwo2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "2")
  assertEqual "testParallelVTwo2" expected actual

testParallelVThree1 :: Test
testParallelVThree1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "3")
  assertEqual "testParallelVThree1" expected actual

testParallelVThree2 :: Test
testParallelVThree2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "3")
  assertEqual "testParallelVThree2" expected actual

testParallelVFour1 :: Test
testParallelVFour1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "4")
  assertEqual "testParallelVFour1" expected actual

testParallelVFour2 :: Test
testParallelVFour2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "4")
  assertEqual "testParallelVFour2" expected actual

testParallelVFive1 :: Test
testParallelVFive1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "5")
  assertEqual "testParallelVFive1" expected actual

testParallelVFive2 :: Test
testParallelVFive2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "5")
  assertEqual "testParallelVFive2" expected actual

testParallelVSix1 :: Test
testParallelVSix1 = TestCase $ do
  let expected = 6
```

```
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "6")
  assertEqual "testParallelVSix1" expected actual

testParallelVSix2 :: Test
testParallelVSix2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "6")
  assertEqual "testParallelVSix2" expected actual

testParallelVSeven1 :: Test
testParallelVSeven1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "7")
  assertEqual "testParallelVSeven1" expected actual

testParallelVSeven2 :: Test
testParallelVSeven2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "7")
  assertEqual "testParallelVSeven2" expected actual

testParallelVEight1 :: Test
testParallelVEight1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "8")
  assertEqual "testParallelVEight1" expected actual

testParallelVEight2 :: Test
testParallelVEight2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "8")
  assertEqual "testParallelVEight2" expected actual

testParallelVNine1 :: Test
testParallelVNine1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "9")
  assertEqual "testParallelVNine1" expected actual

testParallelVNine2 :: Test
testParallelVNine2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "9")
  assertEqual "testParallelVNine2" expected actual

testParallelVTen1 :: Test
testParallelVTen1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "10")
  assertEqual "testParallelVTen1" expected actual

testParallelVTen2 :: Test
testParallelVTen2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "10")
```

```
  assertEqual "testParallelVTen2" expected actual

testParallelVEleven1 :: Test
testParallelVEleven1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "11")
  assertEqual "testParallelVEleven1" expected actual

testParallelVEleven2 :: Test
testParallelVEleven2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "11")
  assertEqual "testParallelVEleven2" expected actual

testParallelVTwelve1 :: Test
testParallelVTwelve1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "12")
  assertEqual "testParallelVTwelve1" expected actual

testParallelVTwelve2 :: Test
testParallelVTwelve2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "12")
  assertEqual "testParallelVTwelve2" expected actual

testParallelVThirteen1 :: Test
testParallelVThirteen1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "13")
  assertEqual "testParallelVThirteen1" expected actual

testParallelVThirteen2 :: Test
testParallelVThirteen2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "13")
  assertEqual "testParallelVThirteen2" expected actual

testParallelVFourteen1 :: Test
testParallelVFourteen1 = TestCase $ do
  let expected = 6
  actual <- (readFileHelper' "./test/ValidCases/medium3.txt" "p" "14")
  assertEqual "testParallelVFourteen1" expected actual

testParallelVFourteen2 :: Test
testParallelVFourteen2 = TestCase $ do
  let expected = 7
  actual <- (readFileHelper' "./test/ValidCases/alphabet8.txt" "p" "14")
  assertEqual "testParallelVFourteen2" expected actual

testParallelList :: Test
testParallelList = TestList [TestLabel "testParallelVOne1"      testParallelVOne1,
                             TestLabel "testParallelVOne2"      testParallelVOne2,
                             TestLabel "testParallelVOne3"      testParallelVOne3,
                             TestLabel "testParallelVOne4"      testParallelVOne4,
```

```
                    TestLabel "testParallelVOne5"      testParallelVOne5,
                    TestLabel "testParallelVTwo1"      testParallelVTwo1,
                    TestLabel "testParallelVTwo2"      testParallelVTwo2,
                    TestLabel "testParallelVThree1"    testParallelVThree1,
                    TestLabel "testParallelVThree2"    testParallelVThree2,
                    TestLabel "testParallelVFour1"     testParallelVFour1,
                    TestLabel "testParallelVFour2"     testParallelVFour2,
                    TestLabel "testParallelVFive1"     testParallelVFive1,
                    TestLabel "testParallelVFive2"     testParallelVFive2,
                    TestLabel "testParallelVSix1"      testParallelVSix1,
                    TestLabel "testParallelVSix2"      testParallelVSix2,
                    TestLabel "testParallelVSeven1"    testParallelVSeven1,
                    TestLabel "testParallelVSeven2"    testParallelVSeven2,
                    TestLabel "testParallelVEight1"    testParallelVEight1,
                    TestLabel "testParallelVEight2"    testParallelVEight2,
                    TestLabel "testParallelVNine1"     testParallelVNine1,
                    TestLabel "testParallelVNine2"     testParallelVNine2,
                    TestLabel "testParallelVTen1"      testParallelVTen1,
                    TestLabel "testParallelVTen2"      testParallelVTen2,
                    TestLabel "testParallelVEleven1"   testParallelVEleven1,
                    TestLabel "testParallelVEleven2"   testParallelVEleven2,
                    TestLabel "testParallelVTwelve1"   testParallelVTwelve1,
                    TestLabel "testParallelVTwelve2"   testParallelVTwelve2,
                    TestLabel "testParallelVThirteen1" testParallelVThirteen1,
                    TestLabel "testParallelVThirteen2" testParallelVThirteen2,
                    TestLabel "testParallelVFourteen1" testParallelVFourteen1,
                    TestLabel "testParallelVFourteen2" testParallelVFourteen2]

main :: IO ()
main = do
  {- Doing these throw-away binds because of the following warning:
      " A do-notation statement discarded a result of type ‘Counts’
        Suppress this warning by saying ‘_ <- runTestTT testList’ "   -}
  putStrLn ""
  putStrLn "Running testSequentialList"
  _ <- runTestTT testSequentialList
  -- All the parallel algorithms are identical hence why V2 to V14 have only 2 test cases compared to V1
  putStrLn "Running testParallelList"
  _ <- runTestTT testParallelList
  putStrLn ""
  return ()
```

Memory Used in B (Max Resid	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	119176	85176	488888	574568	549968	682120	320040	574672	484272	515656	491376	489048	326040	442064	348592	512312
TC alphabet20.txt	123360	85176	489904	520160	529512	548248	316944	412504	527736	544888	558232	632568	324480	478136	313208	417496
TC alphabet20.txt	123360	85176	450848	545600	554960	513664	316528	604920	583704	546968	604856	541072	323488	414608	350400	510912
TC alphabet20.txt	108672	85176	422424	544944	502064	521640	322560	478768	512800	515680	492368	480640	316504	418000	376936	479248
TC alphabet20.txt	123360	85176	482928	578088	516008	581544	322768	475368	572432	495752	518472	525496	351384	377608	353008	446936
Average	119585.6	85176	466998.4	552672	530502.4	569443.2	319768	509246.4	536188.8	523788.8	533060.8	533764.8	328379.2	426083.2	348428.8	473380.8
Total Time Elapsed (seconds)	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	24.922	21.831	18.83	50.97	18.1	46.523	7.721	16.492	18.751	19.401	21.901	21.821	12.01	19.161	12.371	19.421
TC alphabet20.txt	32.750	20.991	16.453	35.64	24.731	45.92	10.801	20.34	22.901	27.331	22.301	29.271	10.531	24.961	11.051	20.541
TC alphabet20.txt	32.530	20.861	26.611	45.191	24.09	48.301	8.271	21.921	16.581	23.404	25.85	25.73	10.831	22.471	13.951	16.49
TC alphabet20.txt	32.101	21.141	22.631	45.84	24.3	49.08	11.111	23.62	19.511	24.771	21.602	25.231	15.47	23.421	13.251	22.813
TC alphabet20.txt	30.631	21.151	21.481	56.52	23.601	44.96	8.061	16.56	17.134	24.441	16.872	24.702	11.01	22.32	12.972	22.121
Average	30.5868	21.195	21.2012	46.8322	22.9644	46.9568	**9.193**	19.7866	18.9756	23.8696	21.7052	25.351	**11.9704**	22.4668	**12.7192**	20.2772
Sparks Converted	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	0	0	4686	509410	12069	640430	2381	42705	3205	2180	5308	7379	1809	39555	466	51291
TC alphabet20.txt	0	0	6781	488123	11555	641876	458	49432	7219	1654	5965	4190	1632	52121	329	48636
TC alphabet20.txt	0	0	6664	476158	6130	634491	1839	35016	3759	3375	6310	7882	1518	62529	1158	46333
TC alphabet20.txt	0	0	5235	576369	15737	634071	3291	48186	2855	1559	2092	2072	370	56251	404	60664
TC alphabet20.txt	0	0	1397	594480	5922	597604	3844	45070	6435	3691	4171	7921	1954	63524	1476	47479
Average	0	0	4952.6	528908	10282.6	629694.4	2362.6	44081.8	4694.6	2491.8	4769.2	5888.8	1456.6	54796	766.6	50880.6
Sparks Overflowed	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	0	0	0	7108	0	5682	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	4971	0	5407	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	5061	0	4893	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	5187	0	5435	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	5567	0	5260	0	0	0	0	0	0	0	0	0	0
Average	0	0	0	5578.8	0	5335.4	0	0	0	0	0	0	0	0	0	0
Sparks Dud	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Average	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sparks GC'd	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	0	0	1157499578	2438377348	1158470955	1991753157	48410	573861647	578254269	1159234985	577561265	1155149220	17040	573459716	22522	572072827
TC alphabet20.txt	0	0	1156454465	1947377372	1154264626	1860558432	51828	573284617	579286691	1158556785	579298297	1158783513	16989	571279809	22833	594405757
TC alphabet20.txt	0	0	1156284977	2008715800	1156365750	1744509474	49345	574250184	578606056	1161770874	579472572	1169864161	17442	572719799	17242	573658402
TC alphabet20.txt	0	0	1157196789	2001681303	1151020939	2001391105	35540	571939401	578470808	1158314025	578817541	1157961170	23091	572283553	20314	572715145
TC alphabet20.txt	0	0	1159648878	1865026839	1156668700	1989891089	34051	572428829	577820614	1157010748	578210907	1155709615	16537	572391438	17434	571758427
Average	0	0	1157416937	2052235732	1155358194	1917620651	43834.8	573152935.6	578487687.6	1158977483	578672116.4	1159493536	18219.8	572426863	20069	576922111.6
Sparks Fizzled	SeqNaive (N4)	SeqNaive (N1)	Par1 (N4)	Par2 (N4)	Par3 (N4)	Par4 (N4)	Par5 (N4)	Par6 (N4)	Par7 (N4)	Par8 (N4)	Par9 (N4)	Par10 (N4)	Par11 (N4)	Par12 (N4)	Par13 (N4)	Par14 (N4)
TC alphabet20.txt	0	0	5513711	356468186	10566164	423103297	19993	12988185	3524666	4582207	4371470	9640343	18309	13347527	12364	14996896
TC alphabet20.txt	0	0	6958661	341066845	10346966	425186837	18204	12960224	5886130	4344771	5028808	6278057	18455	16942302	12083	15256146
TC alphabet20.txt	0	0	7055856	332519590	6773540	414566777	19416	11790800	3465758	6918009	4997979	9710210	17825	14603859	18085	12737445
TC alphabet20.txt	0	0	5924169	389607600	13869779	427294430	34565	15751346	3314610	4636384	2567178	4812524	11784	15472997	15130	14540546
TC alphabet20.txt	0	0	2772913	389192597	6571392	404767115	36285	14994014	5158708	7080777	3608653	9309658	18647	15126560	18056	15899012
Average	0	0	5645062	361770963.6	9625568.2	418983691.2	25692.6	13696913.8	4269974.4	5512429.6	4114817.6	7950158.4	17004	15098649	15143.6	14686009

Memory Used in B (Max Resid	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	92192	222000	320040	471152	530216	794632	1039672
TC alphabet20.txt	92904	250128	316944	479392	563304	821840	1060552
TC alphabet20.txt	91904	222112	316528	468368	695728	914280	1192664
Average	92333.33333	231413.3333	317837.3333	472970.6667	596416	843584	1097629.333
Total Time Elapsed (seconds)	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	24.111	12.322	7.721	13.001	12.131	17.061	28.261
TC alphabet20.txt	23.36	12.481	10.801	14.831	15.171	19.391	24.231
TC alphabet20.txt	23.492	13.442	8.271	16.393	17.041	24.641	28.401
Average	23.65433333	12.74833333	**8.931**	14.74166667	14.781	20.36433333	26.246
Sparks Converted	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	0	132	2381	1008	1805	7078	3076

TC alphabet20.txt	0	132	458	1102	1031	6609	1158
TC alphabet20.txt	0	132	1839	3961	11285	7626	14250
Average	0	132	1559.333333	2023.666667	4707	7104.333333	6161.333333
Sparks Overflowed	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0
Average	0	0	0	0	0	0	0
Sparks Dud	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0
TC alphabet20.txt	0	0	0	0	0	0	0
Average	0	0	0	0	0	0	0
Sparks GC'd	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	54906	52708	48410	50530	42290	33204	47351
TC alphabet20.txt	54898	52668	51828	45185	46104	33344	49766
TC alphabet20.txt	54899	52632	49345	38528	24106	31573	22923
Average	54901	52669.33333	49861	44747.66667	37500	32707	40013.33333
Sparks Fizzled	Par5 (N1)	Par5 (N2)	Par5 (N4)	Par5 (N6)	Par5 (N8)	Par5 (N12)	Par5 (N16)
TC alphabet20.txt	15584	17650	19993	18952	27075	32421	20262
TC alphabet20.txt	15592	17690	18204	24415	23355	31359	19566
TC alphabet20.txt	15591	17726	19416	28839	37014	32676	35064
Average	15589	17688.66667	19204.33333	24068.66667	29148	32152	24964

Memory Used in B (Max Resid	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	66480	104256	66504
TC medium1.txt	84408	227712	66528
TC medium2.txt	66504	104232	66528
TC alphabet1.txt	66576	104464	66576
TC alphabet2.txt	66576	104464	66576
TC alphabet4.txt	66576	101160	66576
TC alphabet6.txt	66576	99440	66576
TC alphabet8.txt	66576	99368	66576
TC alphabet16.txt	84840	201840	66600
TC alphabet17.txt	84960	316376	66600
TC alphabet18.txt	85080	278008	66600
TC alphabet19.txt	85200	325208	66600
TC alphabet20.txt	85176	321536	66600
TC alphabet21.txt	85440	347560	66600
Average	75783.42857	195401.7143	66574.28571
Total Time Elapsed (seconds)	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	0.011	0.011	0.011
TC medium1.txt	0.021	0.011	0.013
TC medium2.txt	0.011	0.011	0.01
TC alphabet1.txt	0.011	0.011	0.012
TC alphabet2.txt	0.011	0.011	0.013
TC alphabet4.txt	0.011	0.011	0.011
TC alphabet6.txt	0.011	0.01	0.014
TC alphabet8.txt	0.011	0.01	0.01
TC alphabet16.txt	0.252	0.111	0.01
TC alphabet17.txt	0.911	0.272	0.011
TC alphabet18.txt	2.471	1.262	0.012
TC alphabet19.txt	7.041	3.491	0.012
TC alphabet20.txt	23.481	8.221	0.011
TC alphabet21.txt	64.562	24.621	0.01
Average	7.058285714	2.718857143	0.01142857143
Sparks Converted	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	0	0	0
TC medium1.txt	0	661	0
TC medium2.txt	0	10	0
TC alphabet1.txt	0	0	0
TC alphabet2.txt	0	0	0

TC alphabet4.txt	0	0	0
TC alphabet6.txt	0	0	0
TC alphabet8.txt	0	10	0
TC alphabet16.txt	0	726	0
TC alphabet17.txt	0	2044	0
TC alphabet18.txt	0	652	0
TC alphabet19.txt	0	2758	0
TC alphabet20.txt	0	4197	0
TC alphabet21.txt	0	2449	0
Average	0	964.7857143	0
Sparks Overflowed	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	0	0	0
TC medium1.txt	0	0	0
TC medium2.txt	0	0	0
TC alphabet1.txt	0	0	0
TC alphabet2.txt	0	0	0
TC alphabet4.txt	0	0	0
TC alphabet6.txt	0	0	0
TC alphabet8.txt	0	0	0
TC alphabet16.txt	0	0	0
TC alphabet17.txt	0	0	0
TC alphabet18.txt	0	0	0
TC alphabet19.txt	0	0	0
TC alphabet20.txt	0	0	0
TC alphabet21.txt	0	0	0
Average	0	0	0
Sparks Dud	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	0	0	0
TC medium1.txt	0	0	0
TC medium2.txt	0	0	0
TC alphabet1.txt	0	0	0
TC alphabet2.txt	0	0	0
TC alphabet4.txt	0	0	0
TC alphabet6.txt	0	0	0
TC alphabet8.txt	0	0	0
TC alphabet16.txt	0	0	0
TC alphabet17.txt	0	0	0
TC alphabet18.txt	0	0	0
TC alphabet19.txt	0	0	0
TC alphabet20.txt	0	0	0
TC alphabet21.txt	0	0	0
Average	0	0	0
Sparks GC'd	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	0	40	0
TC medium1.txt	0	2904	0
TC medium2.txt	0	0	0
TC alphabet1.txt	0	0	0
TC alphabet2.txt	0	0	0
TC alphabet4.txt	0	0	0
TC alphabet6.txt	0	0	0
TC alphabet8.txt	0	0	0
TC alphabet16.txt	0	16830	0
TC alphabet17.txt	0	16019	0
TC alphabet18.txt	0	33832	0
TC alphabet19.txt	0	29754	0
TC alphabet20.txt	0	34254	0
TC alphabet21.txt	0	52175	0
Average	0	13272	0
Sparks Fizzled	SeqNaive (N1)	Par5 (N4)	SeqDP (N1)
TC small1.txt	0	0	0
TC medium1.txt	0	8836	0
TC medium2.txt	0	1488	0

TC alphabet1.txt	0	0	0	
TC alphabet2.txt	0	2	0	
TC alphabet4.txt	0	26	0	
TC alphabet6.txt	0	210	0	
TC alphabet8.txt	0	1548	0	
TC alphabet16.txt	0	9211	0	
TC alphabet17.txt	0	18704	0	
TC alphabet18.txt	0	9703	0	
TC alphabet19.txt	0	26731	0	
TC alphabet20.txt	0	34490	0	
TC alphabet21.txt	0	33867	0	
Average	0	10344	0	

Memory Used in B (Max Resid	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	127600	393472	443488	170456
TC multiline2.txt (18 Lines of ra	86968	227376	232768	66600
TC slowMultiline1.txt (11 Lines	90120	416000	403880	365080
TC slowMultiline2.txt (6 Lines c	87760	388368	386816	375912
TC slowMultiline3.txt (3 Lines c	86344	353944	376128	57000
Average	95758.4	355832	368616	207009.6
Total Time Elapsed (seconds)	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	2.082	0.932	0.781	0.011
TC multiline2.txt (18 Lines of ra	0.022	0.021	0.02	0.011
TC slowMultiline1.txt (11 Lines	225.081	135.811	149.691	0.011
TC slowMultiline2.txt (6 Lines c	139.371	57.76	76.351	0.011
TC slowMultiline3.txt (3 Lines c	72.541	42.83	34.01	0.011
Average	87.8194	47.4708	52.1706	0.011
Sparks Converted	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	0	22087	2118	0
TC multiline2.txt (18 Lines of ra	0	561	448	0
TC slowMultiline1.txt (11 Lines	0	6469	7311	0
TC slowMultiline2.txt (6 Lines c	0	20051	5910	0
TC slowMultiline3.txt (3 Lines c	0	11051	1706	0
Average	0	12043.8	3498.6	0
Sparks Overflowed	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	0	0	0	0
TC multiline2.txt (18 Lines of ra	0	0	0	0
TC slowMultiline1.txt (11 Lines	0	0	0	0
TC slowMultiline2.txt (6 Lines c	0	0	0	0
TC slowMultiline3.txt (3 Lines c	0	0	0	0
Average	0	0	0	0
Sparks Dud	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	0	0	0	0
TC multiline2.txt (18 Lines of ra	0	0	0	0
TC slowMultiline1.txt (11 Lines	0	0	0	0
TC slowMultiline2.txt (6 Lines c	0	0	0	0
TC slowMultiline3.txt (3 Lines c	0	0	0	0
Average	0	0	0	0
Sparks GC'd	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	0	449366	715103	0
TC multiline2.txt (18 Lines of ra	0	11443	14404	0
TC slowMultiline1.txt (11 Lines	0	551736	534619	0
TC slowMultiline2.txt (6 Lines c	0	222914	297214	0
TC slowMultiline3.txt (3 Lines c	0	109255	158708	0
Average	0	268942.8	344009.6	0
Sparks Fizzled	SeqNaive (N1)	Par5 (N4) (Sequ	Par5 (N4) (Parall	SeqDP (N1)
TC multiline1.txt (210 Lines of	0	471263	148480	0
TC multiline2.txt (18 Lines of ra	0	20389	14881	0
TC slowMultiline1.txt (11 Lines	0	218445	236459	0
TC slowMultiline2.txt (6 Lines c	0	194060	122137	0
TC slowMultiline3.txt (3 Lines c	0	99830	51303	0
Average	0	200797.4	114652	0

Memory Used in B (Max Resid	SeqDP (N1)
TC alphabet1.txt	66576
TC alphabet2.txt	66576
TC alphabet4.txt	66576
TC alphabet6.txt	66576
TC alphabet8.txt	66576
TC alphabet16.txt	66600
TC alphabet17.txt	66600
TC alphabet18.txt	66600
TC alphabet19.txt	66600
TC alphabet20.txt	66600
TC alphabet21.txt	66600
TC alphabet24.txt	57000
TC alphabet35.txt	5013896
TC alphabet40.txt	9052496
TC alphabet45.txt	8902464
TC alphabet60.txt	26268208
TC alphabet70.txt	68564976
TC alphabet100.txt	287471664
TC alphabet130.txt	1109463456
Average	79764560
Total Time Elapsed (seconds)	SeqDP (N1)
TC alphabet1.txt	0.012
TC alphabet2.txt	0.013
TC alphabet4.txt	0.011
TC alphabet6.txt	0.014
TC alphabet8.txt	0.01
TC alphabet16.txt	0.01
TC alphabet17.txt	0.011
TC alphabet18.txt	0.012
TC alphabet19.txt	0.012
TC alphabet20.txt	0.011
TC alphabet21.txt	0.01
TC alphabet24.txt	0.013
TC alphabet35.txt	0.011
TC alphabet40.txt	0.022
TC alphabet45.txt	0.031
TC alphabet60.txt	0.071
TC alphabet70.txt	0.171
TC alphabet100.txt	0.69
TC alphabet130.txt	2.31
Average	0.1813157895