Parallel Functional Programming

Word Search in a Grid

Name : Swetha Shanmugam
UNI: ss6357

Introduction

Given a word and a grid, the problem statement is to find all occurrences of the word. The word can be present in four directions - Horizontally to the right or left, and vertically down or up. Let's consider the below example:

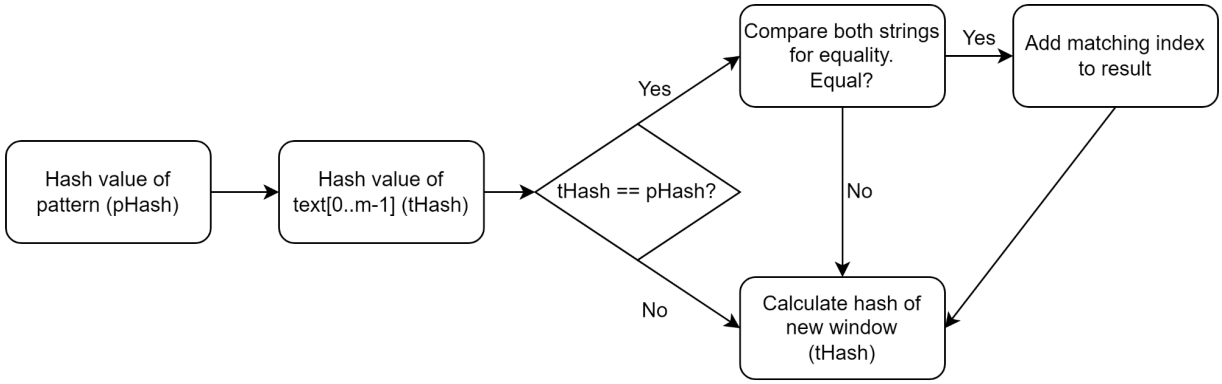
S	F	J	**U**	**P**
F	D	G	A	C
B	**U**	G	B	V
V	**P**	F	N	**P**
L	**P**	**U**	J	**U**

Given the word "UP", we need to look for all its occurrences. In the above example, there are totally 4 occurrences of "UP".

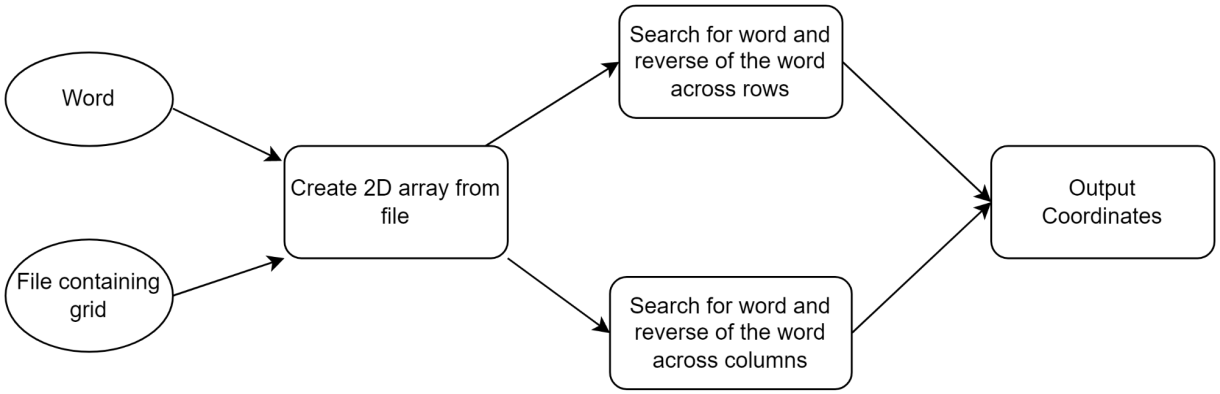
In this project, I've written a sequential and parallel implementation of a Haskell program to find a word in a grid.

Algorithm

In order to perform the word search, I've implemented the Rabin-Karp string-matching algorithm. Given below is the code flow of the Rabin-Karp algorithm implemented:



Given below is the main code flow of the program:



- Given the filename and word to search, a 2D array of the input grid is created.
- The reverse of the word is computed. If the word is present horizontally from right to left, then the reverse is present from right to left.
- The word and it's reverse are searched in every row of the grid using Rabin Karp algorithm.
- The word and it's reverse are searched in every column of the grid using Rabin Karp algorithm.

Sample Results

Let's run the program on the sample grid example shown above for the word "UP". Results are:

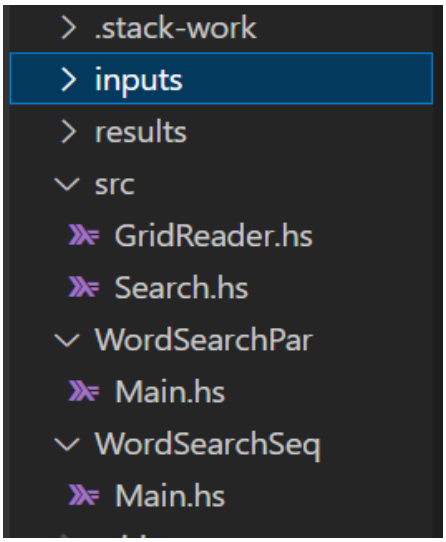
```
"Row matches found at coordinates:"
(0,3)
(4,2)
```

```
"Column matches found at coordinates:"
(2,1)
(4,4)
```

For the word “DOWN”, results are:

```
"Pattern not found across row"
"Pattern not found across column"
```

Project Structure



The project is built using ‘stack’.

- The `WordSearchSeq` folder contains the main file for the sequential execution of the program.
- The `WordSearchPar` folder contains the main file for the parallel execution of the program.
- The `src` folder contains two modules - `GridReader` which creates the grid from an input file, and the Search module which has the Rabin Karp algorithm to search for the word in the grid.
- The `inputs` folder contains sample input grids.
- The `results` folder contains the results.

Code

WordSearchSeq/Main.hs

```
module Main (main) where

import Search(rabinKarpMain)
import GridReader(createGridFromInput)


import System.Environment ( getArgs )
import System.Exit ( die )
```

```haskell
import Control.Monad ( when)
import qualified Data.ByteString as S
import Data.ByteString.Char8 as C8 (pack)
import Data.Char (toUpper, isAlpha)



main :: IO ()
main = do
  args <- getArgs
  case args of
    [patt, nVal, filename] -> do
      let n = (read nVal :: Int)
      let m = length patt

      when(m < 2) $
        die "Pattern must be of length >= 2"

      when(m > n) $
        die "Pattern length must be less than or equal to grid size"

      when (False `elem` map isAlpha patt) $
        die "Only alphabets supported in input"

      let pattUpp = map toUpper patt
      contents <- S.readFile filename

      let text = createGridFromInput contents n
      let pattern =  (S.unpack . C8.pack) pattUpp
      let rev_pattern = (S.unpack . C8.pack) (reverse pattUpp)

      let rowmatches = map (rabinKarpMain pattern rev_pattern text m n True)
                           [0..n-1]
      let colmatches = map (rabinKarpMain pattern rev_pattern text m n False)
                           [0..n-1]
      let filteredRowMatches = concat $ filter (not . null) rowmatches
      let filteredColMatches = concat $ filter (not . null) colmatches
      if not (null filteredRowMatches) then do
        print "Row matches found at coordinates:"
        sequence_ [print w | w <- filteredRowMatches]
      else do
        print "Pattern not found across row"
      if not (null filteredColMatches) then do
        print "Column matches found at coordinates:"
        sequence_ [print w | w <- filteredColMatches]
```

```
      else do
        print "Pattern not found across column"
    _ -> die "Usage: stack exec -- word-search-seq-exe <pattern> <grid_size>
<file_containing_grid>"
```

The above code is the main program for the sequential implementation. The following steps are performed:

- Input pattern, grid size, and filepath containing input grid.
- The file contents and pattern are read as ByteStrings for efficiency.
- The reverse of the pattern is computed.
- A 2-D array of Word8 is created from the input grid using the function `createGridFromInput`.
- The matches across every row and column are computed by calling the `rabinKarpMain` function.

WordSearchPar/Main.hs

```
module Main (main) where

import Search(rabinKarpMain)
import GridReader(createGridFromInput)


import System.Environment ( getArgs )
import System.Exit ( die )
import Control.Monad ( when )
import Control.Parallel.Strategies ( rseq, parBuffer, using)
import qualified Data.ByteString as S
import Data.ByteString.Char8 as C8 (pack)
import Data.Char (toUpper,isAlpha)



main :: IO ()
main = do
  args <- getArgs
  case args of
    [patt, nVal, filename] -> do
      let n = (read nVal :: Int)
      let m = length patt
```

```
      when(m < 2) $
        die "Pattern must be of length >= 2"

      when(m > n) $
        die "Pattern length must be less than or equal to grid size"

      when (False `elem` map isAlpha patt) $
        die "Only alphabets supported in input"

      let pattUpp = map toUpper patt
      contents <- S.readFile filename

      let text = createGridFromInput contents n
      let pattern =  (S.unpack . C8.pack) pattUpp
      let rev_pattern = (S.unpack . C8.pack) (reverse pattUpp)


      let rowmatches = map (rabinKarpMain pattern rev_pattern text m n True)
                           [0..n-1] `using` parBuffer 50 rseq
      let colmatches = map (rabinKarpMain pattern rev_pattern text m n False)
                          [0..n-1] `using` parBuffer 50 rseq
      let filteredRowMatches = concat $ filter (not . null) rowmatches
      let filteredColMatches = concat $ filter (not . null) colmatches
      if not (null filteredRowMatches) then do
        print "Row matches found at coordinates:"
        sequence_ [print w | w <- filteredRowMatches]
      else do
        print "Pattern not found across row"
      if not (null filteredColMatches) then do
        print "Column matches found at coordinates:"
        sequence_ [print w | w <- filteredColMatches]
      else do
        print "Pattern not found across column"
    _ -> die "Usage: stack exec -- word-search-par-exe <pattern> <grid_size>
<file_containing_grid>"
```

The above code is the main program for the parallel implementation.

src/GridReader.hs

```
{-
  Module responsible for reading input into a 2-D array.
```

```haskell
-}
module GridReader
    ( createGridFromInput
    ) where

import qualified Data.Word8 as W
import Data.Array ( Array, array )
import qualified Data.ByteString as S
{-
  Given a bytestring and size of the grid, create
  a grid of dimensions (size X size)
-}
createGridFromInput :: S.ByteString -> Int -> Array (Int, Int) W.Word8
createGridFromInput contents size = let word_arr = map W.toUpper $ filter
                                        W.isAlpha $ S.unpack contents
                                    in array ((0,0), (size-1, size-1))
                                      ([ ((i,j), word_arr !! (size*i + j))
                                       | i  <- [0..size-1], j <- [0..size-1]])
```

The above function creates a 2D array from the input contents in `ByteString`.

src/Search.hs

```haskell
{-
  Module containing the Rabin-Karp algorithm.
-}
module Search
    ( rabinKarpMain
    ) where

import Data.Array ( (!), Array )
import qualified Data.Word8 as W

-- No of alphabets in input
d :: Int
d = 26

-- Prime number used in hash function
q :: Int
q = 13
```

```haskell
{-
  Calculate the hash value of the given word.
-}
calcHashPatt :: Int -> [W.Word8] -> Int
calcHashPatt = foldl (\ h x -> ((h * d) + fromEnum x) `mod` q)

{-
  Calculate the hash value of a string present in the grid.
  If 'isRow' is True, calculate the hash value of a string at arr(fixedIdx,f)
of size (l-f+1)
  If 'isRow' is False, calculate the hash value of a string at arr(f, fixedIdx)
of size (l-f+1)
-}
calcHash :: Int -> Int -> Int -> Array (Int, Int) W.Word8 -> Int -> Bool -> Int
calcHash fixedIdx f l text h isRow
  | f == l = h
  | isRow = calcHash fixedIdx (f+1) l text (((h * d) + fromEnum (text !
    (fixedIdx, f))) `mod` q) isRow
  | otherwise = calcHash fixedIdx (f+1) l text (((h * d) + fromEnum (text ! (f,
    fixedIdx))) `mod` q) isRow

{-
  Check if a substring present in grid matches the pattern.
  If 'isRow' is True, compare pattern with string starting at
arr(fixedIdx,baseIdx) of size (l-f+1)
  If 'isRow' is False, compare pattern with string starting at
arr(baseIdx,fixedIdx) of size (l-f+1)
  Returns a boolean.
-}
areEqual :: Int -> Int -> Int -> [W.Word8] -> Array (Int, Int) W.Word8 -> Int
-> Bool -> Bool
areEqual baseIdx f l patt text fixedIdx isRow
  | f == l = True
  | isRow = ((patt !! f) == (text ! (fixedIdx, baseIdx+f))) &&
      areEqual baseIdx (f+1) l patt text fixedIdx isRow
  | otherwise = ((patt !! f) == (text ! (baseIdx+f, fixedIdx))) &&
      areEqual baseIdx (f+1) l patt text fixedIdx isRow

{-
  Calculate the new hash value of a string in grid from existing hash by
sliding by 1 character.
-}
calcSlidingHash :: Int -> Int -> Int -> Int -> Int -> Array (Int, Int) W.Word8
-> Int -> Bool -> Int
calcSlidingHash tHash baseIdx l m h text fixedIdx isRow
```

```
  | baseIdx < l = let
              nHash = if isRow then
                        (d * (tHash - fromEnum ( text ! (fixedIdx, baseIdx) ) *
                         h) +
                         fromEnum(text ! (fixedIdx, baseIdx+m))) `mod` q
                      else
                        (d * (tHash - fromEnum ( text ! (baseIdx, fixedIdx) ) *
                         h) +
                         fromEnum(text ! (baseIdx+m, fixedIdx))) `mod` q
            in
              if nHash < 0 then
                nHash + q
              else
                nHash
  | otherwise = tHash

{-
  Update the result with matched index if the word matches with the pattern.
  Returns updated result list.
-}
findMatch :: [W.Word8] -> Array (Int, Int) W.Word8 -> Int -> Int -> Int -> Bool
-> [Int] -> Int -> [Int]
findMatch patt text i fixedIdx m isRow res matched_idx
 = let isMatch = areEqual i 0 (m-1) patt text fixedIdx isRow
       new_res = if isMatch then
                   matched_idx : res
                 else
                   res
   in new_res

{-
  The main Rabin-Karp algorithm that iterates through a row/column
  and compares the hash value of words with the pattern and it's reverse.
  It returns a list of indices of matches.
-}
rabinKarp :: Int -> Int -> Int -> Array (Int, Int) W.Word8 -> [W.Word8] ->
             [W.Word8] -> Int
            -> Int -> [Int] -> [Int] -> Int -> Int -> Int -> Bool -> [Int]
rabinKarp tHash pHash pRevHash text patt revPatt i l res revRes m h fixedIdx
isRow
    | i == l = res ++ revRes
    | tHash == pHash = let
                           new_res = findMatch patt text i fixedIdx m isRow res
                                     i
                           new_tHash = calcSlidingHash tHash i l m h text
```

```
                                        fixedIdx isRow
                        in
                           rabinKarp new_tHash pHash pRevHash text patt revPatt
                            (i+1) l new_res revRes m h fixedIdx isRow
    | tHash == pRevHash = let
                            new_res = findMatch revPatt text i fixedIdx m isRow
                                      revRes (i+m-1)
                            new_tHash = calcSlidingHash tHash i l m h text
                                        fixedIdx isRow
                           in
                             rabinKarp new_tHash pHash pRevHash text patt
                             revPatt (i+1) l res new_res m h fixedIdx isRow
    | otherwise = let new_tHash = calcSlidingHash tHash i l m h text fixedIdx
                                  isRow
                  in
                    rabinKarp new_tHash pHash pRevHash text patt revPatt (i+1)
                    l res revRes m h fixedIdx isRow

{-
  Calculates the hash of pattern, reversed pattern and first word on the
row/column.
  It then calls the main Rabin-Karp algorithm which returns matches.
  Returns a list of (x,y) coordinates of matches.
-}
rabinKarpMain :: [W.Word8] -> [W.Word8] -> Array (Int, Int) W.Word8 -> Int ->
Int -> Bool -> Int -> [(Int, Int)]
rabinKarpMain patt revPatt text m n isRow fixedIdx =
                                  let tHash = calcHash baseIdx 0 m text 0 isRow
                                      pHash = calcHashPatt 0 patt
                                      pRevHash = calcHashPatt 0 revPatt
                                      h = foldl (\acc _ -> (acc*d) `mod` q) 1 $
                                          replicate 1 (m-1)
                                      res = rabinKarp tHash pHash pRevHash text
                                patt revPatt 0 (n-m+1) [] [] m h fixedIdx isRow
                                  in
                                     if isRow then
                                        zip (repeat fixedIdx) res
                                     else
                                        zip res (repeat fixedIdx)
```

The `rabinKarpMain` function is externally called by `Main.hs` on every row and column of the grid. This function returns the coordinates of matches. It internally uses the other functions in the code (descriptions of which can be found in the comments).

Test Inputs

For testing, we have considered 3 inputs of sizes 100x100, 500x500, and 750x750. These input files contain the word “PROGRAMMING” present in all 4 directions. The files are present in the inputs/ folder.

Parallelisation

We can improve the performance of word search by parallelising the search across the rows and columns. All the performance tests have been done on the 750x750 grid.

Let’s first look at the performance on 1 core using the sequential implementation:

```
stack exec -- word-search-seq-exe PROGRAMMING 750
D:\PFP\word-search\inputs\ip750.txt +RTS -s -ls

"Row matches found at coordinates:"
(175,26)
(227,109)
(377,51)
"Column matches found at coordinates:"
(346,32)
(294,70)
(604,107)

222,853,632 bytes allocated in the heap
     101,454,552 bytes copied during GC
      27,629,712 bytes maximum residency (4 sample(s))
         821,104 bytes maximum slop
              57 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     207 colls,     0 par    0.094s   0.178s     0.0009s    0.0562s
  Gen  1       4 colls,     0 par    0.078s   0.081s     0.0203s    0.0376s

  TASKS: 3 (1 bound, 2 peak workers (2 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.001s elapsed)
```

```
  MUT     time  623.156s  (683.527s elapsed)
  GC      time    0.172s  (  0.259s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time  623.328s  (683.787s elapsed)

  Alloc rate    357,620 bytes per MUT second

  Productivity 100.0% of total user, 100.0% of total elapsed
```

Let's discuss the different parallelisation strategies below.

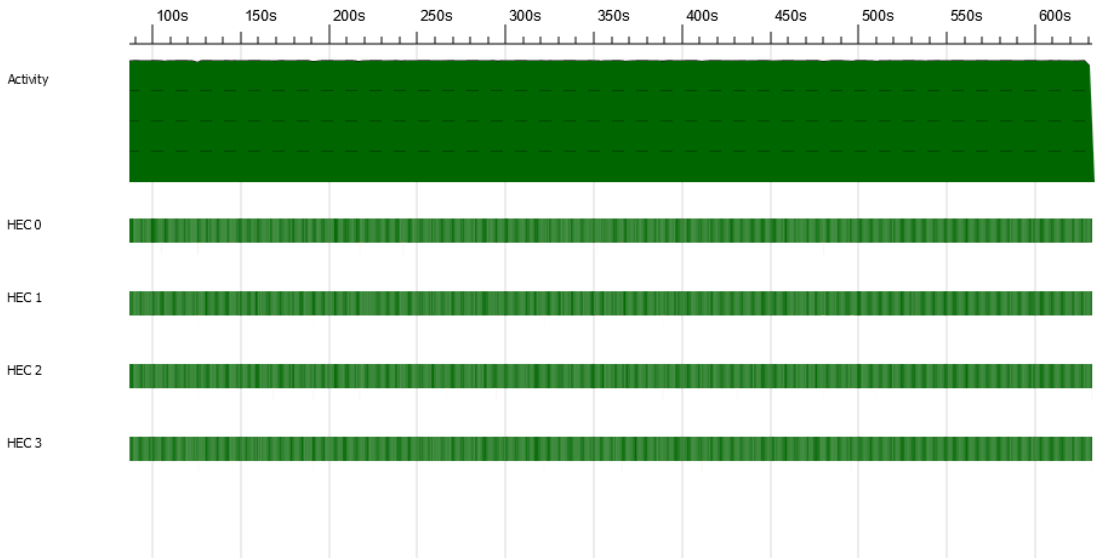
- We can parallelise search using `parList` with `rseq` the following way:

```
let rowmatches = map (rabinKarpMain pattern rev_pattern text m n True)
                          [0..n-1] `using` parList rseq
let colmatches = map (rabinKarpMain pattern rev_pattern text m n False)
                         [0..n-1] `using` parList rseq
```

Given below is the result:

```
stack exec -- word-search-par-exe PROGRAMMING 750
D:\PFP\word-search\inputs\ip750.txt 50 +RTS -s -ls -N4
"Row matches found at coordinates:"
(175,26)
(227,109)
(377,51)
"Column matches found at coordinates:"
(346,32)
(294,70)
(604,107)
     262,579,160 bytes allocated in the heap
     188,270,024 bytes copied during GC
      52,901,864 bytes maximum residency (5 sample(s))
       8,565,896 bytes maximum slop
             108 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0       123 colls,   123 par    0.516s   0.248s     0.0020s    0.0096s
Gen  1         5 colls,     4 par    0.281s   0.160s     0.0321s    0.0718s
```

```
  Parallel GC work balance: 22.20% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 1500 (1500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.002s elapsed)
  MUT     time  1738.094s  (631.572s elapsed)
  GC      time    0.797s  (  0.409s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time  1738.891s  (631.983s elapsed)

  Alloc rate    151,073 bytes per MUT second

  Productivity 100.0% of total user, 99.9% of total elapsed
```



We can see that there is not a big improvement. The total execution time is around 631 s. This could be because there are way too many sparks created at the same time and their are only 4 cores to evaluate all of them increasing the MUT time.

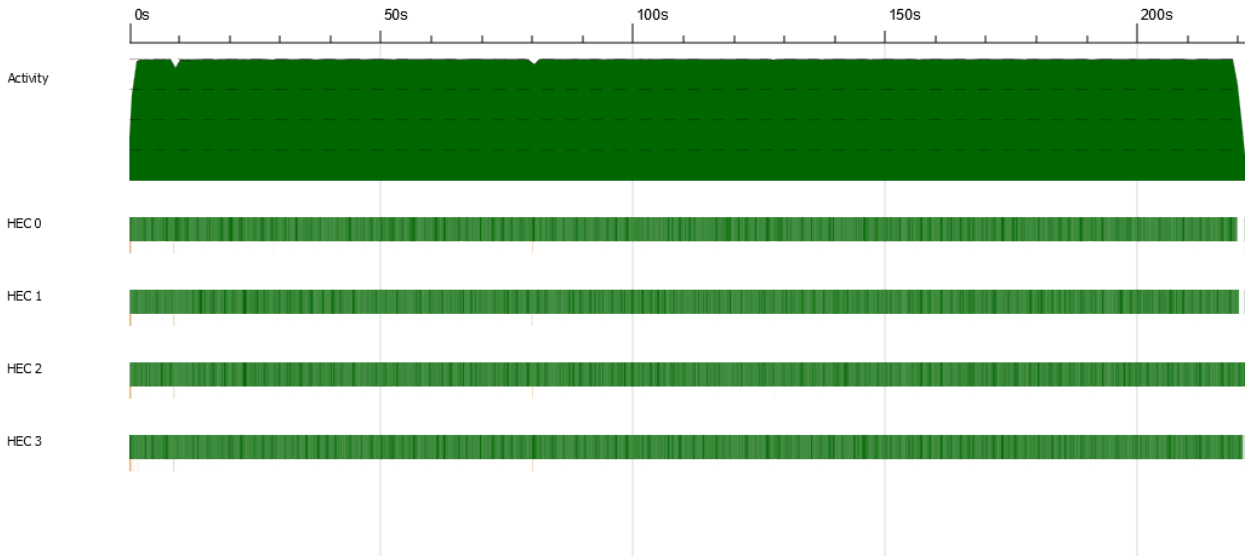
- Let's now try using `parBuffer` with `rpar`

```
let rowmatches = map (rabinKarpMain pattern rev_pattern text m n True)
                          [0..n-1] `using` parBuffer 50 rpar
let colmatches = map (rabinKarpMain pattern rev_pattern text m n False)
```

```
                        [0..n-1] `using` parBuffer 50 rpar
```

Given below are the results:

```
"Row matches found at coordinates:"
(175,26)
(227,109)
(377,51)
"Column matches found at coordinates:"
(346,32)
(294,70)
(604,107)
     261,814,824 bytes allocated in the heap
     139,329,432 bytes copied during GC
      27,798,424 bytes maximum residency (6 sample(s))
      49,523,296 bytes maximum slop
             133 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0        77 colls,    77 par    0.328s   0.141s     0.0018s    0.0095s
Gen  1         6 colls,     5 par    0.391s   0.248s     0.0414s    0.0696s

  Parallel GC work balance: 14.97% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 3193 (1541 converted, 0 overflowed, 2 dud, 52 GC'd, 1598 fizzled)

  INIT    time    0.000s  (  0.002s elapsed)
  MUT     time  788.219s  (220.825s elapsed)
  GC      time    0.719s  (  0.390s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time  788.938s  (221.217s elapsed)

  Alloc rate    332,160 bytes per MUT second

  Productivity  99.9% of total user, 99.8% of total elapsed
```



We see that the evaluation time is around 221s (speedup of 3.1) which is a huge improvement. This could be because we are limiting the number of sparks created at a time and hence reducing MUT time. But there are way too many sparks created and a lot of them were fizzled since they were already evaluated by the main program. So using `rseq` instead might be helpful, since rseq forces evaluation to WHNF

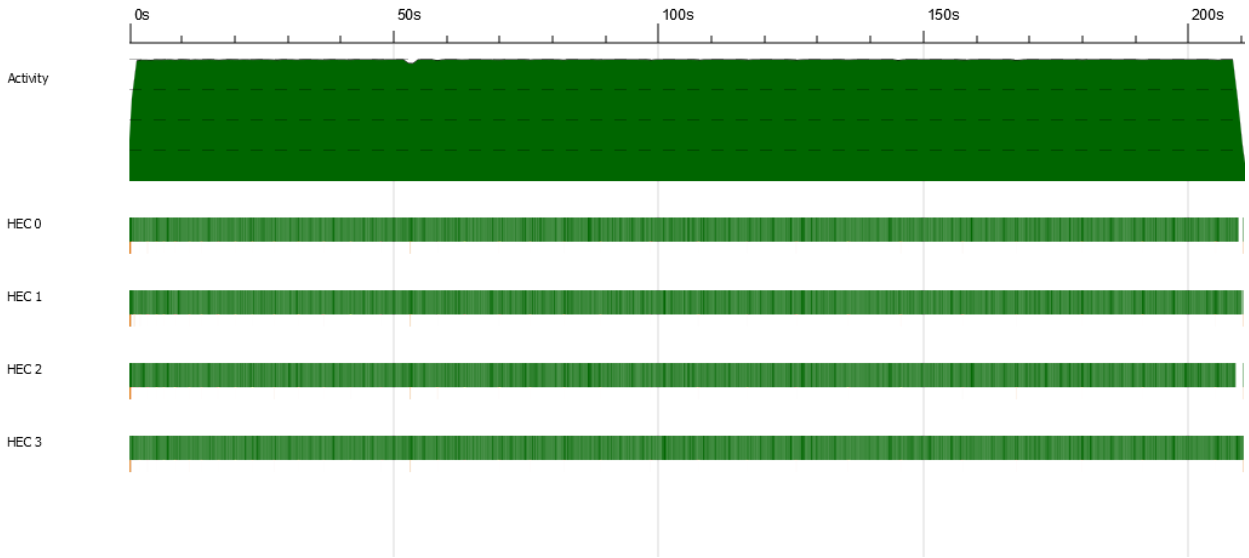
- Let's now try using `parBuffer` with `rseq`

```
let rowmatches = map (rabinKarpMain pattern rev_pattern text m n True)
                          [0..n-1] `using` parBuffer 50 rseq
let colmatches = map (rabinKarpMain pattern rev_pattern text m n False)
                         [0..n-1] `using` parBuffer 50 rseq
```

Given below are the results:

```
"Row matches found at coordinates:"
(175,26)
(227,109)
(377,51)
"Column matches found at coordinates:"
(346,32)
(294,70)
(604,107)
     250,390,472 bytes allocated in the heap
     140,407,224 bytes copied during GC
      29,107,784 bytes maximum residency (6 sample(s))
      52,999,792 bytes maximum slop
             148 MiB total memory in use (0 MB lost due to fragmentation)
```

```
                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0        85 colls,    85 par    0.266s   0.166s     0.0020s    0.0101s
Gen  1         6 colls,     5 par    0.531s   0.260s     0.0433s    0.0700s

  Parallel GC work balance: 13.83% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 1500 (1465 converted, 0 overflowed, 0 dud, 30 GC'd, 5 fizzled)

  INIT    time    0.000s  (  0.001s elapsed)
  MUT     time  780.391s  (209.905s elapsed)
  GC      time    0.797s  (  0.426s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time  781.188s  (210.332s elapsed)

  Alloc rate    320,852 bytes per MUT second

  Productivity  99.9% of total user, 99.8% of total elapsed
```



The execution time is 210s with a speedup of 3.26, this is similar to the result using `rpar`. We also notice a significant drop in the numer of fizzled and GC'd sparks. Hence this is a much better strategy than using `rpar`.

We can conclude that using `parBuffer` with `rseq` is the most effective parallelisation strategy.

Results

I have recorded the performance results on running the sequential and parallel implementation on multiple cores.

System specifications:

Processor: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz
Cores: 2
Logical Processors: 4
RAM: 8 GB

Size	**100**		**500**		**750**	
Cores	**Time (s)**	**Speedup**	**Time (s)**	**Speedup**	**Time (s)**	**Speedup**
1	0.247	1	121	1	684	1
2	0.267	0.9	79	1.53	385	1.77
3	0.134	1.8	38	3.18	337	2.03
4	0.099	2.43	37	3.27	210	3.26

We can see that the speedup is significant for large input grids. The maximum speedup is approx. **3.26** when utilising 4 cores for input sizes 500 and 750.