Project: WordLadder

Overview

The goal of the project was to show Haskell parallelism in the context of the word-ladder problem. The word-ladder challenge is defined by finding the shortest path in a graph in which individual words are vertices, while an edge is formed by an a transition function.

Vertice
Edge
Vertice

Multiple transition functions can be used. For the purpose of this project, a simple transition rule of substituting one letter has been chosen. That is, two words are considered connected if:

- They have the same length
- They differ exactly by one letter

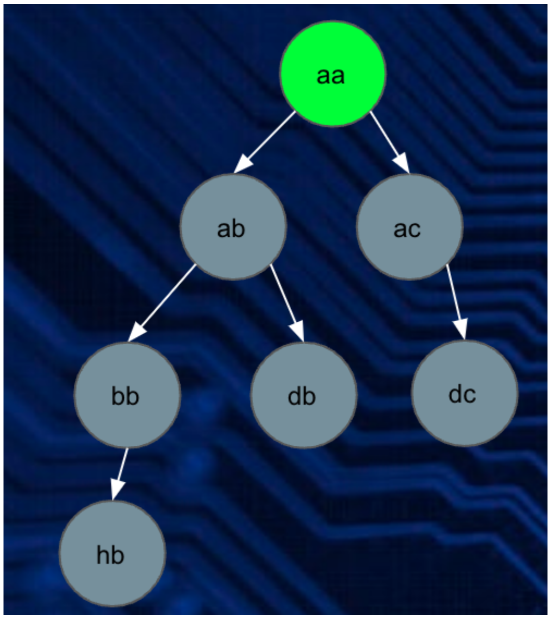
The following two graphs are examples of connected words:

Because of the nature of finding the shortest path, the Breadth-First Search algorithm is the best approach to solve the problem.

The BFS Algorithm

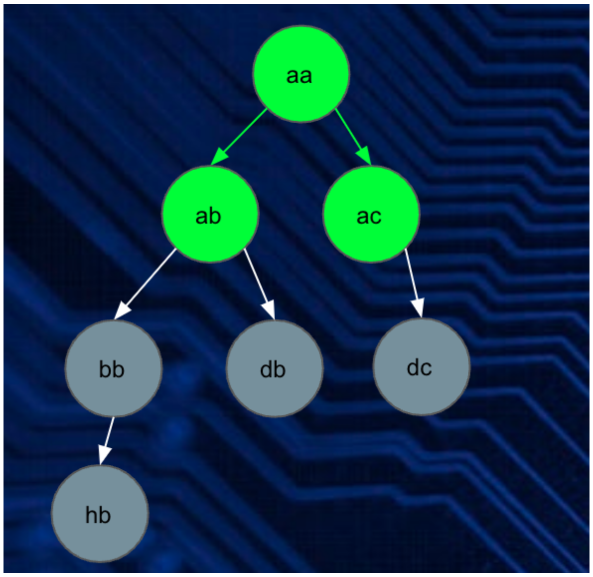
The BFS in our implementation runs as the following:

1. Assume the starting word is “aa”, and he ending word is “hb”.



- Paths: [[aa]]
- Dictionary: [ab, ac, bb, db, dc, hb, zz]

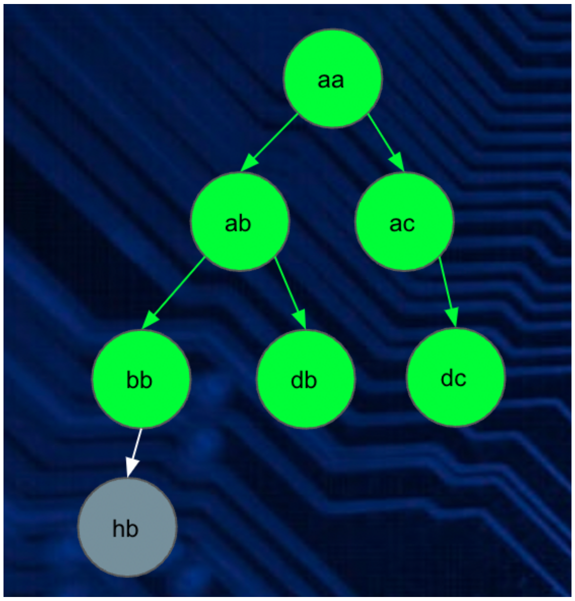
2.



- Paths: [[ab, aa], [ac, aa]]

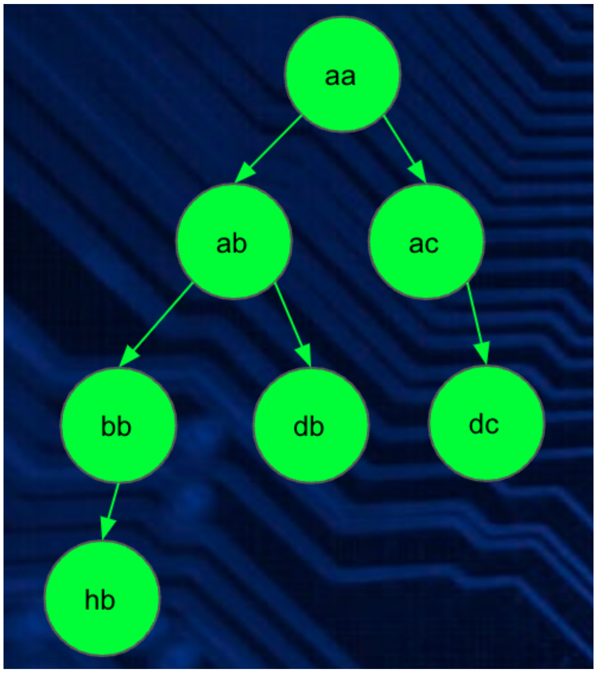
- Dictionary: [bb, db, dc, hb, zz]
- Note: the list that represent a single path (for example, [ab, aa] is reversed because it's more efficient to append a new element on the left in Haskell

3.



- Paths: [[bb, ab, aa], [db, ab, aa], [dc, ac, aa]]
- Dictionary: [hb, zz]

4.



- Paths: [[hb, bb, ab, aa], [db, ab, aa], [dc, ac, aa]]

    - Dictionary: [zz]
    - “hb” is in the paths and thus found. So the program can stop

Algorithm Implementations

The program is limited to paths with a maximum of 20 depths so that it stops searching if the target word can’t be reached after traversing 20 edges.

```
return $ process 20 [[BSU.pack fromWord]] (BSU.pack toWord) dict
…
showResults Nothing = "Unable to find a ladder in 20"
```

Because of the transition function choice, the program only finds paths between words of equal length. Words in the dictionary file that have different length than stating and target words are disregarded.

```
allLowerAlphaLength :: Int -> ByteString -> Bool
allLowerAlphaLength desiredLength word | (BS.length word) /= desiredLength = False
allLowerAlphaLength _ word = BSU.all (\c -> and [(isAlpha c), (isLower c)]) word
```

The algorithmic implementations are as follows

1. Read the content from the .txt file; filter out the dictionary to only keep words of the appropriate length, lower letter, and alphabetic words

   ```
   contents <- readFile filename

   ...

   createDict content desiredLength = filter (allLowerAlphaLength desiredLength) $
   BSU.words content
   . . .
   allLowerAlphaLength desiredLength word | (BS.length word) /= desiredLength = False
   ```

   where
   - content is the unsplitted .txt file content
   - “desiredLength” is a variable that specifies the length of toWord and fromWord
   - BSU.words splits bytestring into words

2. Construct a set of paths thus traveled (green vertices and edges in the last section); initialize that set with just one path containing just the starting word.

   ```
   process 20 [[BSU.pack fromWord]] (BSU.pack toWord) dict
   ```

   where

- the process function tries to perform the BFS algorithm. It starts from the fromWords and tries to reach the toWord. It uses oneCharDifference function to find the next vertices (nodes) for search
- `[[BSU.pack fromWord]]` creates a list of list with the initial starting word
- BSU.pack converts string type into bytestring type
- “dict” is a set of splitted bytestrings from the .txt words.

3. For every path in the queue, take the first word in the path and find all words connected to it

```
next = findNextWords (head p) d
...
findNextWords currentWord dict = Set.filter suitable dict
  where
    suitable word = oneCharDifference word currentWord
```

where

- The “suitable” function returns True/False to tell whether provided words is suitable as the next word in the path
- "word" is a single word that's only one char different from current word

4. Append the newly found words to each of the paths; if a path has multiple words to go, it branches out to all of them

```
eval d p =
  let
    next = findNextWords (head p) d
  in
    (map (\w -> w:p) (Set.toList next), next)
```

where

- `(\w -> w:p)` conducts the appending of newly found words

5. Remove the newly found words from the dictionary.

```
›
dict' = Set.difference dict (Set.unions nexts)
```

where

- “dict” is a set and “nexts” is a list of list of new words.
- “Set.unions” converts “nexts” into a set
- “Set.difference” removes “nexts” set from “dict”

6. If any of the new words have been the target word, the search ends; otherwise, go back to 3) and repeat until either the word is reached or the program hits the maximum limit (20 depths).

```
let
```

```
 . . .
  found = filter (\p -> head p == to) paths'
in
  case found of
    (p:_) -> Just $ reverse p
    [] -> process (limit-1) paths' to dict'
```

where

- “reverse” is for better presenting the answer
- paths’ is the updated paths (updated queue) that we just branched out
- dict’ is the updated set of not visited vertices words

Sequential implementation

The sequential implementation has a couple of differences from a naive approach implemented in homework 4, which used built-in data structures like String and Lists. In homework 4, the following code detects if the words differ by exactly one letter, forming the word transition function:

```
oneCharDifference :: String -> String -> Bool
oneCharDifference a b | (length a) /= (length b) = False
oneCharDifference a b = (length $ filter (\(a1,b1) -> a1 /= b1) $ zip a b) == 1
```

In the final project code, I made several derivations from the initial code in homework 4.

First, it uses ByteStrings to represent the words. That allows the use of the efficient **packZipWith** function operating strictly on the byte data, allowing for faster edge finding than usual strings because ByteString has a contiguous memory buffer without splits in the middle, making iterations faster. It also uses **count** from the ByteString library – for the same reason, ByteString is faster than usual strings with the “count". The following code forms the basis of the edge detection algorithm, where the majority of the CPU time is being spent in the sequential implementation

```
oneCharDifference :: ByteString -> ByteString -> Bool
oneCharDifference a b | (length a) /= (length b) = False
oneCharDifference a b = BSU.count '1' diffs == 1
  where
    diffs = (BSU.packZipWith (\ca cb -> if ca /= cb then '1' else '0') a b)
```

Second, this final project code uses a **Set** to represent the dictionary, allowing the removal of elements contained in a set in **log n** time because “Set” is implemented with a balanced binary tree structure

```
dict' = Set.difference dict (Set.unions nexts)
```

Parallel implementation

In order to show the speedup achieved by parallelism, the parallel version uses almost the same implementation as the sequential implementation, except for the following parallelization. The crucial operation done in parallel is evaluating the paths showing in the following code: each path spawns a new spark (concurrent computation running in parallel) that can land on a different runtime thread.

```
results = parMap rpar (eval dict) paths
paths' = concat . parMap rpar fst $ results
nexts = parMap rpar snd results
```

It’s worth noting in this parallel implementation that an observed shortcoming was that the paths couldn’t easily share the dictionary; in the sequential implementation, one optimization change that was tried was pruning the dictionary after every path. This meant that no word was added twice. But, in the case of the parallel approach, this would require a synchronized shared variable, and as such it wasn’t used. Instead, all new words from a given round are gathered together and removed from the dictionary at once.

Performance results

In order to more easily time the program, a testing suite has been developed. The program was inspected mainly with three methods:

1. The POSIX **time** command for wall-clock timing
2. Haskell's built-in profiling suite
3. Haskell **eventlog** and the associated tool **threadscope** for visualizing parallelism.

Because even the existence of profiling data showed significant slowdowns, the program was profiled with the profile information, but recompiled without it for performance timings (the timing below does not show profiling). The results were gathered on a system with an Intel Core i7 CPU with 4 hardware processors and 8 virtual cores. The RTS in the parallel implementation was set to 8, 4, and 2, respectively.

The exact speed may vary slightly depending on the system load, but the following results have clearly confirmed the faster overall speed of the parallel implementation (although N2 is slower than sequential because of overhead introduced by the parallelism's synchronization). In the POSIX results, the N8 parallel implementation is 2.3 times faster than the sequential implementation (0m0.757s vs. 0m1.757s). In the Threadscope results, the N8 parallel implementation is 2.7 times faster than the sequential implementation (586ms vs 1.603s).

An example set of results

Parallel -N8

```
real	0m0.757s
user	0m0.000s
sys	0m0.046s
```

Parallel -N4

```
real	0m0.984s
user	0m0.000s
sys	0m0.076s
```

Parallel -N2

```
real	0m2.209s
user	0m0.015s
sys	0m0.030s
```

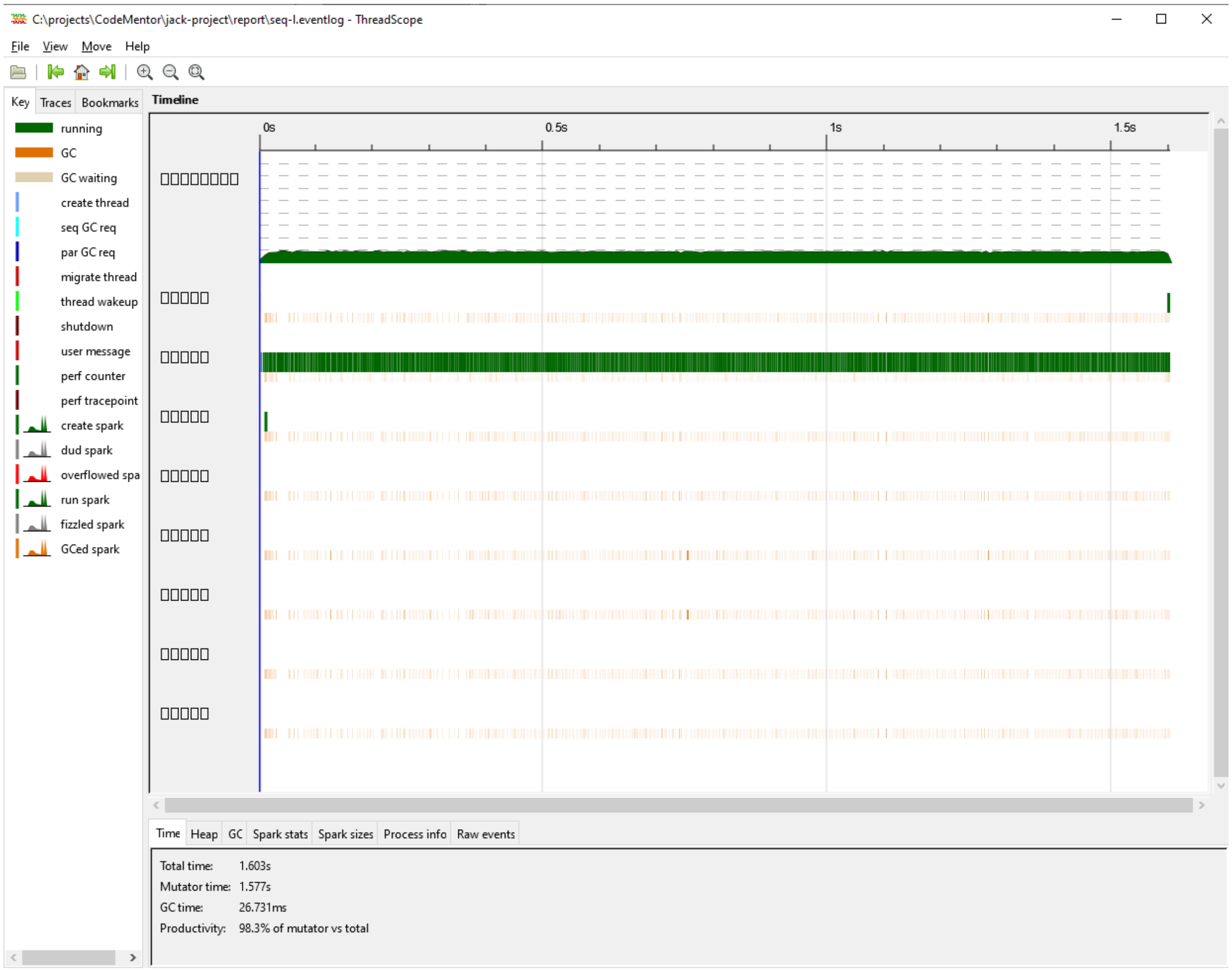
Sequential

```
real	0m1.757s
```

```
user    0m0.030s
sys     0m0.030s
```

Threadscope analysis

Sequential

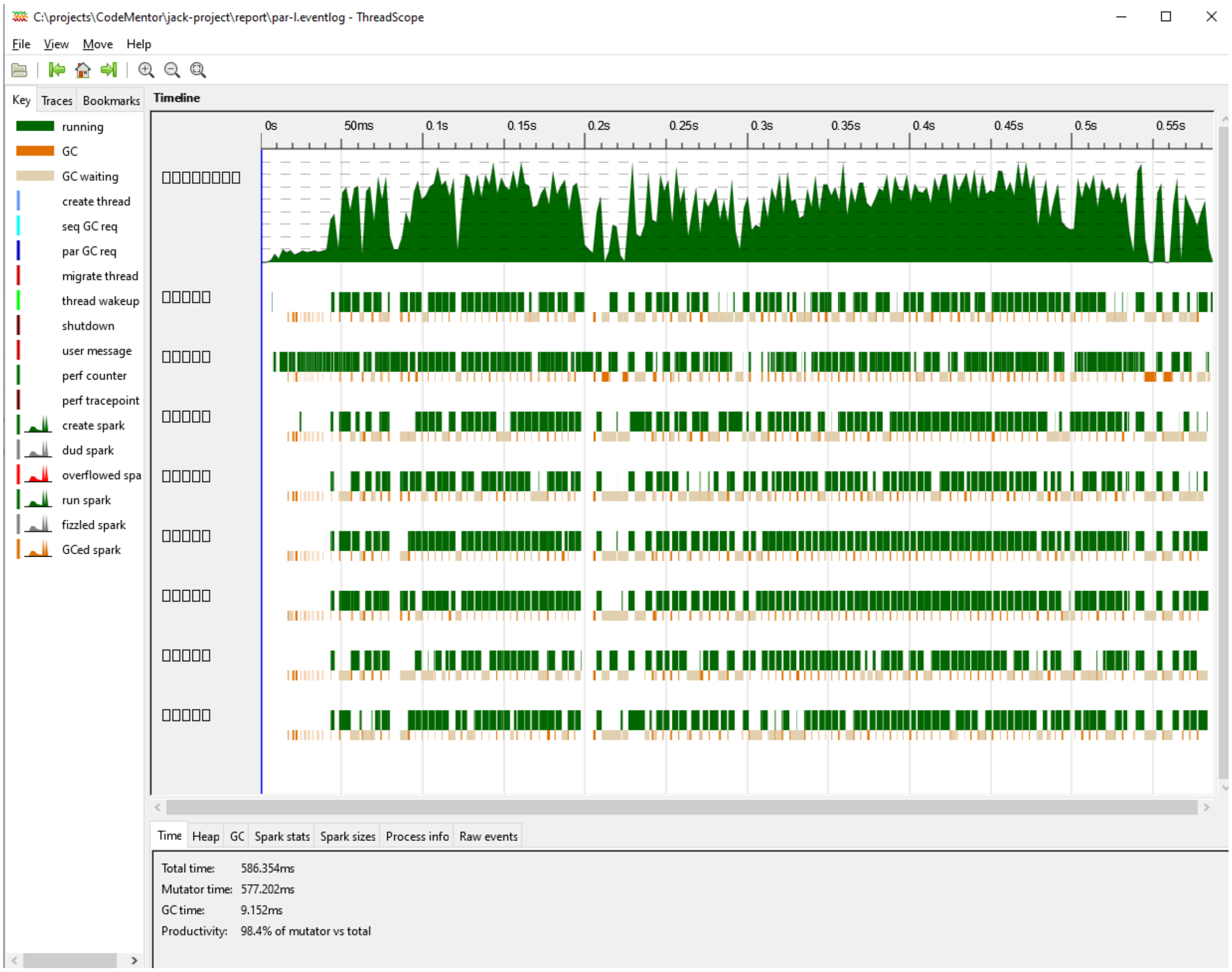


C:\projects\CodeMentor\jack-project\report\seq-l.eventlog - ThreadScope

Total time: 1.603s
Mutator time: 1.577s
GC time: 26.731ms
Productivity: 98.3% of mutator vs total

Parallel

With **+RTS -N8**



Total time: 586.354ms
Mutator time: 577.202ms
GC time: 9.152ms
Productivity: 98.4% of mutator vs total

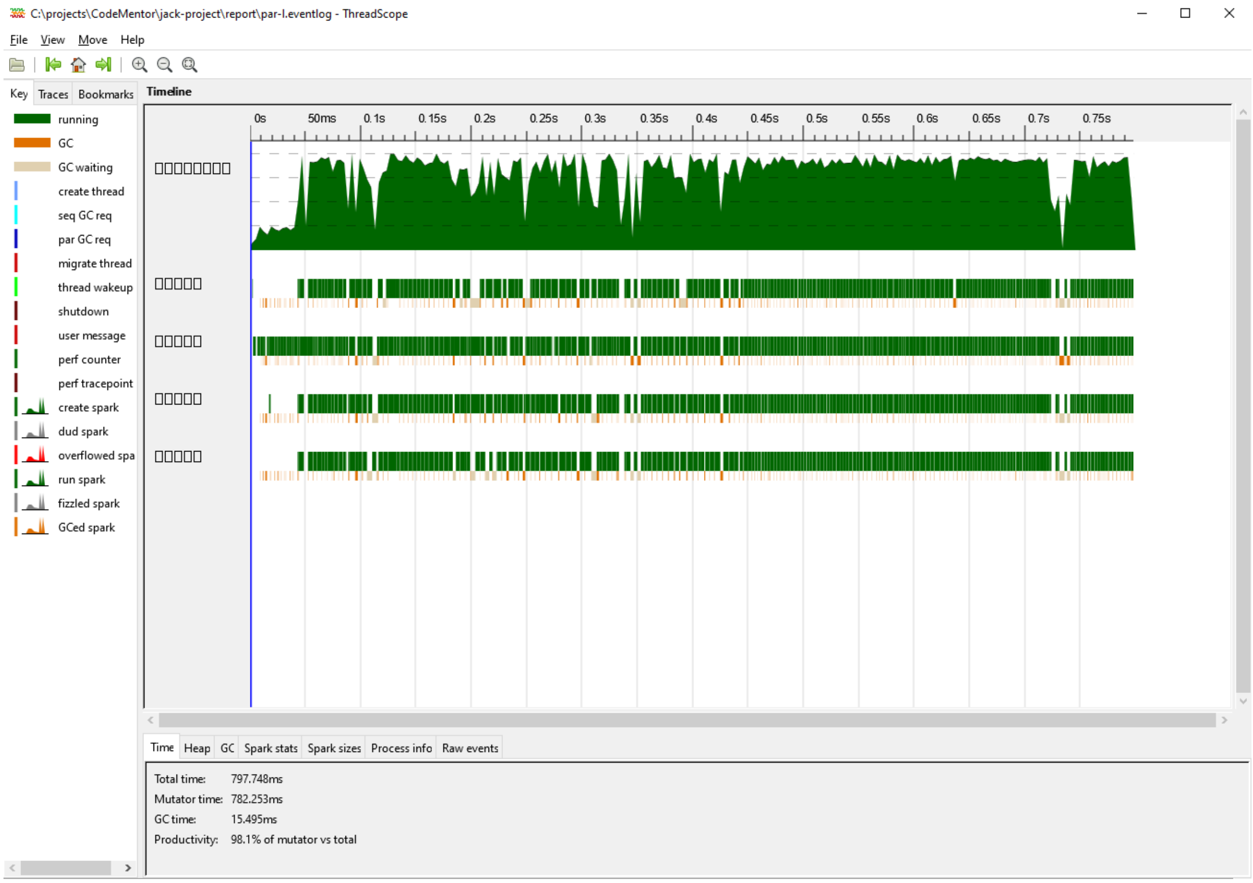
There are several possibilities of why the CPU seems to be wasting more time in the N8 implementation than in N4, N2, or sequential implementation.

- Synchronization of all threads.
- Communication between threads creates potential wait time where the CPU is idle

A possible way to decrease it is to minimize thread synchronization by:

- Splitting the problem to be better for parallelization
- Divide the data into equal sizes so that the threads don’t have to wait for each other to finish

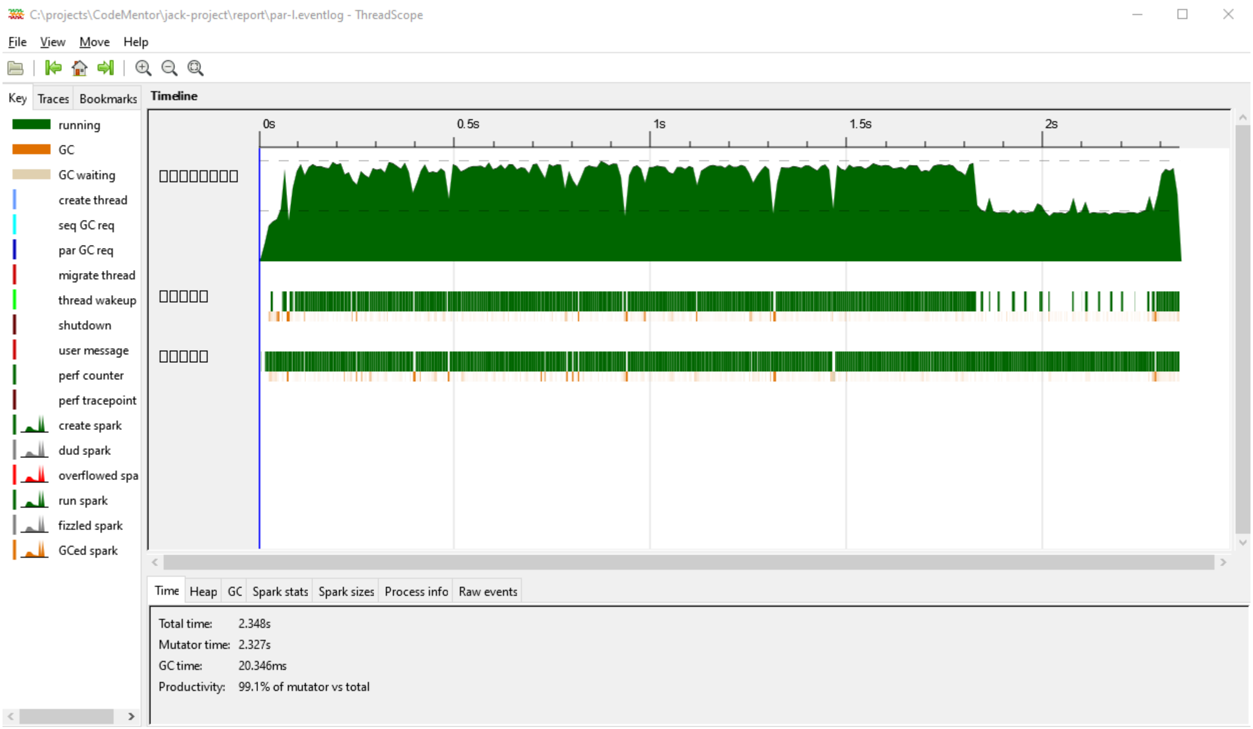
With **+RTS -N4**



C:\projects\CodeMentor\jack-project\report\par-l.eventlog - ThreadScope

```
Total time:     797.748ms
Mutator time:   782.253ms
GC time:        15.495ms
Productivity:   98.1% of mutator vs total
```

With **+RTS -N2**



```
Total time:     2.348s
Mutator time:   2.327s
GC time:        20.346ms
Productivity:   99.1% of mutator vs total
```

The best results were obtained with the parallelism of N=8, making maximum use of the CPUs virtual cores, and the speed is **2.7** times after than the sequential implementation.

Running and Building

Building

The program was built using **stack**, with **LTS 20.4** resolver and **9.25** GHC version.
The dependencies are **containers**, **parallel**, and **bytestring** (in version 0.11.1 or later).

https://www.stackage.org/lts-20.4

Threadscope

For installation instructions, see:
https://github.com/haskell/ThreadScope

Running

The necessary prerequisites are

- A POSIX-like operating system with **bash**, OR
- Windows with **MSYS2** installed
- Haskell Stack

After downloading the archive, simply run
$ ./run.sh full

To run all the tests. The reports and files will be generated in a folder called **report**.

Conclusion

N2 is slower than sequential because of the overhead introduced by parallelism

But N8 is 2.3 times faster (according to POSIX) and 2.7 times faster (according to threadscope) than sequential.

Thus, we concludes the success of the parallel implementation in the word ladder problem

Source code

Main.hs

```haskell
{-

WordLadder solver.

Shows the performance difference between parallel and sequential execution.
See run.sh for performance tests.

Usage: wordLadder <[par|seq]> <dictionary-filename> <from-word> <to-word>
$ ./wordLadder par words.txt bar none

-}

{-# LANGUAGE OverloadedStrings #-}

import qualified Prelude
import Prelude hiding (words, putStr, length, readFile)

import System.Environment
import Data.Char
import qualified Data.Set as Set

import Control.Parallel.Strategies

import qualified Data.ByteString as BS
import qualified Data.ByteString.Char8 as BSU
import Data.ByteString hiding (map, filter, head, concat, reverse, all)

noArgErrorMessage :: String
noArgErrorMessage = "Usage: wordLadder <dictionary-filename> <from-word> <to-word>"

createDict :: ByteString -> Int -> [ByteString]
createDict content desiredLength = filter (allLowerAlphaLength desiredLength) $ BSU.words
content

allLowerAlphaLength :: Int -> ByteString -> Bool
allLowerAlphaLength desiredLength word | (BS.length word) /= desiredLength = False
allLowerAlphaLength _ word = BSU.all (\c -> and [(isAlpha c), (isLower c)]) word

type Words = Set.Set ByteString

-- |
-- The algorithm:
--  Take all the paths
--  For every path p:
```

```haskell
--    Find all hops from the last element of that path
--    Multiply that path by all the hops
--    Example:
--      path is ["ac", "ab", "aa"], hops from "ac" are "bc" and "cc",
--      we end up with:
--      [
--        ["bc", "ac", "ab", "aa"],
--        ["cc", "ac", "ab", "aa"],
--      ]
--      for that path.
--      Also remove every hop from the dictionary after it's found.
--  Gather all paths (concat) into a new set
--  If any of the paths ends with `to`, end the algorithm.
--  Otherwise, repeat.
process :: Int -> [[ByteString]] -> ByteString -> Words -> Maybe [ByteString]
process 0 _ _ _ = Nothing
process limit paths to dict =
  if Set.null dict
    then Nothing
    else
      let
        results = map (eval dict) paths
        paths' = concat . map fst $ results
        nexts = map snd results
        dict' = Set.difference dict (Set.unions nexts)

        found = filter (\p -> head p == to) paths'
      in
        case found of
          (p:_) -> Just $ reverse p
          [] -> process (limit-1) paths' to dict'
  where
      eval d p =
        let
          next = findNextWords (head p) d
        in
          (map (\w -> w:p) (Set.toList next), next)

processPar :: Int -> [[ByteString]] -> ByteString -> Words -> Maybe [ByteString]
processPar 0 _ _ _ = Nothing
processPar limit paths to dict =
  if Set.null dict
    then Nothing
    else
      let
        results = parMap rpar (eval dict) paths
        paths' = concat . parMap rpar fst $ results
        nexts = parMap rpar snd results
        dict' = Set.difference dict (Set.unions nexts)
        found = filter (\p -> head p == to) paths'
      in
        case found of
          (p:_) -> Just $ Prelude.reverse p
          [] -> processPar (limit-1) paths' to dict'
  where
      eval d p =
        let
          next = findNextWords (head p) d
        in
          (map (\w -> w:p) (Set.toList next), next)

findNextWords :: ByteString -> Words -> Words
findNextWords currentWord dict = Set.filter suitable dict
  where
    suitable word = oneCharDifference word currentWord

oneCharDifference :: ByteString -> ByteString -> Bool
oneCharDifference a b | (length a) /= (length b) = False
oneCharDifference a b = BSU.count '1' diffs == 1
  where
    diffs = (BSU.packZipWith (\ca cb -> if ca /= cb then '1' else '0') a b)

showResults :: Maybe [ByteString] -> ByteString
showResults Nothing = "Unable to find a ladder in 20"
showResults (Just path) = BSU.unlines path

main :: IO ()
main = do
  args <- getArgs
  case args of
    [_, _, fromWord, toWord] | (Prelude.length fromWord) /= (Prelude.length toWord) ->
      do
        Prelude.putStrLn ("🟩" ++ fromWord ++ "🟩 and 🟩"++toWord++"🟩 must be the same
length")
    [parflag, filename, fromWord, toWord] ->
      do
         contents <- readFile filename
         let dict = Set.fromList $ createDict contents (Prelude.length fromWord)
         results <- case parflag of
                      "seq" -> do
                        Prelude.putStrLn "Sequential mode"
                        return $ process 20 [[BSU.pack fromWord]] (BSU.pack toWord) dict
                      "par" -> do
                        Prelude.putStrLn "Parallel mode"
                        return $ processPar 20 [[BSU.pack fromWord]] (BSU.pack toWord)
dict
                      _ -> error "Set the par or seq flag"
         BSU.putStr (showResults results)
    _ -> Prelude.putStrLn noArgErrorMessage
```

Run.sh

```
# +RTS - runtime options follow that
# -N8 - number of threads (8)
# -l - enable eventlog for threadscope
# -p - enable profiling (Slow!!)

EXECUTABLE=./.stack-work/dist/8a54c84f/build/word-ladder-exe/word-ladder-exe.exe

function run () {
       ${EXECUTABLE} $1 words.txt sage fool +RTS -N2 $2
       NAME=$1$2

       [ -f word-ladder-exe.eventlog ] && mv word-ladder-exe.eventlog
report/$NAME.eventlog
       [ -f word-ladder-exe.prof ] && mv word-ladder-exe.prof report/$NAME.prof
}

if [ $1 = "full" ]
then
       echo "Running full set of runs"

       echo "Clearing out old results"
       rm -rf report
       mkdir -p report

       echo "Building without profile information for accurate timing"
       (stack build) > build.log 2>&1

       echo "Par timing"
       (time run par "-l") >> report/report.txt 2>&1
       echo "Seq timing"
       (time run seq "-l") >> report/report.txt 2>&1

       echo "Building with profile information"
       stack clean
       (stack build --profile) >> report/build.log 2>&1

       echo "Gathering Par profile"
       run par "-p" > /dev/null
       echo "Gathering Seq profile"
       run seq "-p" > /dev/null
else
       echo "Running run on the current build with mode $1 with flag $2"
```

```
        run $1 $2
fi
```