PFP Fall 2021

Parallel Functional Programming Final Project Report: Parallel Word Ladder

Yiqu Liu(uni: yl4617)
Daisy Wang(uni: yw3753)

1. Background

Word Ladder, also known as Doublets, or word-links, is a well-studied word game problem invented by Lewis Carroll. The target of this problem is to find the shortest transformation sequence from one word to a target word based on a given set of words, in which any two adjacent words differ by one character and each word must be a proper English word.

2. Our Approach

The traditional word ladder problem focuses on changing one character in each step and not modifying the length of each word. In our project, we extend the scope of this problem a bit by defining, adding or removing one character in each step as a valid transformation.

We implemented a parallel version of the Breadth-first searching algorithm to find the shortest path from the given word to the target word. We parallelized the generation of each layer. Our haskell implementation led to 1.9x speed up than the original sequential version when running on a 8-core machine with 8 threads assigned to it.

3. Implementation Architecture

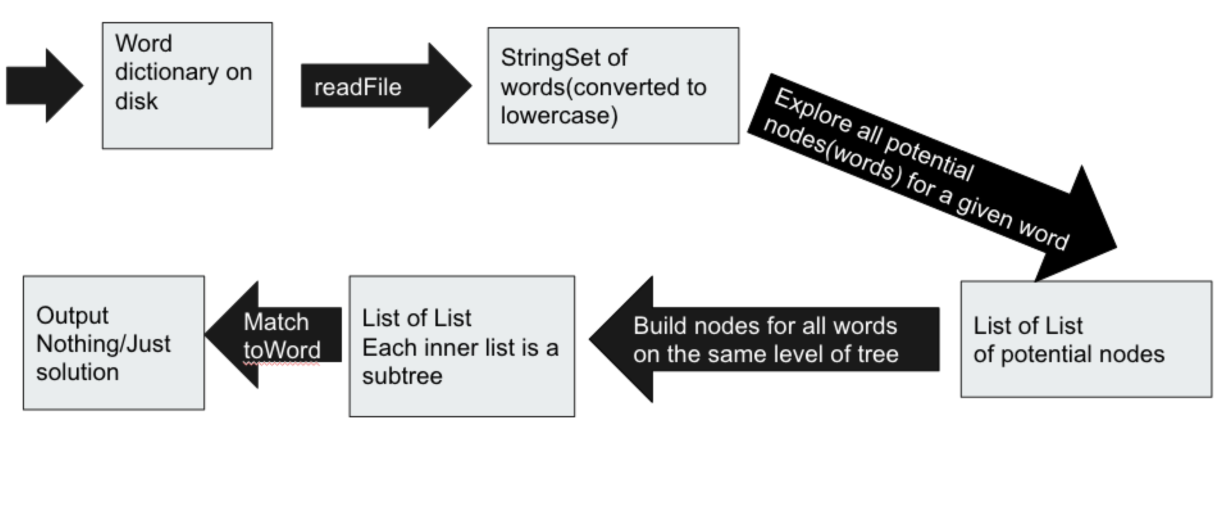
3.1 Sequential

We implemented a Breadth-first search to find the shortest path. This algorithm begins at the root node of the tree and then visits all nodes level by level. BFS uses a queue to keep the status of its searching. It takes the head of the queue, looking for all the possible children of this node. Once a node is processed, all the children will be added to the end of the queue.

The sequential version of our word ladder works as follow:

1. Read the word.txt file into memory.
2. Cast all the words into lower cases and generate a StringSet based on the result.
3. Set up a queue and put the root node in the queue.
4. Take the first element in the queue
5. Check if this node is the target node. If so, return the result.
6. Check if this node has been visited before. If so, move to the next node.
7. Generate all the possible children of it, and insert those elements into the end of the queue.
8. Go back to step 4 until the queue is empty.

The graph below shows the major components of our implementation.

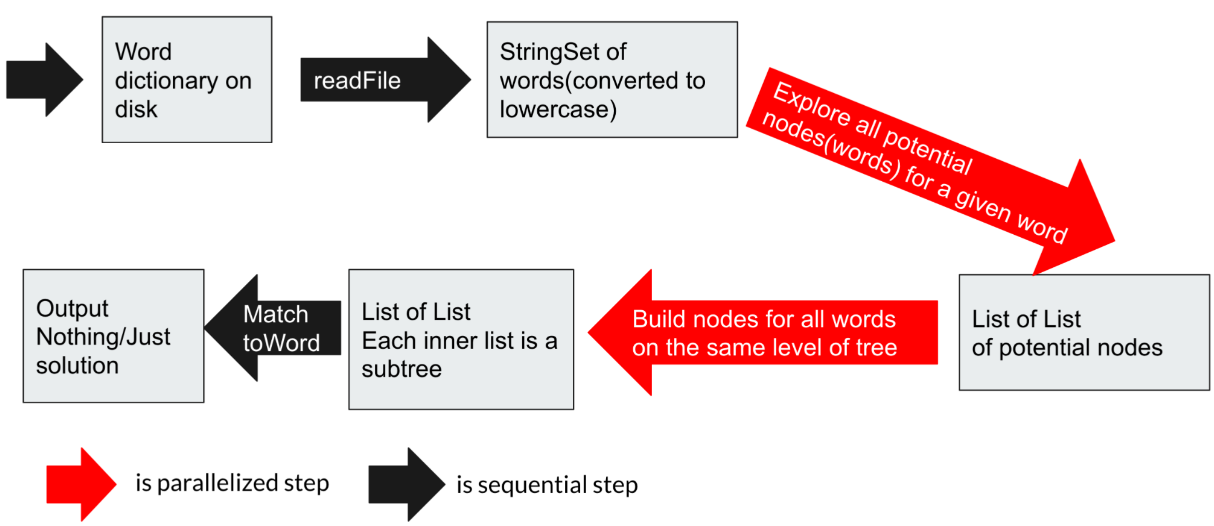


3.2 Parallel

Based on our sequential implementation, we optimized our program by applying parallelism to two key transitions. The parallel implementation of our program is shown in the graph below. The arrow in black indicates a sequential step and the arrow in red indicates parallel step.

In the second step, based on the current level of the tree, we are trying to explore the next level by building a list of words with all combinations of the alphabet. In this step, parallelism is implemented using *rpar* & *rseq.*

For each node in the current level, the previous exploration is applied. Parallelism in this step is realized by using *parMap.* In the best case, it should compute the alphabet combination for each node in parallel.



4. Experimental Design

We expect to experiment with parallelizing, exploring different strategies to receive benefits in performance, we will be using thread scope for runtime analyzing, and considering the balance between empirical results and the number of cores.

4.1 Parallelization

In our project, in order to parallel the BFS searching, some changes need to be made to the original version. Instead of maintaining a queue, we maintain a list, which only stores all the nodes in the current level. We process level by level, which means it will not move to the next level until the result of every node in this level is generated. Our final solution composed of the following steps:

1. Read the word.txt file into memory.
2. Cast all the words into lower cases and generate a StringSet based on the result.
3. Set an empty list and put the root node into it.
4. Build nodes for words on the same level of the tree
5. Explore all potential nodes for a given node and check if it is a valid word
6. Check if the word is the same as the target word. If so, return the result.
7. Go back to step 3

Obviously, step2, step 4 and 5 could be parallized. Experiments are conducted by parallelizing these three parts.

4.2 Experiment

1. Monitor the effect of parallelism on the tree node exploring
    a. Monitor the effect of using rpar and rseq
    b. Monitor the effect of using parMap
2. Monitor the effect of parallelism on the construction of an entire level in a tree
3. Find the effect of parallelism with change of word dictionary size
4. Find the effect of parallelism with the change of maximum depth of tree

4.3 Experiment Preparation

In order to see the performance of parallelization, we have to enlarge the searching scope.

1. Large word dictionary. This dictionary includes more than 290,000 words. It is downloaded from a public Github repository: https://github.com/dwyl/english-words
2. Start word and end word: If the word ladder can be found within a few steps, it's hard to see the advantages of the parallel version over the sequential one. In order to avoid this situation, we chose two words between which there's no valid word ladder: *gimlets* and *affinage*

5. Performance

5.1 Sequential Performance

The maximum searching depth is set to 50. The sequential version of word ladder uses 14.274 seconds when searching for the ladder between *gimlets* and *affinage*.

```
no ladder
   7,459,860,744 bytes allocated in the heap
     538,069,752 bytes copied during GC
      82,794,456 bytes maximum residency (9 sample(s))
         210,984 bytes maximum slop
             163 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      7056 colls,     0 par    0.606s   0.647s     0.0001s    0.0017s
  Gen  1         9 colls,     0 par    0.189s   0.270s     0.0300s    0.1234s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

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

  INIT    time    0.001s  (  0.007s elapsed)
  MUT     time   13.175s  ( 13.339s elapsed)
  GC      time    0.795s  (  0.918s elapsed)
  EXIT    time    0.000s  (  0.010s elapsed)
  Total   time   13.970s  ( 14.274s elapsed)

  Alloc rate    566,214,972 bytes per MUT second

  Productivity  94.3% of total user, 93.5% of total elapsed
```

5.2 Parallel Performance

5.2.1 Parallel the word generating part

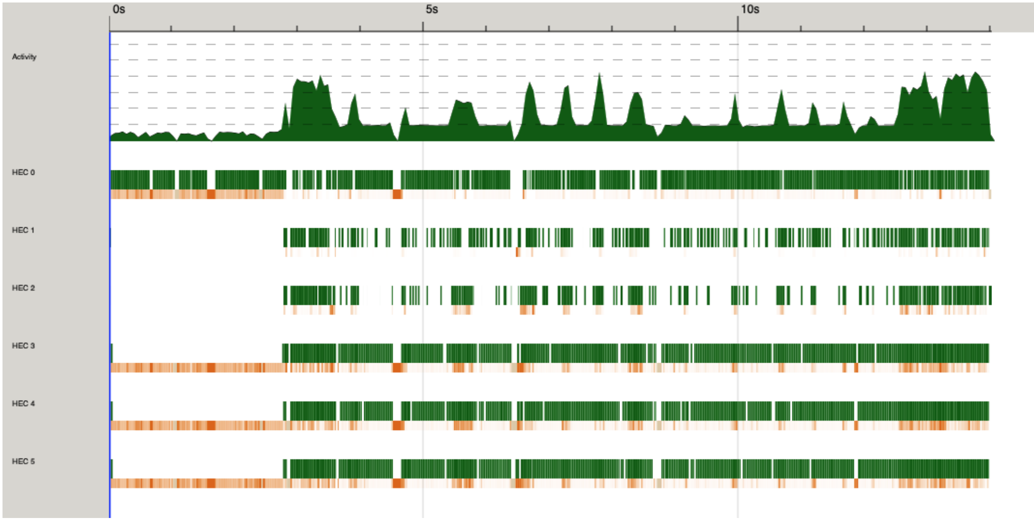
Given a word a, the possible next step would be

1. Insert one character: a[0]+c+a[1..], a[..1]+c+a[2..], …
2. Change one character: a[0]+c+a[2..], a[..1]+c+a[2..]...
3. Delete one character: a[0]+a[2..], a[..1]+c+a[3..]

These three choices of transformations could be generated at the same time. Checking if the word is valid also moved in this stage. We use Par Monad to synchronize this part. After changing this part, the performance of the search is improved.

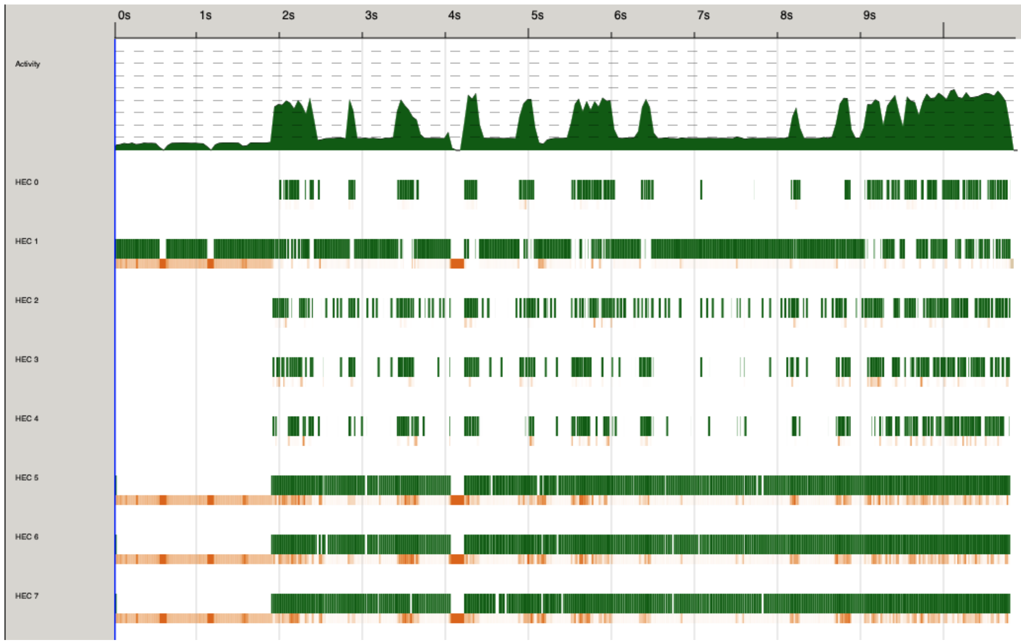
We ran the program using *gimlets* and *affinage*, with the maximum searching depth set to 50, on an 8-core machine. The performance on a 6 threads is as followed:

```
 5,557,280,848 bytes allocated in the heap
 1,011,806,232 bytes copied during GC
   139,776,816 bytes maximum residency (10 sample(s))
       404,688 bytes maximum slop
           296 MiB total memory in use (0 MB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
Gen  0      3899 colls,  3899 par    4.305s   1.138s     0.0003s    0.0081s
Gen  1        10 colls,     9 par    0.850s   0.344s     0.0344s    0.1396s

Parallel GC work balance: 24.73% (serial 0%, perfect 100%)

TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

SPARKS: 922961 (161789 converted, 45447 overflowed, 0 dud, 293611 GC'd, 307426 fizzled)

INIT    time    0.001s  (  0.008s elapsed)
MUT     time   15.866s  (  9.504s elapsed)
GC      time    5.155s  (  1.482s elapsed)
EXIT    time    0.000s  (  0.010s elapsed)
Total   time   21.022s  ( 11.004s elapsed)

Alloc rate    350,257,549 bytes per MUT second

Productivity  75.5% of total user, 86.4% of total elapsed
```



The performance on 8 threads is as followed:

```
no ladder
   5,884,125,592 bytes allocated in the heap
   1,252,260,624 bytes copied during GC
     140,330,080 bytes maximum residency (10 sample(s))
         420,768 bytes maximum slop
             302 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3767 colls,  3767 par    5.402s   1.193s     0.0003s    0.0137s
  Gen  1        10 colls,     9 par    0.782s   0.413s     0.0413s    0.1650s

  Parallel GC work balance: 20.12% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

  SPARKS: 922965 (244335 converted, 53648 overflowed, 0 dud, 242409 GC'd, 300653 fizzled)

  INIT    time    0.001s  (  0.007s elapsed)
  MUT     time   16.460s  (  9.221s elapsed)
  GC      time    6.184s  (  1.607s elapsed)
  EXIT    time    0.000s  (  0.012s elapsed)
  Total   time   22.645s  ( 10.847s elapsed)

  Alloc rate    357,483,136 bytes per MUT second

  Productivity  72.7% of total user, 85.0% of total elapsed
```



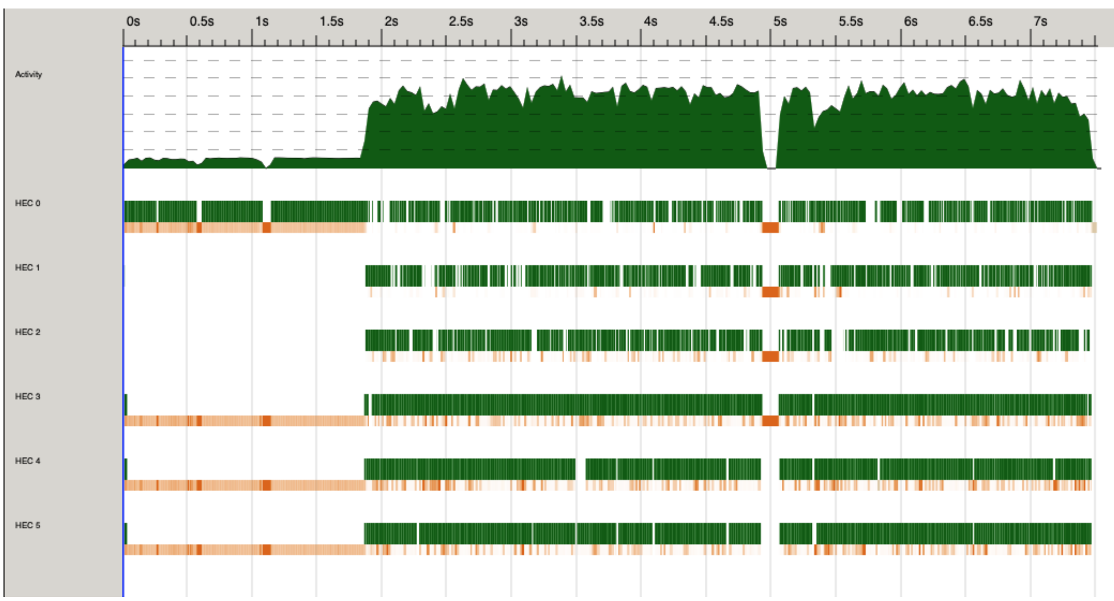
Compared to the original version, the parallelized one was sped up 0.76 times. There's still space to improve.

5.2.2 Parallel the children generating part

Every node on the same level has to generate its possible one-hop words, which can be done simultaneously. We use par Monad to optimize this part by creating sparks for each node and concat the results after every result is generated.
The performance is dramatically improved by implementing this parallelization.

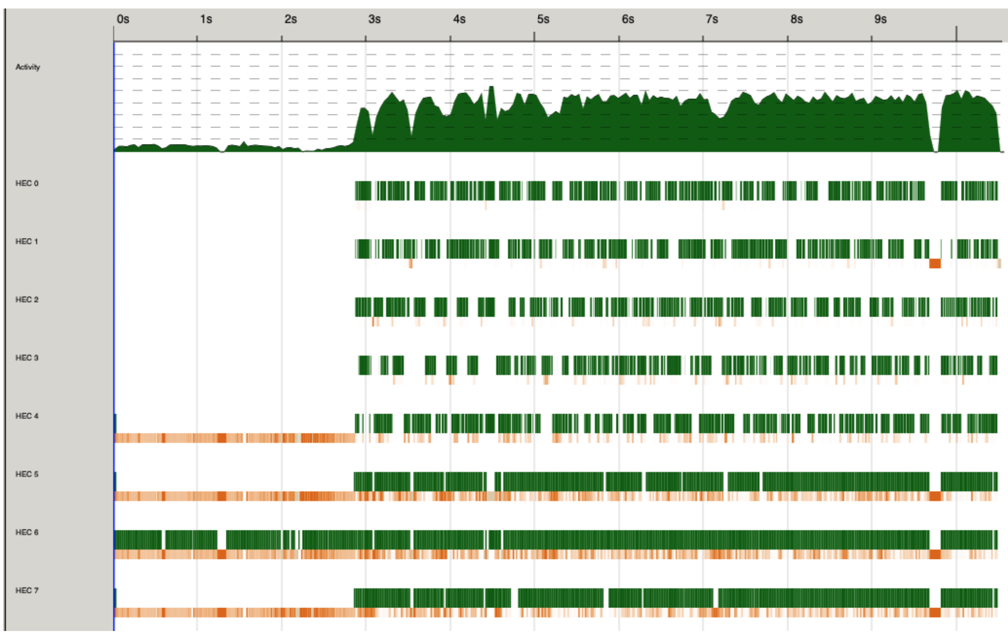
We ran the program using *gimlets* as start word and *affinage* as end word, with the maximum searching depth set to 50, on an 8-core machine. The performance on a 6 threads is as followed:

```
no ladder
   5,552,402,896 bytes allocated in the heap
     710,017,976 bytes copied during GC
     139,135,808 bytes maximum residency (10 sample(s))
         419,008 bytes maximum slop
             293 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2698 colls,  2698 par    2.619s   0.838s     0.0003s    0.0082s
  Gen  1        10 colls,     9 par    0.838s   0.306s     0.0306s    0.1298s

  Parallel GC work balance: 29.85% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

  SPARKS: 962180 (135075 converted, 0 overflowed, 0 dud, 177068 GC'd, 650037 fizzled)

  INIT    time    0.001s  (  0.015s elapsed)
  MUT     time   18.459s  (  6.346s elapsed)
  GC      time    3.456s  (  1.144s elapsed)
  EXIT    time    0.001s  (  0.011s elapsed)
  Total   time   21.917s  (  7.516s elapsed)

  Alloc rate    300,801,144 bytes per MUT second

  Productivity  84.2% of total user, 84.4% of total elapsed
```



The performance on 8 threads:

```
no ladder
  6,761,241,816 bytes allocated in the heap
  1,695,271,400 bytes copied during GC
    140,023,464 bytes maximum residency (10 sample(s))
        448,856 bytes maximum slop
            305 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3012 colls,  3012 par    5.940s   1.490s     0.0005s    0.0400s
  Gen  1        10 colls,     9 par    0.895s   0.371s     0.0371s    0.1381s

  Parallel GC work balance: 10.70% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

  SPARKS: 1087060 (306483 converted, 0 overflowed, 0 dud, 155176 GC'd, 625401 fizzled)

  INIT    time    0.001s  (  0.013s elapsed)
  MUT     time   21.671s  (  8.654s elapsed)
  GC      time    6.835s  (  1.861s elapsed)
  EXIT    time    0.000s  (  0.002s elapsed)
  Total   time   28.507s  ( 10.530s elapsed)

  Alloc rate    311,990,970 bytes per MUT second

  Productivity  76.0% of total user, 82.2% of total elapsed
```

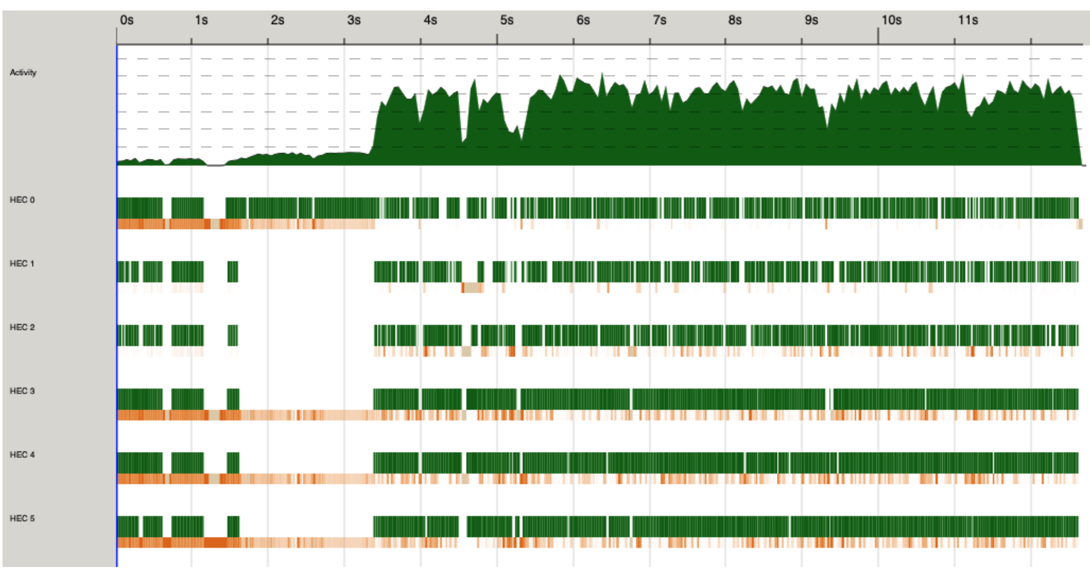


On a 6-thread parallelization, it is 1.9 times faster than the sequential version.

5.2.3 Parallel the dictionary reading part

As we mentioned before, we cast all the words to lower cases. That happens before the BFS starts. Surprisingly, after we parallelized this part, the performance did not improve much.
On 6 threads, there's an around 50% slowdown than the previous version.

```
no ladder
   5,594,024,920 bytes allocated in the heap
   1,112,421,600 bytes copied during GC
     229,661,288 bytes maximum residency (10 sample(s))
       3,761,560 bytes maximum slop
             451 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2744 colls,  2744 par    6.502s   1.775s     0.0006s    0.0625s
  Gen  1        10 colls,     9 par    1.335s   0.613s     0.0613s    0.3147s

  Parallel GC work balance: 27.48% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

  SPARKS: 1428875 (600862 converted, 0 overflowed, 0 dud, 178892 GC'd, 649121 fizzled)

  INIT    time    0.001s  (  0.008s elapsed)
  MUT     time   23.938s  ( 10.277s elapsed)
  GC      time    7.836s  (  2.388s elapsed)
  EXIT    time    0.000s  (  0.005s elapsed)
  Total   time   31.776s  ( 12.677s elapsed)

  Alloc rate    233,684,239 bytes per MUT second

  Productivity  75.3% of total user, 81.1% of total elapsed
```



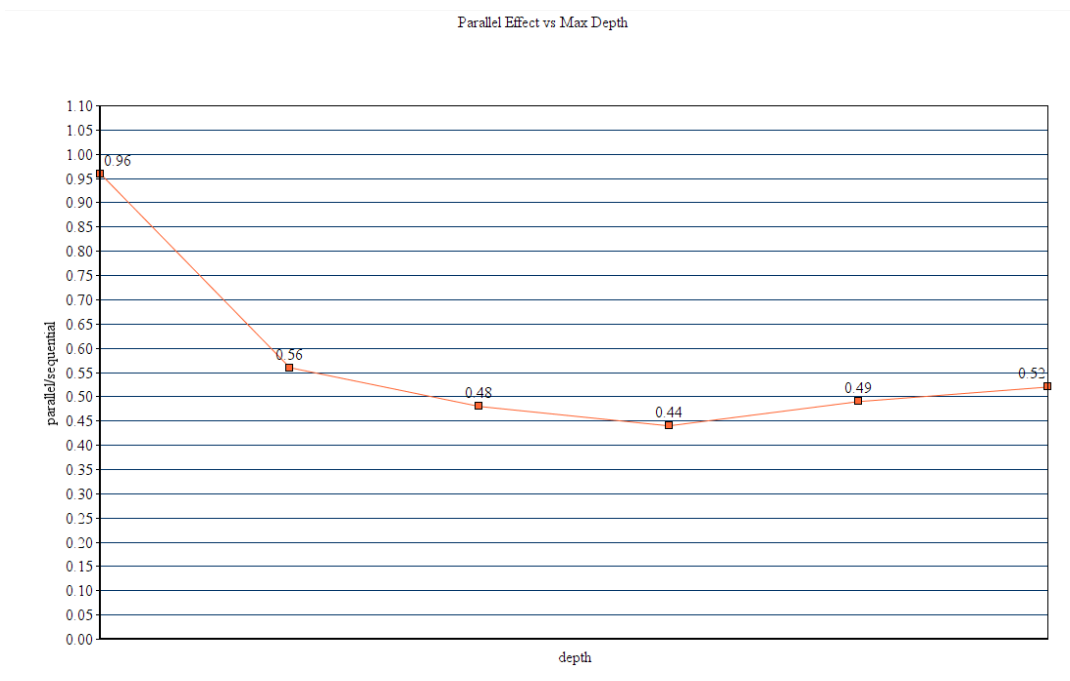
Our guessing is, even if the casting work is paralleled, this job is a relatively simple one which takes only a very short time. So parallelizing this part does not bring much benefit compared to the overhead of creating sparks, garbage collection, context switching, and so on.

6. Further Analysis

While testing the overall performance and effect of applying parallelization, we also monitored the effects of parallelism while changing the maximum allowed depth of tree, The performance of the result is in the table below. While sticking with the same start and end word, same word dictionary, run with the same degree of parallelism, we keep track of both sequential and parallel performance over different max depth and record the result into the table below.

	Sequential	**Parallel**
10	1.83s user , cpu 1.998 total	3.78s user , cpu 1.918 total
20	8.03s user , cpu 8.200 total	13.35s user , cpu 4.591 total
50	15.34s user , cpu 16.878 total	21.43s user , cpu 8.136 total
100	14.36s user , cpu 15.093 total	20.91s user , cpu 6.613 total
200	14.07s user , cpu 14.466 total	21.15s user , cpu 7.094 total
500	13.35s user , cpu 13.656 total	21.29s user , cpu 7.110 total

Compute the above table into a line graph, where y-axis represents the ratio of parallel run time versus sequential run time, and x-axis keeps track of the depth. We get the line chart below.



Based on the line chart, we don't see the effects of parallelism getting greater as the max depth increases. The degree of benefit brought by using parallelism has been roughly consistent as max depth increases.

7. Code

pfp_final.hs

```
import Data.Char(toLower)
import System.Environment(getArgs)
import qualified Data.Set as Set
import System.IO(hPutStrLn, stderr)
import System.Exit(exitFailure)
import Data.List as List
import Control.Parallel.Strategies hiding(parMap)
import Control.DeepSeq



type StringSet = Set.Set String
parMap :: (a -> b) -> [a] -> Eval [b]
parMap _ [] = return []
parMap f (a:as) = do
    b <- rpar (f a)
    bs <- parMap f as
    return (b:bs)

-- pmap :: (a -> b) -> [a] -> [b]
-- pmap f xs = map f xs `using` parList rseq


-- concatMap1             :: (a -> [b]) -> [a] -> [b]
-- concatMap1 f xs        =  concat $ pmap f xs

-- usage :: IO ()
-- usage = do
--    pn <- getProgName
--    die $ "Usage: " ++ pn ++ " <dictionary-filename> <from-word>
<to-word>"
```

```haskell
readDict :: String -> Int -> IO StringSet
readDict filename _ =
  (Set.fromList . (map (map toLower)) . words) `fmap` readFile
filename


search :: StringSet -> String -> String -> Int -> Maybe [String]
search dictionary fromWord toWord maxDepth =
  bfs [[fromWord]] (Set.singleton fromWord) maxDepth
  where
     bfs :: [[String]] -> StringSet -> Int -> Maybe [String]
     bfs _ _ 0 = Nothing
     bfs paths visited depth =
          case filter ((==toWord) . head) paths of
               (solution:_) -> Just solution
               [] -> bfs paths'' visited' (depth - 1)
             where
              paths' = concat $ runEval $ parMap takeAStep paths
              (paths'', visited') = foldr validStep ([], visited)
paths'
              validStep np@(w:_) (existing, v)
                 | not (Set.member w v)
                       = (np : existing, Set.insert w v)
                 | otherwise = (existing, v)
              validStep [] _ = error "validStep: empty list?"
              takeAStep :: [String] -> [[String]]
              takeAStep [] = error "takeAStep: empty list?"
              takeAStep p@(x:_) = concat $ runEval $ parMap
allletters $ zip (inits x) (tails x)
--                takeAStep p@(x:_) = concat $ zipWith (allletters)
(inits x) (tails x)
                where
                  -- pair = zip (inits x) (tails x)
                  -- helper p = zip (map fst p) (map (tail . snd)
(init p))
                  allletters (pre,w@(_:ws)) =runEval $ do
                                             as' <- rpar (force
(generateNextHop pre w))
```

```
                                                          bs' <- rpar (force
(generateNextHop pre ws))
                                                          rseq as'
                                                          rseq bs'
                                                          return (as' ++ bs')
                   allletters (_,[]) = []
--                   allletters pre w@(_:ws) = runEval $ do
--                                                              as' <- rpar
(force (generateNextHop pre w))
--                                                              bs' <- rpar
(force (generateNextHop pre ws))
--                                                              rseq as'
--                                                              rseq bs'
--                                                              return (as' ++
bs')
--                   allletters _ [] = []
                   generateNextHop pre w = [ b : p | c <- ['a'..'z'],
let b = (pre ++ [c] ++ w), Set.member b dictionary, not $ Set.member
b visited]
maxSteps :: Int
maxSteps = 200

main :: IO ()
main = do args <- getArgs
          case args of
            [filename, start, end] -> do
               contents <- readDict filename (length start)
               case search contents start end maxSteps of
                Nothing -> putStrLn "no ladder"
                Just _ -> putStrLn "parallelized"

            _ -> do
              -- pn <- getProgName
                    hPutStrLn stderr $ "Usage: wordLadder
<dictionary-filename> <from-word> <to-word>"
                    exitFailure
```