WordLadder

Parallel Functional Programming Fall 2022

Aruj Jain (aj2933) and Benjamin Magid (bmm2179)

December 20, 2022

---

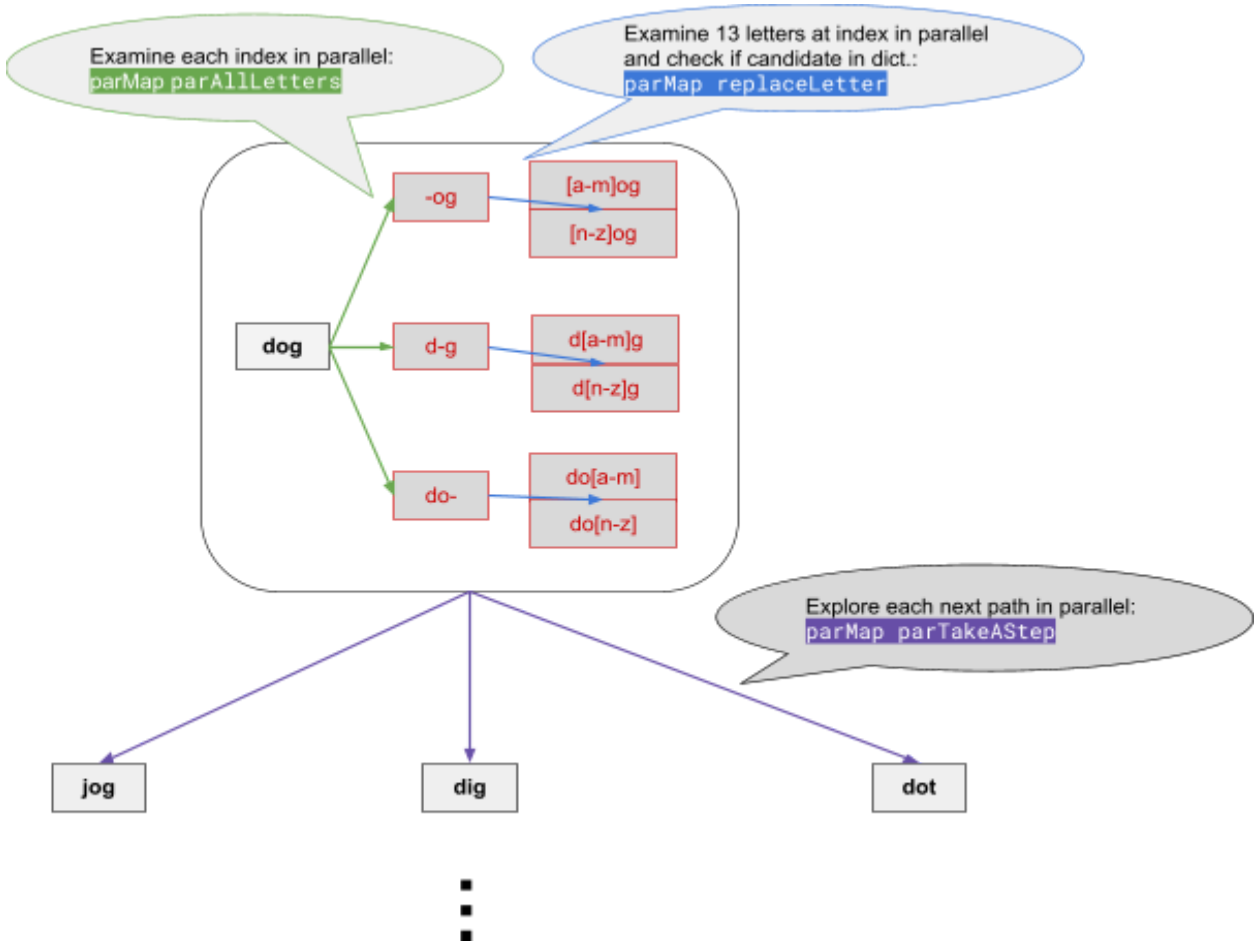
1. Background

The WordLadder program takes in a dictionary and two words of equal length, and returns a sequence of words from the dictionary beginning with the start word and ending the target word by only changing one letter at a time. If the given words are not in the dictionary or the program cannot find a sequence of length 20 or shorter, the program assumes no word ladder exists. We developed this project based on the sequential version of a word ladder solver explored in Homework 4 and similarly implemented a BFS-style search.

To make the problem more applicable, we used a larger dictionary (found here) than what we used in class. This dictionary, while larger, was already pruned for lowercase, alphanumeric words only, and we focused our parallelization efforts on the BFS-style search instead of any possible optimizations to the diocnary file IO.

2. Implementation

Overall Architecture and Limitations



Seeing as we are parallelizing a breadth first search style algorithm, there were some natural limitations to the amount of parallelization we could implement. BFS requires an up-to-date set of visited nodes to ensure we are looking for the shortest path. By design therefore, we must end our concurrent calculations at each level in the search tree to synchronize our list of visited candidate words. This is precisely our limitation and approach: how much can we efficiently parallelize in each level of the tree?

The first and most obvious place to parallelize is the exploration of each child (purple above). We can generate a new spark for each branch we explore, ready to sync back our list of visited nodes once we have explored all the children at the next level.

For each node (potential next word in the path), we then parallelize the exploration of each index of that word. If a word is 5 letters long, we generate 5 sparks to explore its potential children in parallel. This parallelization is highlighted in green above.

These two points of parallelization were clear from the start, and we streamlined their parallel implementation by writing our own `parMap` function. What was less clear and required some experimentation was how we can parallelize the modification of a letter at a given index. Now that we have different sparks for every branch and every index of a word, can we efficiently speed up the process of trying out all 26 letters in that position?

First Attempts

Implementation #1: Naive ReplaceLetter

Our first approach was to naively apply our parMap function to the generation of all the letters at a given index. This meant generating 26 sparks for each index to do a relatively inexpensive computation, string concatenation.
As expected, the overhead and garbage collection from this spark generation proved to be slower than the sequential implementation especially because most of the candidate words generated from this process were not in the dictionary, so they were later thrown out of the candidate list when we sequentially sycnhronize the tree level.

```
parTakeAStep :: [String] -> [[String]]
parTakeAStep [] = error "takeAStep: empty list?"
parTakeAStep p@(x:_) = concat $ parMap allLetters $ zip (inits x) (tails
x)
   where
       allLetters (pre,(_:ws)) = replaceLetter pre ws
       allLetters (_,[]) = []
       replaceLetter pre w = [ b : p | c <- ['a'..'z'], let b = (pre ++
[c] ++ w) ]
```

Implementation #2: Parallel ReplaceLetter with Chunk Size = 1

Our next approach leveraged these 26 sparks generated for each letter at a given index to compute a more expensive instruction, checking if a candidate word is in the dictionary and not in our list of visited candidates so far. As mentioned before, we still at each level have to check our list of visited nodes sequentially, but we can save ourselves time by generating better quality candidate words. These 26 sparks now generate all the valid words created from modifying a letter at a given index. This method proved to be about as fast as our sequential benchmark

because of the overhead involved in generating so many sparks for a still relatively small process.

```
parAllLetters :: (String, String) -> [[String]]
parAllLetters (_, []) = []
parAllLetters (pre, suf) = concat $ parMap replaceLetter [(['a'..'z'],
(pre, suf))]

replaceLetter :: ([Char], (String, String)) -> [[String]]
replaceLetter (_, (_, [])) = error "replaceLetter: empty suffix?"
replaceLetter (lc, (pre, (_:ws))) =
     [ b : p | c <- lc, let b = (pre ++ [c] ++ ws), Set.member b
     dictionary, not $ Set.member b visited]
```

Final Implementation

Implementation #3: Parallel ReplaceLetter with Chunk Size = 13

After some testing, we found that we can split the alphabet into two parts and do the word candidacy testing with two sparks. This proved faster than our sequential benchmark as two sparks to test 13 letter modifications and set memberships was the right balance of parallel speed up and garbage collection slow down. As in the previous implementation, we utilize each working spark to not only generate candidate words with letter modifications in that index; we also check if these are valid candidates in the dictionary and not already seen.

```
parAllLetters :: (String, String) -> [[String]]
parAllLetters (_, []) = []
parAllLetters (pre, suf) = concat $ parMap replaceLetter [(lc, (pre, suf))
| lc <- chunksOf 13 ['a'..'z']]
```

3. Performance

Machine Specs: *Macbook Air 2020, Apple M1 Chip, 8 Cores (4 performance and 4 efficiency)*

Let us examine the runtime of the three different versions of our word ladder program to evaluate the practicality of each approach, and to explore the relationship between the number of cores used and the runtime.

The runtime of the sequential word ladder program by Professor Stephen Edwards returns an approximate average runtime of **0.750 seconds**, which serves as a helpful baseline to try and beat with the parallel implementation.

Runtime Data Tables

Table 1: Runtimes (seconds) for Implementation #1

Number of Cores	Trial 1	Trial 2	Trial 3	Average
2	1.110	1.181	1.190	1.160
4	1.108	1.243	1.227	1.193
8	1.301	1.399	1.303	1.334

Table 2: Runtimes (seconds) for Implementation #2

Number of Cores	Trial 1	Trial 2	Trial 3	Average
2	0.638	0.613	0.669	0.640
4	0.663	0.687	0.653	0.668
8	0.824	0.694	0.764	0.761

Table 3: Runtimes (seconds) for Implementation #3

Number of Cores	Trial 1	Trial 2	Trial 3	Average
2	0.514	0.525	0.503	0.514
4	0.489	0.495	0.498	0.494
8	0.658	0.728	0.653	0.680

4. Threadscope Outputs

ThreadScope was a very useful tool in evaluating the performance of different variations of our program, and using that information to fine tune certain functions. Some of the key metrics that we examined were program runtime (seconds), number of sparks created/converted, and how many sparks went to garbage collection. We felt the best performance would be reflected by variations of the program that had the fastest runtime while minimizing garbage collection.

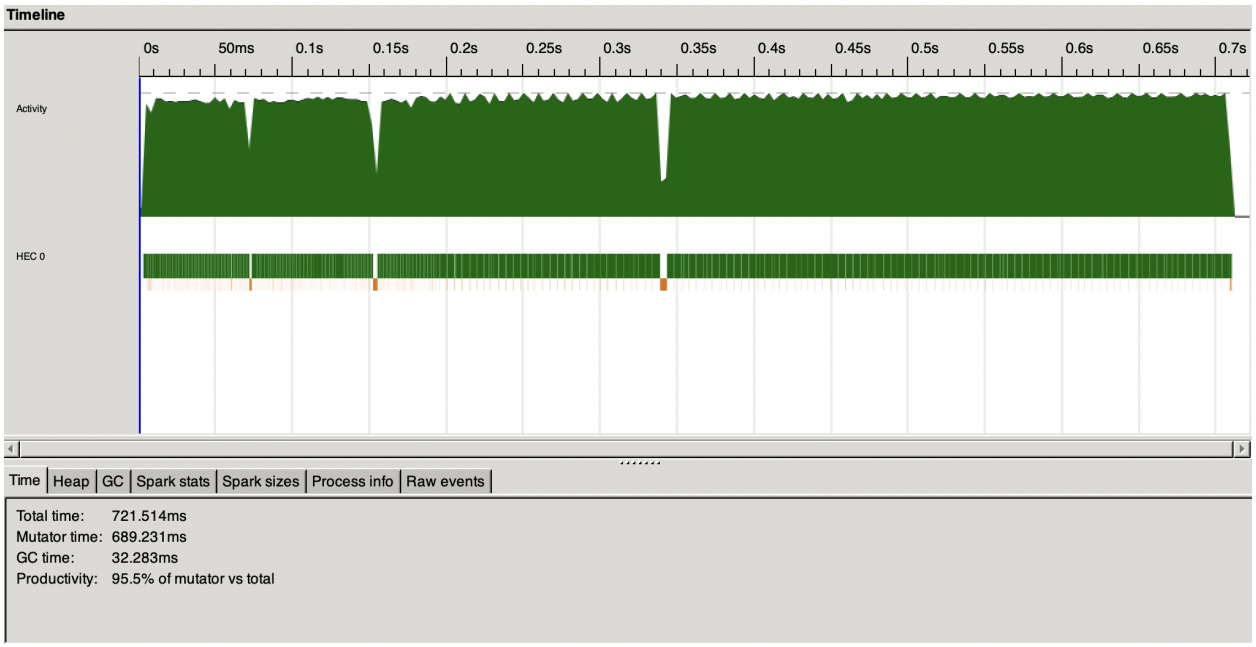


Figure 1: Threadscope for Sequential Implementation

Implementation #1: Naive ReplaceLetter

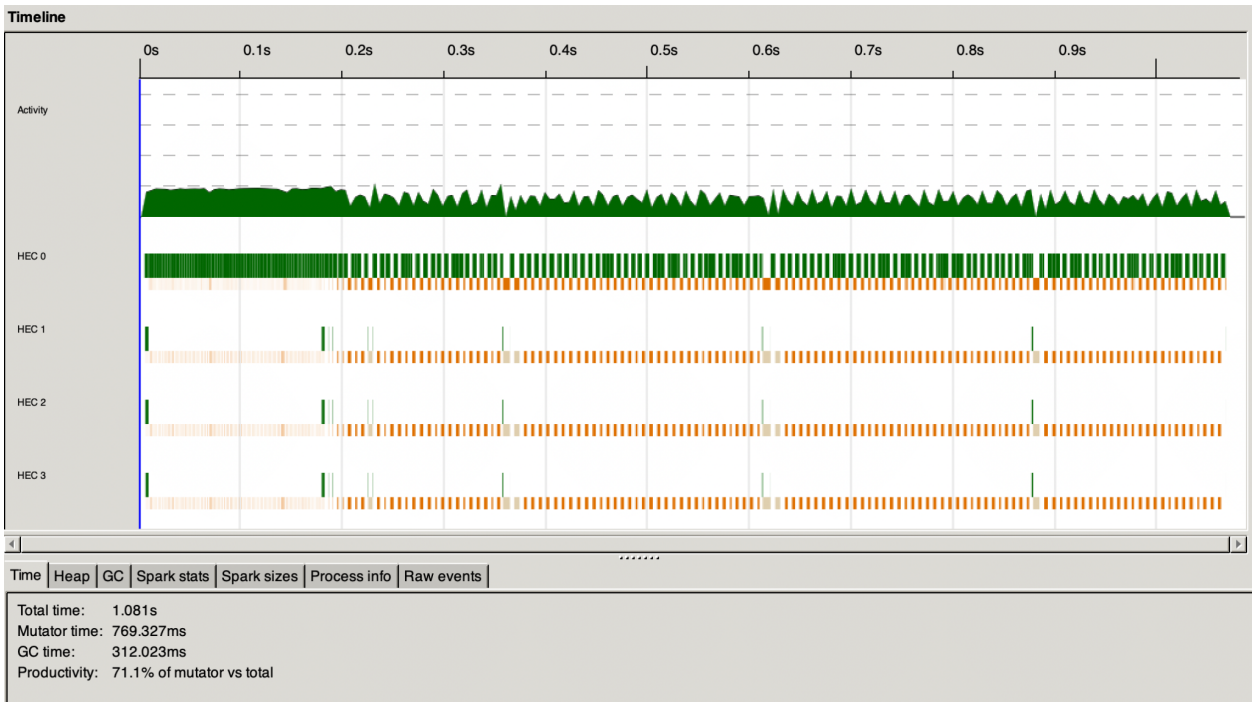


Figure 2: Threadscope for Parallel Implementation #1 with 4 Cores

Implementation #2: Parallel ReplaceLetter with Chunk Size = 1

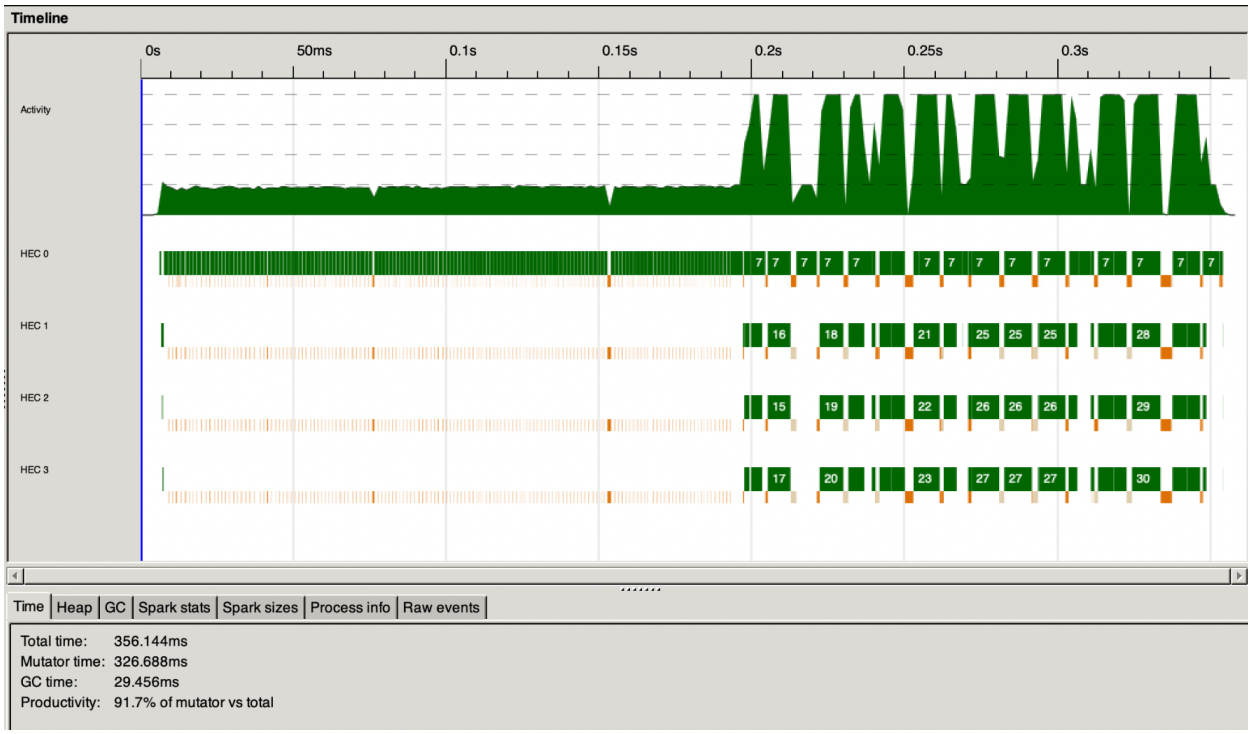


Figure 3: Parallel Parallel Implementation #2 with 4 Cores

Implementation #3: Parallel ReplaceLetter with Chunk Size = 13

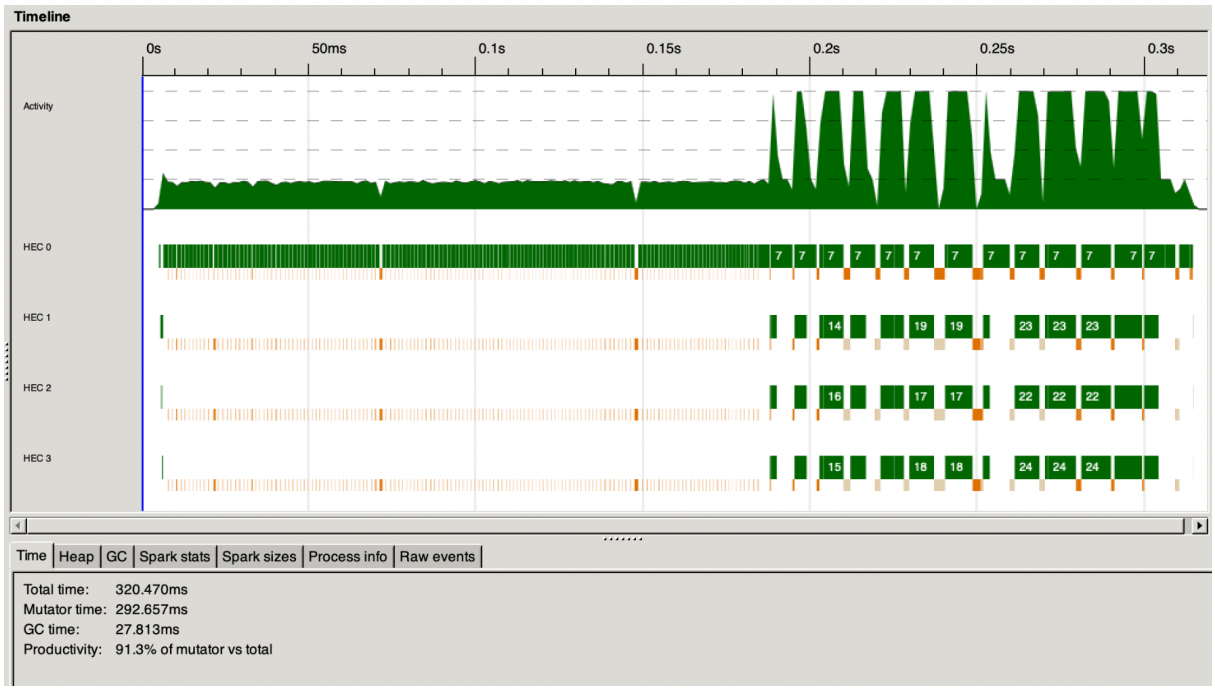


Figure 4: Parallel Implementation #3 with 4 Cores

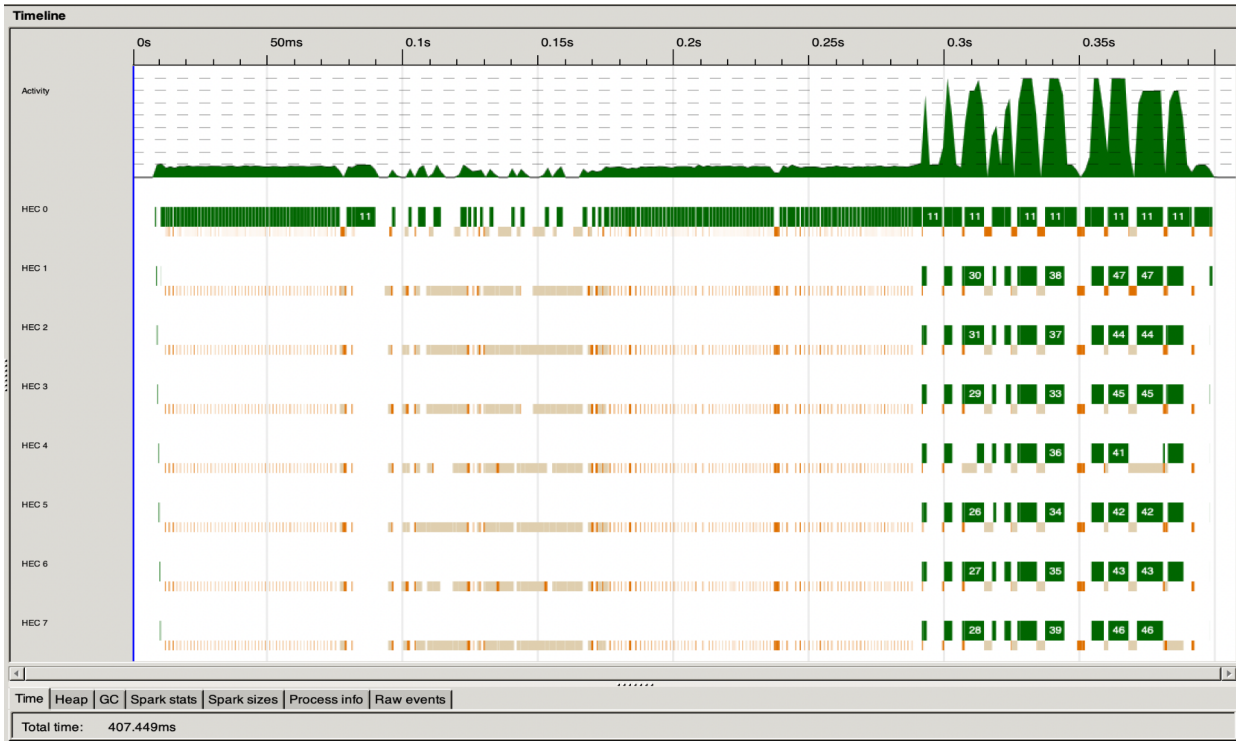


Figure 5: Parallel Implementation #3 with 8 Cores

5. Conclusion

The chunking method implemented in the final version of the program proved to be useful in reducing runtime. Adding the dictionary candidacy check for generated one-letter modifications to the parallel AllLetters function was also successful since it reduced the amount of sequential list validation needed.

As seen in Table 3, running the final implementation on 4 cores leads to the best overall performance across all implementations. This best average runtime of **0.494 seconds** is approximately **34% faster** than the average sequential average runtime (of 0.750 s). Across all implementations, running on 8 cores proved to result in the worst runtime performance. This held true when testing the program on both Apple M1 chip and Intel chip systems, suggesting that the current final implementation cannot make use of too many cores to optimize performance further.

For future development and improvement, we could try parallelizing the reading of the dictionary which was outside the scope of this project. As can be seen in the threadscope outputs, this is a majority of the processing time, so this may be a space for further optimizing the runtime of the algorithm. We could also make the problem harder by allowing the start and end words to be different lengths. This would greatly increase the branching factor for any given word as we can insert, delete, or modify the letters at each index.

Overall, we found, as expected, moderate improvements to performance by parallelizing the WordLadder problem, but were limited by the nature of the algorithm itself as a breadth-first search requires a level-by-level synchronization.

6. Code

```
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import qualified Data.Set as Set
import Control.Monad(unless)
import Data.List(inits, tails)
import qualified Control.Parallel.Strategies as P hiding(parMap)
import Data.List.Split (chunksOf)

type StringSet = Set.Set String

usage :: IO ()
usage = do
```

```
    pn <- getProgName
    die $ "Usage: "++pn++" <dictionary-filename> <from-word> <to-word>"

readDict :: String -> Int -> IO StringSet
readDict filename wordLength =
    (Set.fromList . filter validWord . words) `fmap` readFile filename
    where validWord w = length w == wordLength

parMap :: (a -> b) -> [a] -> [b]
parMap f = P.withStrategy (P.parList P.rseq) . map f

search :: StringSet -> String -> String -> Int -> Maybe [String]
search dictionary fromWord toWord maxDepth =
    search1 [[fromWord]] (Set.singleton fromWord) maxDepth
    where
        search1 :: [[String]] -> StringSet -> Int -> Maybe [String]
        search1 _ _ 0 = Nothing
        search1 paths visited depth =
            case filter ((==toWord) . head) paths of
                (solution:_) -> Just solution
                [] -> search1 newPaths newVisited (depth-1)
                where
                    paths' = concat $ parMap parTakeAStep paths --
parallelize for each immediate next path from node

                    parTakeAStep [] = error "takeAStep: empty list?"
                    parTakeAStep p@(x:_) = concat $ parMap parAllLetters $
zip (inits x) (tails x) -- parallelize for each index of the word
                        where
                            -- replace the letter at the given index with a
letter from a-z in parallel
                            parAllLetters :: (String, String) -> [[String]]
                            parAllLetters (_, []) = []
                            parAllLetters (pre, suf) = parMap replaceLetter
[(lc, (pre, suf)) | lc <- chunksOf 13 ['a'..'z']]
```

```
                            replaceLetter :: ([Char], (String, String)) ->
[String]
                            replaceLetter (_, (_, []))  = error
"replaceLetter: empty suffix?"
                            replaceLetter (lc, (pre, (_:ws))) =
                                [ b | c <- lc, let b = (pre ++ [c] ++ ws),
Set.member b dictionary, not $ Set.member b visited] ++ p

                    (newPaths, newVisited) = foldr validStep ([], visited)
paths'
                    validStep np@(w:_) (exisitng, v)
                        |not (Set.member w v)
                            = (np:exisitng, Set.insert w v)
                        |otherwise = (exisitng, v)
                    validStep [] _ = error "validStep: empty list?"

maxSteps :: Int
maxSteps = 20

main :: IO ()
main = do
   args <- getArgs
   case args of
       [dictFile, fromWord, toWord] -> do
           unless (length fromWord == length toWord) $ do
               die $ "Words must be the same length"
           dictionary <- readDict dictFile (length fromWord)
           unless (Set.member fromWord dictionary) $ die $
               "Word \""++fromWord++"\" not in dictionary"
           unless (Set.member toWord dictionary) $ die $
               "Word \""++toWord++"\" not in dictionary"
           let solution = search dictionary fromWord toWord maxSteps

           case solution of
               Nothing -> die "No solution found"
               Just sol -> do
```

```
                mapM_ putStrLn $ reverse sol
    _ -> usage
```