ParWordle: Parallelized Entropy-based Wordle Solver

Sanjay Rajasekharan & Zac Coeur
December 22, 2022

1 Introduction

Wordle is a popular word game in which a player is given six guesses to decipher a hidden 5-letter word. After each guess, the user is given feedback for each letter in their guess. If a letter in the guess is located at the same position in the answer, then the letter is marked green. If a letter appears in the answer but not in the same position, the letter is marked yellow. Finally, if the letter does not appear in the answer, the letter is marked grey. The player is therefore tasked with assembling the feedback from their previous guesses to find the right answer.

2 Wordle Algorithm

2.1 Background

The algorithm takes an information theory based approach to selecting guesses. The optimal guess maximally reduces of the set of possible answers. In other words, it provides the most information, I. The algorithm, therefore, selects the word with the largest expected information (entropy).

$$p(x) = \frac{\text{\# answers that generate } x}{\text{\# possible answers}} \tag{1}$$

$$I(x) = \log_2(1/p(x)) \tag{2}$$

$$E[I] = \sum_x p(x) * I(x) \tag{3}$$

In the equation above x represents each possible pattern of feedback that can be received after a guess. Additionally, this pattern also represents a subset of the answer space because each feedback pattern uniquely subsets the answers. The distribution of feedback patterns is composed by computing the feedback that would be received for each answer. $p(x)$ captures the frequency of a pattern occurring, and $1/p(x)$ captures the reduction of the answer space. Therefore by looping over each answer, the expected value of a possible guess can be calculated using just $p(x)$.

Algorithm 1 Wordle Guessing Algorithm

1: $A = W = S$ ▷ S is the set of all 5-letter words
2: G = {}
3: **initialize** K ▷ K represents our knowledge about the answer
4: $g =$ **argmax** $\{E(w)|w \in W\}$
5: **while** $g \neq a$ **do** ▷ a is the Wordle solution
6: $K \cup guess(g)$
7: $A = K \rightarrow A$ ▷ filter possible answers utilizing new knowledge
8: $G = G \cup \{g\}$
9: $g =$ **argmax** $\{E(w)|w \in W\}$
10: **end while**
11: **output** G

To solve the word, the algorithm initializes two sets, representing possible answers and allowed guesses. At the start of the game both sets are equivalent to all 5-letter words. The algorithm also keeps track of a knowledge base which contains the union of the all the feedback received. Guesses are made by choosing the word that has the largest entropy. After each guess, if the guess does not equal the answer, the knowledge base, answer space, and previous guesses are updated. Once the algorithm guesses the right answer, the game is over.

2.2 Haskell Implementation

The Wordle guessing algorithm is housed in two scripts: par-wordle-exe and seq-wordle-exe. Each take two command line parameters, an answer and a list of allowed guesses. They can be run as such ./seq-wordle-exe <answer> <guesses-file>. The executables will print the guesses the solver makes until it guesses correctly. The first guess is always "raise", because without any knowledge of the previous word, "raise" has the highest entropy. One of the features of our implementation was the creation of custom type, Knowledge, which can be used to track the feedback provided by a singular guess, or it can track cumulative feedback from multiple. Knowledge utilizes 3 sets. Two track green and yellow clues, which contain (position, character) tuples. The final set tracks grey clues. The grey set contains just characters as position information is irrelevant.

```
data Knowledge = Knowledge { green :: Set.Set (Int, Char)
                           , yellow :: Set.Set (Int, Char)
                           , grey :: Set.Set Char } deriving (Eq, Ord, Show)
initKnowledge :: Knowledge
initKnowledge = Knowledge Set.empty Set.empty Set.empty

-- Updates knowledge for a guess-answer pair
addToKnowledge :: Knowledge -> B.ByteString -> B.ByteString -> Knowledge
addToKnowledge k g a = Knowledge greens yellows greys
   where greens = green k `Set.union` getGreens g a
         yellows = yellow k `Set.union` getYellows g a
         greys = grey k `Set.union` getGreys g a
```

Additionally, although all code can be found in the Code Listings section, the implementation of $E[x]$ is shown below. The function utilizes Data.Map to compute the frequency of distinct Knowledge values. In this case the Knowledge values represent the feedback patterns described in section 2.1.

```
entropy :: B.ByteString -> Knowledge -> Set.Set B.ByteString ->Float
entropy g k as = Map.foldl (+) 0 $ Map.unionWith (*) p inf
   where l = fromIntegral (Set.size as)
         p =  Map.map (/l) $ count (map (addToKnowledge k g) (Set.toList as))
         inf = Map.map (\x ->logBase 2 (1/x)) p
```

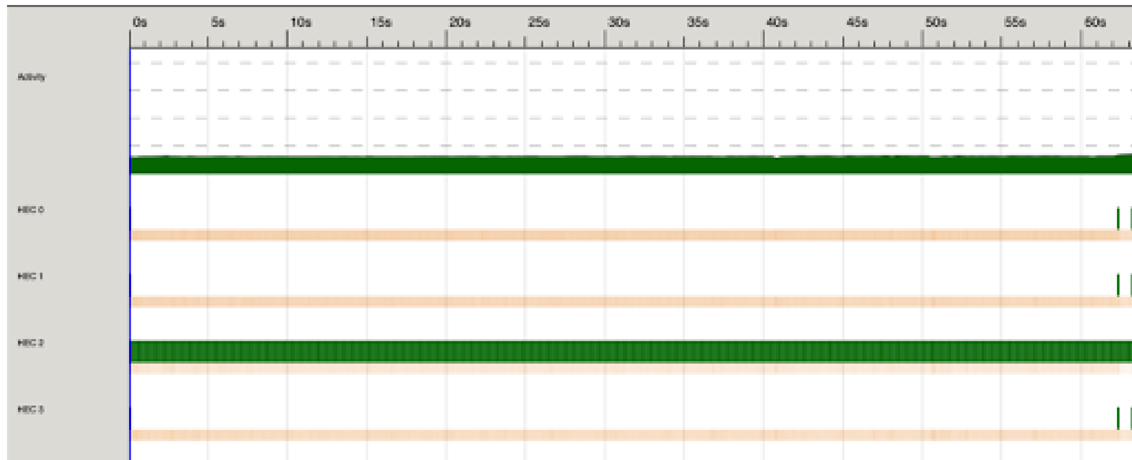
3 Parallelization

The parallelization efforts focused on the entropies function, which maps over the 12,973 word list and calculates the entropy for each guess by calling the entropy function. Each word's entropy is completely independent of the other words, making it a prime candidate for parallelization. Described below are the functions that were attempted to maximize speedup. The following parallelization techniques were added to the last line of entropies in the map function.

```
entropies :: Set.Set B.ByteString -> Knowledge -> [(B.ByteString, Float)]
entropies gs k = e_list
   where g_list = Set.toList gs
         as = possibleWords k gs
         e_list = map (\g -> (g, entropy g k as)) g_list
```

3.1 parList rseq

The first attempt was a basic implementation of parList rseq. However, this resulted in very little parallelization as seen from the threadscope screenshot. All sparks were converted, but they only sparked on one core. It was also much slower than the sequential implementation (by a factor of 2). The issue was that rseq does not force evaluation, and the word list is of type ByteString, which is a lazy datatype.

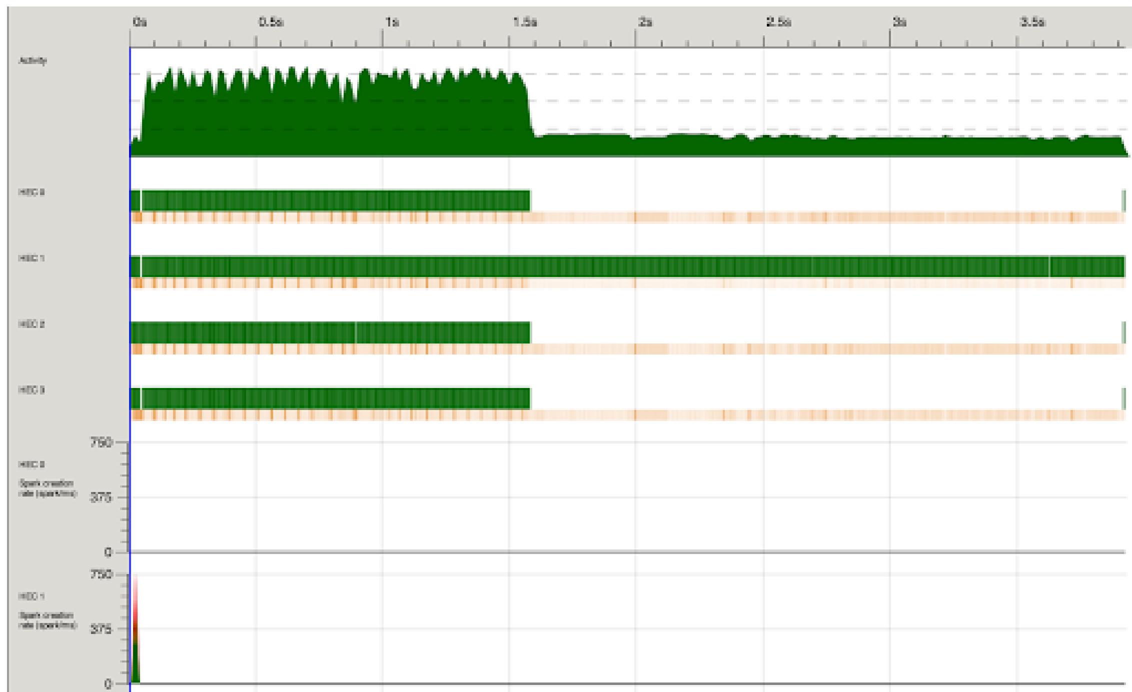


	sequential	**parList rseq**
"hello"	39.17s	63.71s
"inter"	5.84s	6.69s
"shine"	2.26s 8	1.64s

Table 1: Performance of parList rseq

3.2 parList rdeepseq

To fix the laziness issue, rseq was changed to rdeepseq. Evaluation was forced, but the algorithm only ran in parallel for about half of the runtime, and resulted in a runtime relatively similar to the sequential implementation. The issue with this strategy is the number of sparks. At the beginning of the algorithm, the spark creation rate shot up (as shown in threadscope image), the spark pool reached its capacity, and sparks overflowed because of the large size of the list.

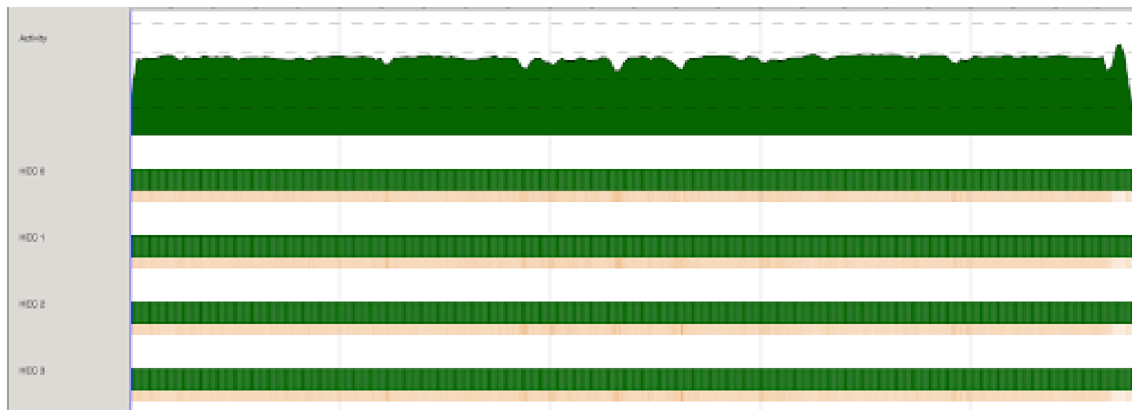


	sequential	**parList rdeepseq**
”hello”	39.17s	36.79s
”inter”	5.84s	3.93s
”shine”	2.26s 8	3.03s

Table 2: Performance of parList rdeepseq

3.3 parBuffer n rdeepseq

Our final strategy was using parBuffer n rdeepseq. In testing, the performance when n=200 was consistently the best when compared with other values, as shown in the table. By using parBuffer 200 rdeepseq, the spark pool never eclipsed 200 sparks, and never overflowed. Almost all sparks converted, and all 4 cores ran in parallel throughout the program as shown in the Threadscope image. Overall, parBuffer 200 rdeepseq resulted in the greatest speedup when compared with other strategies (1.5x-5.5x depending on number of cores) and is the final implementation of parallelism in the Wordle algorithm.

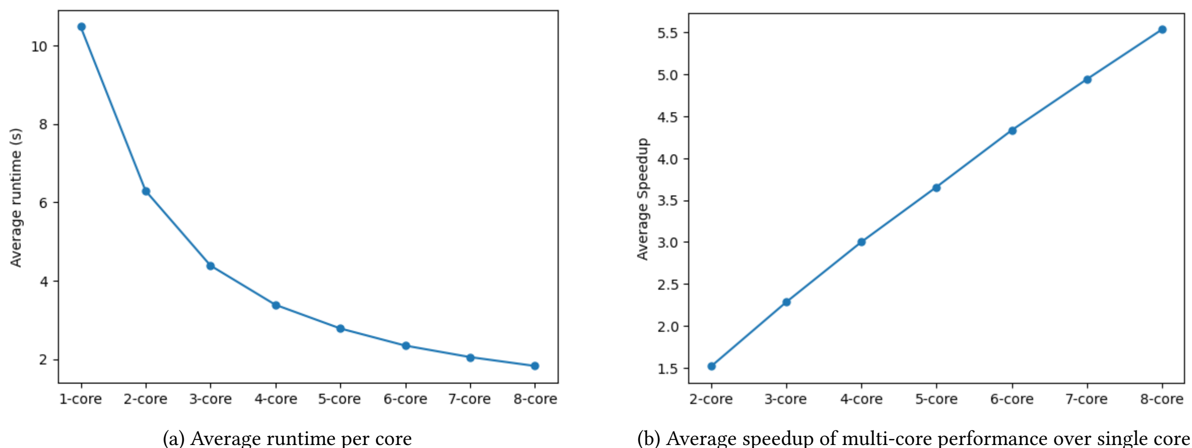


	sequential	**n=100**	**n=200**	**n=300**	**n=400**
”hello”	39.17s	20.58s	20.34s	20.83s	30.17s
”inter”	5.84s	2.66s	2.53s	3.04s	4.22s
”shine”	2.26s 8	1.22s	1.07s	1.02s	1.16s

Table 3: Difference in runtime when n is changed in parBuffer n rdeepseq

4 Results and Conclusion

The results below show the decrease in runtime and increase in speedup in the Wordle solver dependent on the number of cores. The data was taken on a sample size of 500 words on a machine with 8 CPU cores. By testing multiple parallel strategies and selecting parBuffer 200 rdeepseq, the resulting parallel Wordle Solver performs exceedingly well, achieving a speedup of 5.53x with 8 cores. Average speedup was calculated by running the same word on each core count and computing the speedup per word. Using parBuffer keeps the spark pool constant to eliminate overflow, allowing each core to run in parallel for almost the entire length of the program. The final implementation maximizes speedup while also retaining elegance.



(a) Average runtime per core

(b) Average speedup of multi-core performance over single core

Figure 1: Per core performance improvement

cores	avg. runtime (s)
1	10.47
2	6.29
3	4.38
4	3.38
5	2.78
6	2.34
7	2.04
8	1.82

cores	speedup
2	1.52
3	2.28
4	3.00
5	3.66
6	4.33
7	4.94
8	5.53

Table 4: Runtime and Speedup data

5 Code Listings

5.1 Lib.hs

```
module Lib
    ( playPar,
      playSeq,
      getValidWords,
      initKnowledge
      )
where

import qualified Data.Set as Set
import qualified Data.Map as Map
import qualified Data.ByteString.Char8 as B
import Control.Parallel.Strategies

data Knowledge = Knowledge { green :: Set.Set (Int, Char)
                           , yellow :: Set.Set (Int, Char)
                           , grey :: Set.Set Char } deriving (Eq, Ord, Show)

initKnowledge :: Knowledge
initKnowledge = Knowledge Set.empty Set.empty Set.empty


-- parallel wordle playing
playPar :: B.ByteString -> B.ByteString -> Knowledge -> Set.Set B.ByteString -> Int -> IO ()
playPar g a k gs gCount = do
   if g == a then do
      return ()
   else do
      let k' = addToKnowledge k g a
          gs' = Set.delete g gs
          g' = fst $ maxEntropyPar gs' k'
      print g'
      playPar g' a k' gs' (gCount+1)

-- sequential wordle playing
playSeq :: B.ByteString -> B.ByteString -> Knowledge -> Set.Set B.ByteString -> Int -> IO ()
playSeq g a k gs gCount = do
   if g == a then do
      return ()
   else do
      let k' = addToKnowledge k g a
          gs' = Set.delete g gs
          g' = fst $ maxEntropy gs' k'
      print g'
      playSeq g' a k' gs' (gCount+1)

-- Converts a list of ByteStrings to a Set of words of length = 5.
getValidWords:: [B.ByteString] -> Set.Set B.ByteString
getValidWords = Set.fromList . filter (\x -> B.length x == 5)

{-
Takes an array of (index, character) and a Set of ByteString. Filters the Set to only include words that
contain character at index.
-}
matchGreens:: [(Int, Char)] -> Set.Set B.ByteString ->Set.Set B.ByteString
matchGreens [] w = w
matchGreens ((i,c):xs) w = Set.intersection (Set.filter (\x -> B.index x i == c) w) (matchGreens xs w)

{-
Takes an array of characters and a Set of ByteString. Filters the Set to only include words that
include all characters.
-}
matchYellows:: [(Int, Char)] -> Set.Set B.ByteString -> Set.Set B.ByteString
matchYellows [] w = w
```

```haskell
matchYellows ((i, c):xs) w = Set.intersection (Set.intersection (Set.filter (B.elem c) w)
    (Set.filter (\x -> B.index x i /= c) w)) (matchYellows xs w)

{-
Takes an array of characters and a Set of ByteString. Filters the Set to only include words that
do not include any of the characters.
-}
matchGreys:: String -> Set.Set B.ByteString -> Set.Set B.ByteString
matchGreys xs w
  = foldr (\ x -> Set.intersection (Set.filter (B.notElem x) w)) w xs


-- Takes a guess and returns a Set of valid possible words.
possibleWords:: Lib.Knowledge -> Set.Set B.ByteString -> Set.Set B.ByteString
possibleWords k g =
    Set.intersection (Set.intersection (matchGreens (Set.toList (green k)) g)
    (matchYellows (Set.toList (yellow k)) g)) (matchGreys (Set.toList (grey k)) g)

-- Computes the yellow letters of a guess for a given answer
getYellows :: B.ByteString -> B.ByteString -> Set.Set (Int, Char)
getYellows g a = l `Set.difference` getGreens g a
   where s = Set.fromList $ B.unpack a
         l = Set.fromList $ filter (\x -> snd x `Set.member` s) ([0..] `zip` B.unpack g)

-- Computes the green letters of a guess for a given answer
getGreens:: B.ByteString -> B.ByteString -> Set.Set (Int, Char)
getGreens g a =
    Set.fromList ([0..] `zip` B.unpack g) `Set.intersection` Set.fromList ([0..] `zip` B.unpack a)


-- Computes the grey letters of a guess for a given answer
getGreys:: B.ByteString -> B.ByteString -> Set.Set Char
getGreys g a = s `Set.difference` ng
   where s = Set.fromList $ B.unpack g
         ng = Set.map snd (getGreens g a) `Set.union` Set.map snd (getYellows g a)


-- Updates knowledge for a guess-answer pair
addToKnowledge :: Knowledge -> B.ByteString -> B.ByteString -> Knowledge
addToKnowledge k g a = Knowledge greens yellows greys
   where greens = green k `Set.union` getGreens g a
         yellows = yellow k `Set.union` getYellows g a
         greys = grey k `Set.union` getGreys g a

-- Creates a frequency map from a list
count :: Ord a => [a] -> Map.Map a Float
count = Map.fromListWith (+) . (`zip` repeat 1)

-- computes the entropy of a guess
entropy :: B.ByteString -> Knowledge -> Set.Set B.ByteString ->Float
entropy g k as = Map.foldl (+) 0 $ Map.unionWith (*) p inf
   where l = fromIntegral (Set.size as)
         p =  Map.map (/l) $ count (map (addToKnowledge k g) (Set.toList as))
         inf = Map.map (\x ->logBase 2 (1/x)) p

-- parallel computes the entropies of all guesses
entropiesPar :: Set.Set B.ByteString -> Knowledge -> [(B.ByteString, Float)]
entropiesPar gs k = e_list
   where g_list = Set.toList gs
         as = possibleWords k gs
         e_list = map (\g -> (g, entropy g k as)) g_list `using` parBuffer 200 rdeepseq
-- sequentially computes the entropies of all guesses
entropies :: Set.Set B.ByteString -> Knowledge -> [(B.ByteString, Float)]
entropies gs k = e_list
   where g_list = Set.toList gs
         as = possibleWords k gs
         e_list = map (\g -> (g, entropy g k as)) g_list
```

```haskell
maxEntropyHelper :: (B.ByteString, Float) -> [(B.ByteString, Float)] -> (B.ByteString, Float)
maxEntropyHelper m []  = m
maxEntropyHelper (max_guess, max_entr) ((guess, entr):xs)
   | entr > max_entr = maxEntropyHelper (guess, entr) xs
   | otherwise = maxEntropyHelper (max_guess, max_entr) xs

-- parallel computes the maximum entropy of all guesses
maxEntropyPar :: Set.Set B.ByteString -> Knowledge ->  (B.ByteString, Float)
maxEntropyPar gs k
   | Set.size as == 1 = (Set.elemAt 0 as, 0)
   | otherwise = maxEntropyHelper (B.empty, -1) (entropiesPar gs k)
   where as = possibleWords k gs

-- sequentially computes the maximum entropy of all guesses
maxEntropy :: Set.Set B.ByteString -> Knowledge ->  (B.ByteString, Float)
maxEntropy gs k
   | Set.size as == 1 = (Set.elemAt 0 as, 0)
   | otherwise = maxEntropyHelper (B.empty, -1) (entropies gs k)
   where as = possibleWords k gs
```

5.2 Par.hs

```haskell
module Main
   ( main )
where
import Lib
import qualified System.Environment as Env
import qualified System.Exit as Exit
import qualified Data.ByteString.Char8 as B

main :: IO ()
main = do
   args <- Env.getArgs
   if length args /= 2 then do
      n <- Env.getProgName
      Exit.die $ "Usage: " ++ n ++ "<answer> <filename>"
   else do
      let a = head args
      if length a /= 5 then do
         Exit.die "Starting guess must be length 5"
      else do
         let f = args !! 1
         f' <- B.readFile f
         let gSet = (getValidWords . B.words) f'
             k = initKnowledge
             a' = B.pack a
             g = B.pack "raise"
         print g
         playPar g a' k gSet 1
```

5.3 Seq.hs

```haskell
module Main
   ( main )
where
import Lib
import qualified System.Environment as Env
import qualified System.Exit as Exit
import qualified Data.ByteString.Char8 as B

main :: IO ()
main = do
   args <- Env.getArgs
   if length args /= 2 then do
      n <- Env.getProgName
      Exit.die $ "Usage: " ++ n ++ "<answer> <filename>"
```

```haskell
else do
   let a = head args
   if length a /= 5 then do
      Exit.die "Starting guess must be length 5"
   else do
      let f = args !! 1
      f' <- B.readFile f
      let gSet = (getValidWords . B.words) f'
          k = initKnowledge
          a' = B.pack a
          g = B.pack "raise"
      print g
      playPar g a' k gSet 1
```