Word Search

Anshit Shirish Chaudhari (ac4772)
Soamya Agrawal (sa3881)

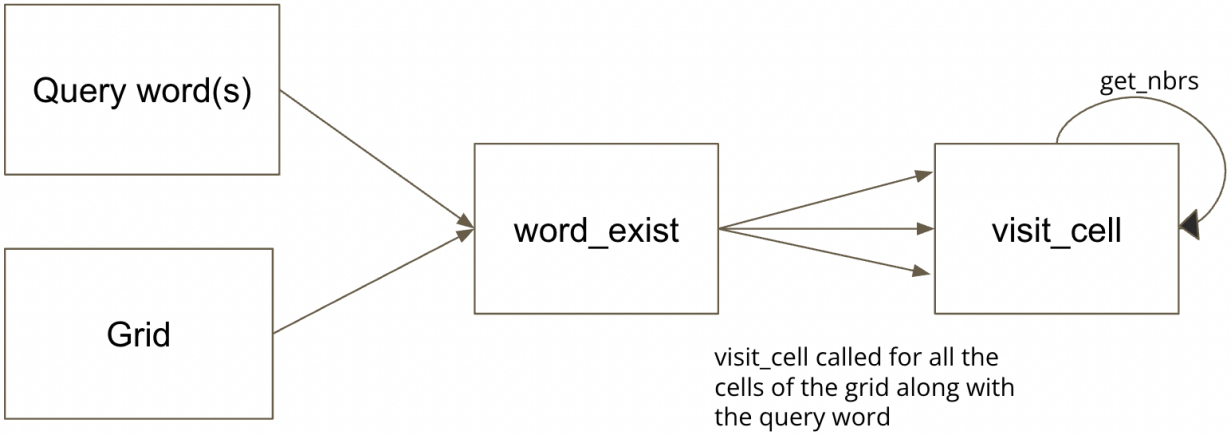
Problem Statement

Finding the word in a matrix is a common game that humans love. The aim of our project is to build a parallel algorithm in Haskell to check if a word is present in the grid of characters. We say a word exists if it can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally, vertically, or diagonally neighboring (similar to a snaking puzzle). The same letter cell cannot be used more than once. Without the loss of generality, we restrict the input grid and the query word to contain only lowercase English letters.

Dictionary-Search (Extension)

After tackling the above challenge, we decided to parallelize the problem of finding top-1000 frequent english words in the input grid. We call this dictionary-search. It takes the input grid and a text-file containing dictionary words (top-1000 frequent english words in this case) and outputs the number of words which are present in the grid.

Algorithm



Following are the components of our algorithm:

- Grid is a m*n matrix of lowercase characters.
- Query word is the word consisting of lowercase characters that we want to search in the grid.
- word_exist method takes query_word and grid as input, and calls visit_cell method for all the cells of the grid.

- visit_cell method takes the query word, grid, and the current cell as input and checks if the initial character of the word is the same as the character present at the cell. In case it matches, then it recursively calls itself, with neighboring cells and the remaining part of the word until there are no more characters left in the word. In case it is not, then it returns False.

Test Inputs

We chose a input grid of 500*500 characters, where lowercase characters (a-z) were randomly generated and saved to a file. We use the process of random generation to reduce the chances of bias towards specific input words.

For testing the word-search algorithm, we chose the input query words “haskell” and “gingera”. “haskell” is present in our input test grid and “gingera” is not present in our input test grid. Since “haskell” is present in the input test grid, the time taken to search it in the grid will be lesser as compared to “gingera”. For the query word, “haskell” the algorithm should return True as soon as the word is found, but for “gingera” it would exhaust all the paths and then return “False”. Due to the different nature of computation for both these queries, we choose these words in our analysis. Note that both the words are of equal length, so it would be fair to test on these two words.

For the dictionary search query, we chose the top 1000 english words from google-10000-english.

Parallelization Strategies

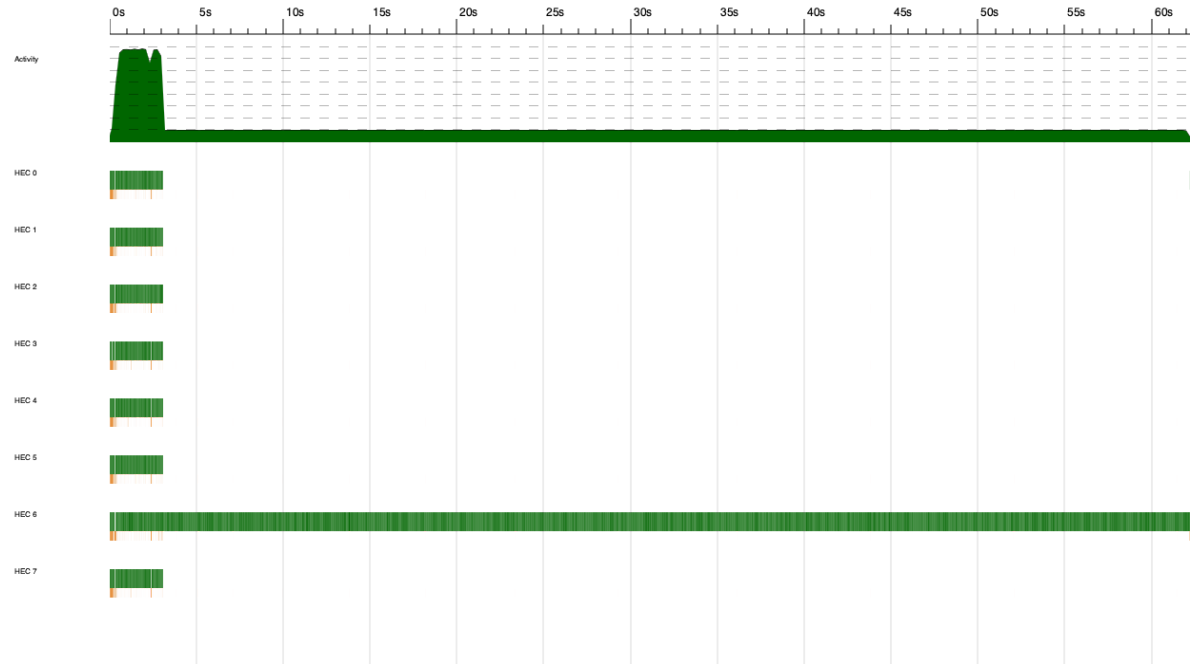
We now discuss the different strategies that we tried out before reaching the optimal solution. For each of the strategies, we evaluate the performance of the algorithm for the test inputs mentioned above.

Word-Search

1. The first step of the algorithm is to call *visit_path* for all the cells. We are essentially visiting all the paths that begin with each cell. We attempted to parallelize these calls because they are independent of each other and we only care about the first call that returns success(true).

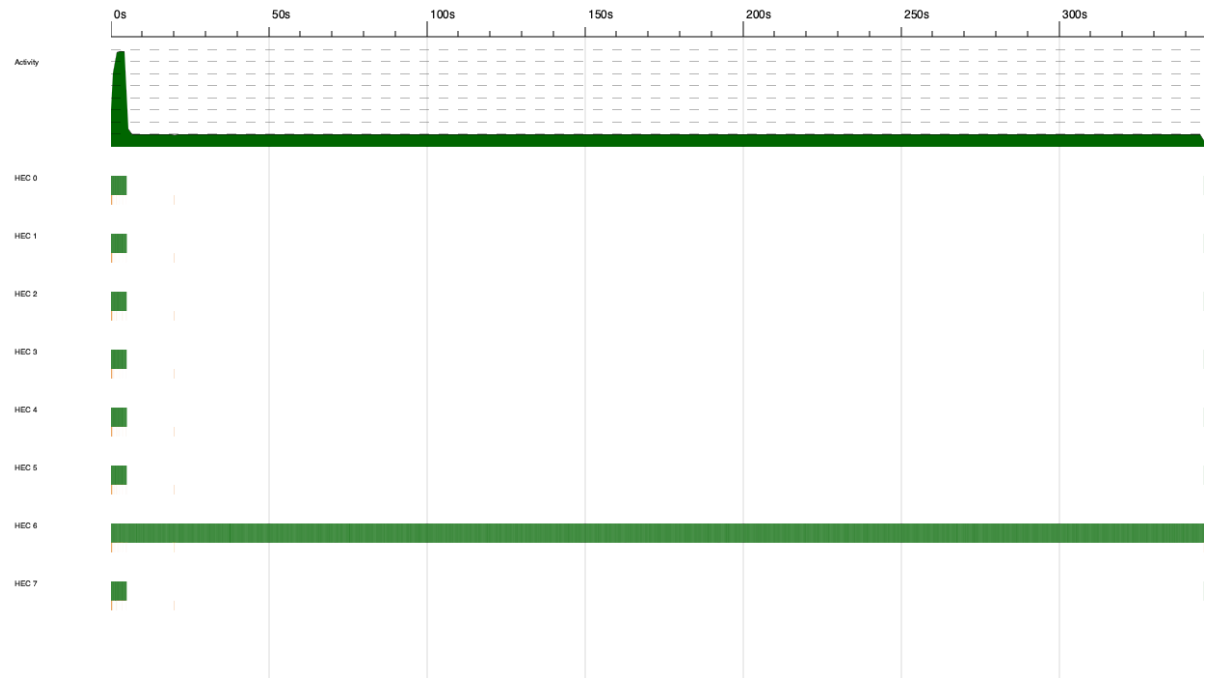
   We use *parList* to visit all the cells in parallel

```
map visit_cell_dash cells `using` parList rseq
```



word-search-exe inputs/large_input_500.txt 500 500 "haskell" +RTS -N8 -s -ls 81.26s user 0.92s system 132% cpu 1:02.22 total

SPARKS: 250000 (18537 converted, 231463 overflowed, 0 dud, 0 GC'd, 0 fizzled)
INIT time 0.001s (0.011s elapsed)
MUT time 79.155s (61.823s elapsed)
GC time 2.097s (0.345s elapsed)
EXIT time 0.000s (0.005s elapsed)
Total time 81.253s (62.184s elapsed)



word-search-exe inputs/large_input_500.txt 500 500 "gingera" +RTS -N8 -s -ls 364.72s user 3.16s system 106% cpu 5:45.71 total

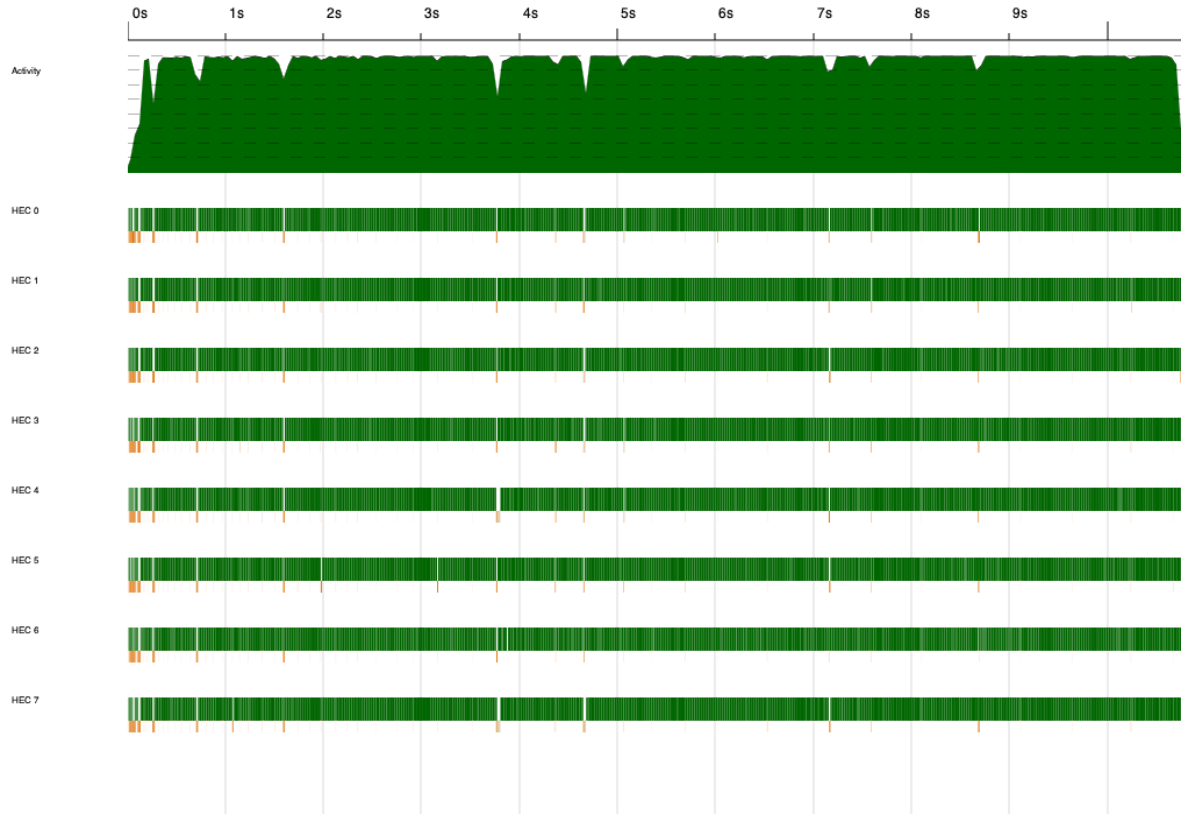
SPARKS: 250000 (17533 converted, 232467 overflowed, 0 dud, 0 GC'd, 0 fizzled)
INIT time 0.001s (0.015s elapsed)
MUT time 363.230s (345.272s elapsed)
GC time 1.483s (0.380s elapsed)
EXIT time 0.001s (0.005s elapsed)
Total time 364.714s (345.673s elapsed)

We see that many m*n sparks are created and most of them are overflowed. If the query word is not present in the matrix, all the sparks will be utilized as we need to search through all the paths. But in other cases, we will only be needing fewer sparks than m*n. We also observe that all the cores are not optimally used.

2. We then move to using parBuffer since it creates limited sparks by regulating the number of outstanding sparks

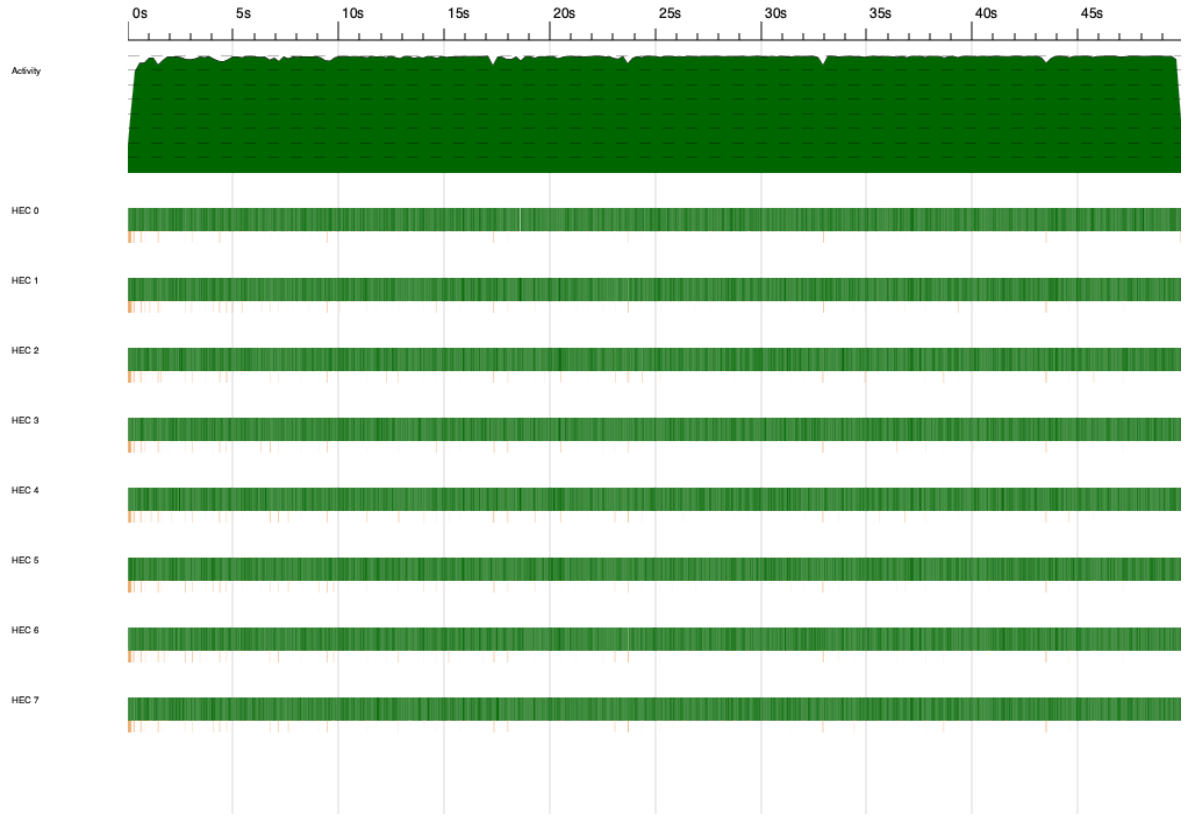
   We use,

```
map visit_cell_dash cells `using` parBuffer 5000 rpar
```



word-search-exe inputs/large_input_500.txt 500 500 "haskell" +RTS -N8 -s -ls 74.37s user 0.64s system 696% cpu 10.772 total

SPARKS: 247835 (178852 converted, 4853 overflowed, 1537 dud, 8815 GC'd, 53778 fizzled)
INIT time 0.001s (0.007s elapsed)
MUT time 72.847s (10.532s elapsed)
GC time 1.482s (0.199s elapsed)
EXIT time 0.042s (0.006s elapsed)
Total time 74.372s (10.744s elapsed)



word-search-exe inputs/large_input_500.txt 500 500 "gingera" +RTS -N8 -s -ls 347.12s user 2.38s system 700% cpu 49.912 total

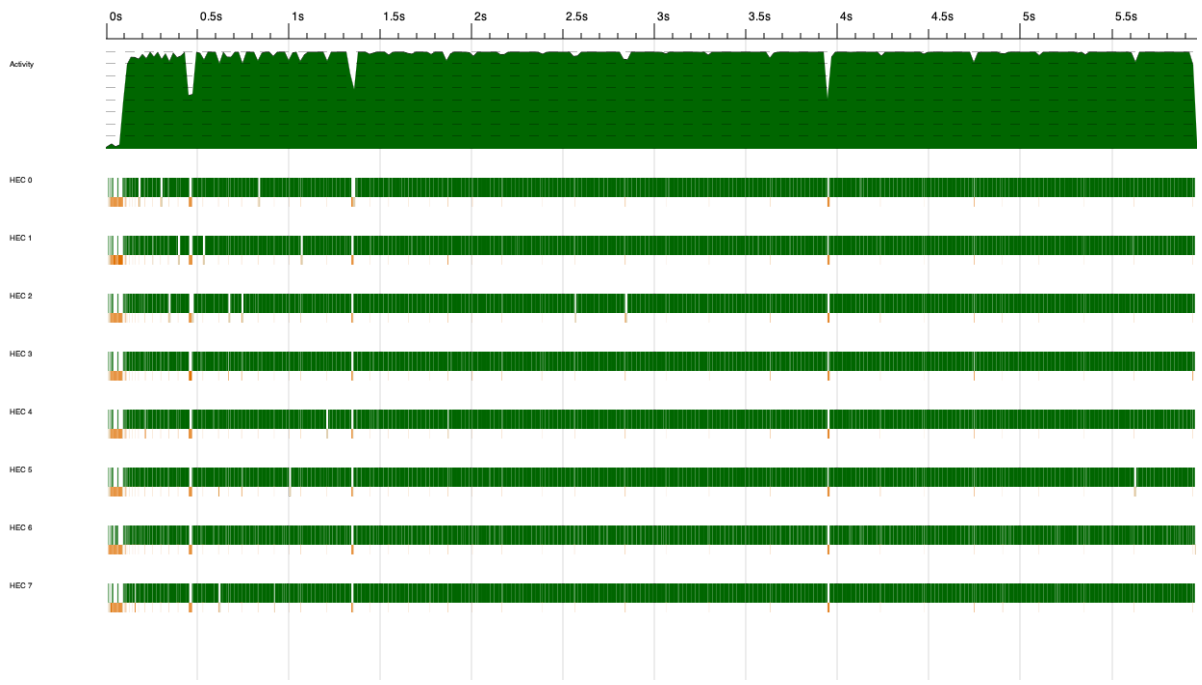
SPARKS: 523351 (305294 converted, 4643 overflowed, 1927 dud, 2040 GC'd, 209447 fizzled)
INIT time 0.001s (0.017s elapsed)
MUT time 345.017s (49.521s elapsed)
GC time 2.098s (0.323s elapsed)
EXIT time 0.000s (0.005s elapsed)
Total time 347.117s (49.866s elapsed)

We saw many sparks getting fizzled. This meant that the main thread computed the result of these sparks and thus the program was not able to make use of extra cores. This means that we need to increase the sequential nature of the program.

3. Instead of using rpar in the command in part 2 above, we tried rseq as the strategy for parBuffer

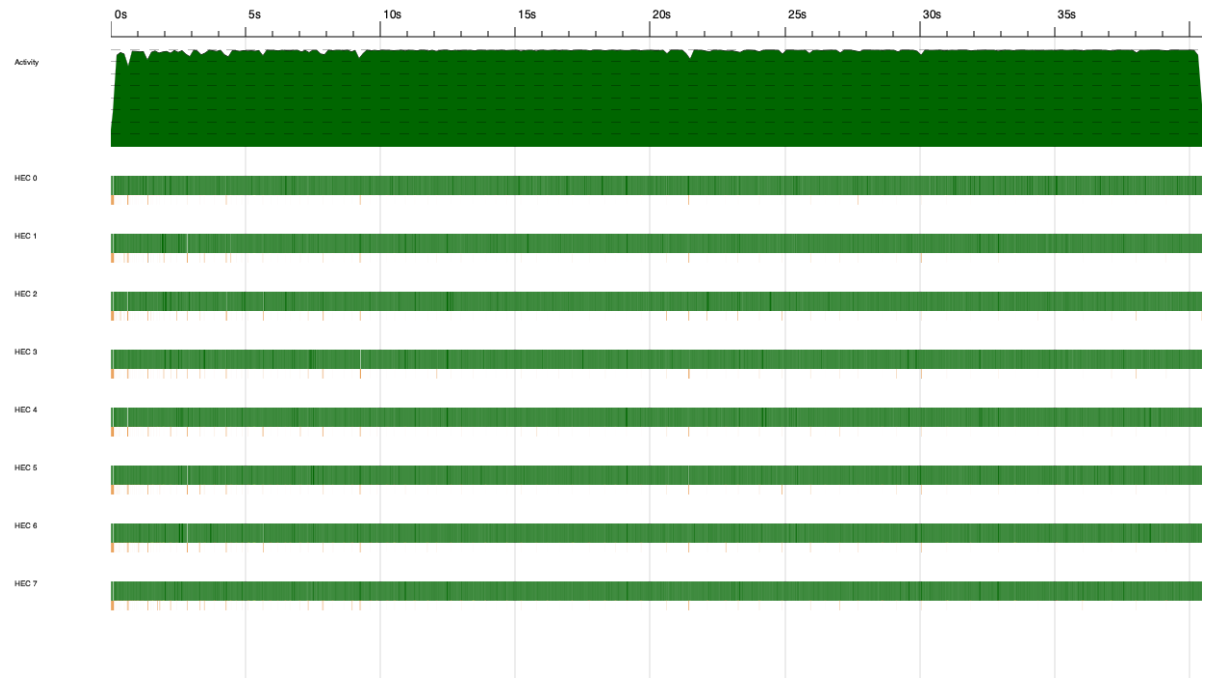
   We use,

```
map visit_cell_dash cells `using` parBuffer 5000 rseq
```



word-search-exe inputs/large_input_500.txt 500 500 "haskell" +RTS -N8 -s -ls 43.99s user 0.62s system 708% cpu 6.293 total

SPARKS: 116683 (111796 converted, 0 overflowed, 0 dud, 4886 GC'd, 1 fizzled)
INIT time 0.001s (0.012s elapsed)
MUT time 43.160s (5.800s elapsed)
GC time 0.815s (0.143s elapsed)
EXIT time 0.012s (0.007s elapsed)
Total time 43.989s (5.962s elapsed)



word-search-exe inputs/large_input_500.txt 500 500 "gingera" +RTS -N8 -s -ls 283.56s user 2.04s system 705% cpu 40.463 total

SPARKS: 250000 (249999 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)
INIT time 0.001s (0.011s elapsed)
MUT time 282.066s (40.183s elapsed)
GC time 1.488s (0.230s elapsed)
EXIT time 0.000s (0.009s elapsed)
Total time 283.556s (40.433s elapsed)

We thus reduce the frizzled sparks to almost 0. This also leads to a decrease in our runtime.

4. So far, we have only parallelised searching on paths with different starting cells. All the child paths are still being evaluated sequentially. We now investigate if we can parallelize child paths.

   The total number of child paths is exponential as there are at most seven (eight neighbours less the previously visited neighbour cell) options to go to from each cell. But depending on the query word, most of the paths get pruned. The nature of the algorithm is to proceed along paths who share prefixes with the input words. Thus, we will not benefit from parallelization.

   To validate our conclusion, we conducted experiments by allocating eight sparks to each child-path exploration. Which meant, each execution of *visit_cell* creates eight sparks, one for each neighbour. This approach led to a severe increase in the number of frizzled sparks and

drastically increased our runtime. This confirmed our hypothesis that due to high pruning on child path level, we do not gain from parallelism.

Dictionary Search

In our example, we use the top-1000 english words as our dictionary. We see that there are three possible avenues for parallelization:

- parallelize iteration on dictionary words, but word-search is executed sequentially
- sequentially iterate through dictionary words, but word-search is executed in parallel
- parallelize dictionary words iteration and word-search

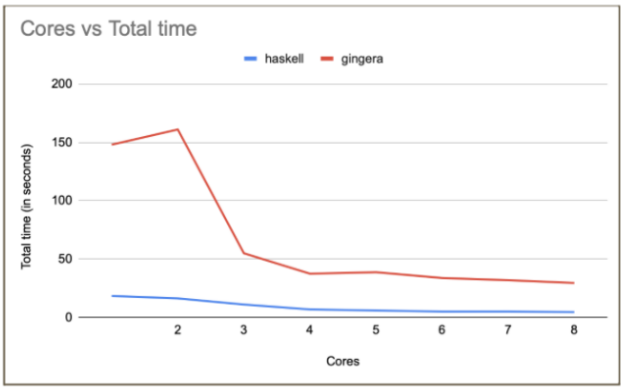
In our experiments, the first two strategies did not perform as good as the last strategy. Hence, we focused on improving the last strategy.

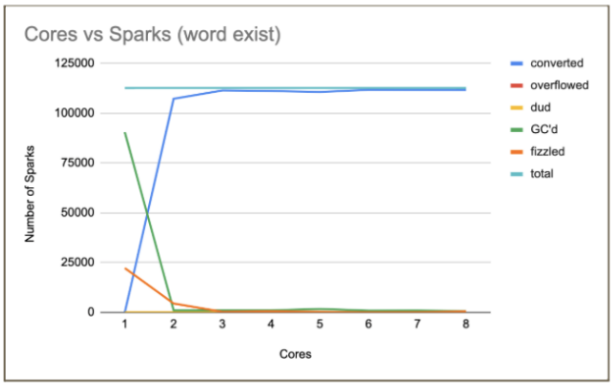
Because of the dual nature of parallelization, we had to conduct experiments with several configurations for the number of parallelization (number of sparks) for dictionary words and word-search. Empirically, we found that using around 10 sparks for dictionary-words and 1000 sparks (we reduced the number of sparks from 5000 in the parallel implementation of word-search discussed in the previous section to 1000 here) for the word-search algorithm gave good results. Giving more sparks to dictionary-words worsened the performance as the word-search algorithm would then not get enough time to execute and thus it would slow down the speed. On the other hand, giving more sparks to the word-search algorithm was also not beneficial as it reduced the parallelization for dictionary-words.

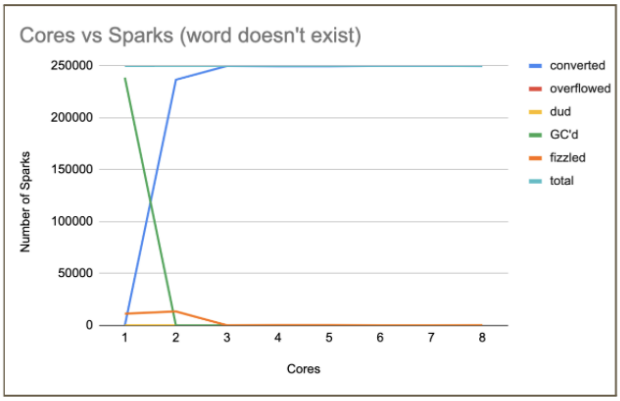
Results

In this section, we analyze the run times and spark metrics for word-search and dictionary-search algorithms.

word-search



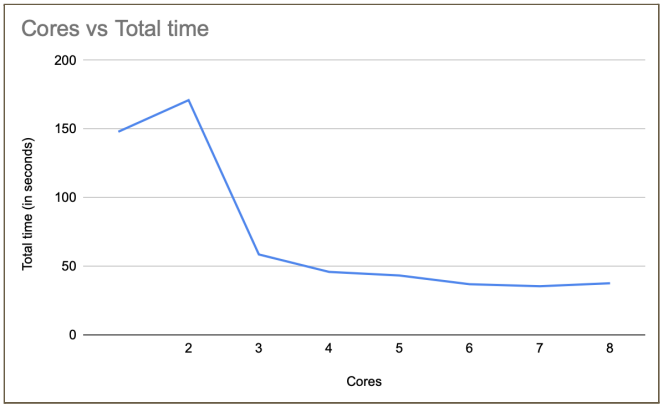


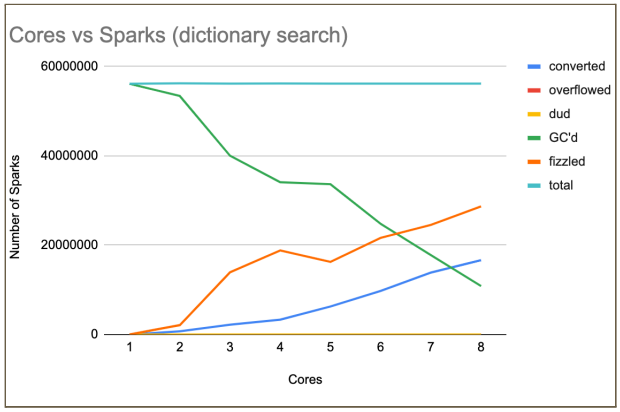


Cores	"haskell"	speedup	"gingera"	speedup
1	18.265	1	148	1
2	16.197	1.127677965	161	0.9192546584
3	10.946	1.668646081	54.871	2.697235334
4	6.728	2.714774078	37.35	3.962516734
5	5.789	3.155121783	38.618	3.832409757
6	4.883	3.740528364	33.652	4.397955545
7	4.91	3.719959267	31.798	4.654380779
8	4.403	4.148307972	29.435	5.028027858

We analyze the runs of word search queries: “haskell” and “gingera”. We see that the runtime for both the queries decrease as the number of cores increase. There is huge improvement in runtime from when we go from 2->3 cores and that can be explained by the dip in garbage collected and fizzled sparks. The rate of decrease of runtime slows down after cores increase from 5, but it still decreases. We see a speedup of 4x for “haskell” query and 5x for “gingera” query.

dictionary-search





Cores	top-1000 words	speedup
1	148	1
2	171	0.865497076
3	58.58	2.526459543
4	45.863	3.227002159
5	43.26	3.421174295
6	36.887	4.012253639
7	35.426	4.177722577
8	37.599	3.936274901

We analyze the run of finding top-1000 english words [1] in the input grid. We observe that the speedup reaches a maximum of 4.1 when the number of cores is 7. As the number of cores increase, we see an increase in the garbage collected sparks and that explains the decrease in speedup beyond 7 cores.

Conclusion

The word-search algorithm achieves a speedup of 4x for words present in the grid and a speedup of 5x for words not present in the matrix. Given the random nature of our input and the query words, a similar speedup could be achieved for other queries as well. The maximum speedup was achieved when we used 8 cores.

The dictionary-search algorithm achieves a speedup of 4x when running on 7 cores. We observe that speedup decreases when the number of cores is 8 due to increase in the number of garbage collected sparks.

References

1. https://github.com/first20hours/google-10000-english
2. https://stackoverflow.com/questions/4978578/how-to-split-a-string-in-haskell
3. https://leetcode.com/problems/word-search/

Code

GitHub Link: https://github.com/anshit-chaudhari/word-search
(This is a private repository at this moment. If you need access, please send a request to Anshit Shirish Chaudhari)

Code Snippet

WordSearch.hs

```
import SearchParallel(word_exist_par)
import InputReader(boolToString, validate, formMatrixFromInput)
import System.Exit(die)
import System.Environment(getArgs)


main :: IO ()
main = do args <- getArgs
      case args of
       [fileName, rows, columns, word] -> do
         contents <- readFile fileName
         if (length word) < 1 then
          die $ "Length of the word should be >= 1"
         else if not (validate word) then
          die $ "word must contain only lowercase English letters"
         else
          putStrLn (boolToString (word_exist_par (formMatrixFromInput contents m n) m n word))
          where
           m = (read rows) :: Int
```

```
            n = (read columns) :: Int
        _ -> do
            die $ "Usage: word_exist <sample_input.txt> <rows> <columns> <word>"
```

DictSearch.hs

```
import DictSearchParallel(dict_search)
import InputReader(formMatrixFromInput)
import System.Exit(die)
import System.Environment(getArgs)

main :: IO ()
main = do args <- getArgs
          case args of
           [fileName, dictFile, rows, columns] -> do
              contents <- readFile fileName
              query_content <- readFile dictFile
              let query_words = words query_content
              let res = dict_search (formMatrixFromInput contents m n) m n query_words
              let present_words = (filter is_present (zip query_words res))
              putStrLn ("#present_words: " ++ (show $ length present_words))
              where
                 m = (read rows) :: Int
                 n = (read columns) :: Int
                 is_present (_,y) = y
           _ -> do
              die $ "Usage: word_exist <sample_input.txt> <rows> <columns> <word>"
```

InputReader.hs

```
module InputReader (boolToString, validate, formMatrixFromInput) where

import Search(word_exist)
import SearchParallel(word_exist_par)
import Data.Array
import Data.Char
import System.Exit(die)
import System.Environment(getArgs)
import Control.Applicative
```

```
-- Convert boolean to String
boolToString :: Bool -> String
boolToString True = "True"
boolToString False = "False"

wordsWhen :: String -> [Char]
wordsWhen s =  case dropWhile isDelimiter s of
                "" -> []
                s' -> (head w) : wordsWhen s''
                    where
                      (w, s'') = break isDelimiter s'
               where
                isDelimiter = (liftA2 (||) (==',') isSpace)


-- Check if string has all lowercase letters
validate :: String -> Bool
validate s = all isLower s


-- Forms grid of characters
formMatrixFromInput:: String -> Int -> Int -> Array (Int,Int) Char
formMatrixFromInput contents rows columns = array ((1,1), (rows, columns)) ([ ((i,j), list !!
(columns*(i-1) + (j-1))) | i <- [1..rows], j <- [1..columns]])
                        where
                         list = wordsWhen contents
```

Search.hs

```
module Search (word_exist) where

import Data.Array
import qualified Data.Set as SetDash

get_nbrs :: Int -> Int -> [(Int, Int)]
get_nbrs i j = [(i-1,j), (i,j+1), (i+1,j), (i,j-1), (i-1,j-1), (i+1,j+1),(i+1,j-1),(i-1,j+1)]
```

```
visit_cell :: Array (Int, Int) Char -> SetDash.Set(Int, Int) -> Int -> Int -> [Char] -> (Int,Int) -> Bool
visit_cell _ _ _ _ [] _  = True
visit_cell matrix visited m n (x:xs) (i,j) | i < 1 || i > m = False
                           | j < 1 || j > n = False
                           | SetDash.member (i,j) visited = False
                           | x /= (matrix ! (i,j)) = False
                           | otherwise = any (==True) children_visits
                           where
                             visited_dash = SetDash.insert (i,j) visited
                             visit_cell_dash = visit_cell matrix visited_dash m n xs
                             children_visits = map visit_cell_dash (get_nbrs i j)

word_exist :: Array (Int, Int) Char -> Int -> Int -> [Char] -> Bool
word_exist _ _ _ []    = True
word_exist matrix m n (x:xs) = any (==True) valid_visits
                    where
                      cells = [(i,j) | i <- [1..m], j <- [1..n]]
                      visit_cell_dash = visit_cell matrix SetDash.empty m n (x:xs)
                      valid_visits = map visit_cell_dash cells
```

SearchParallel.hs

```
module SearchParallel (word_exist_par) where

import Data.Array
import qualified Data.Set as SetDash
import Control.Parallel.Strategies
  ( parBuffer, using, rseq )

get_nbrs :: Int -> Int -> [(Int, Int)]
get_nbrs i j = [(i-1,j), (i,j+1), (i+1,j), (i,j-1), (i-1,j-1), (i+1,j+1),(i+1,j-1),(i-1,j+1)]

visit_cell :: Array (Int, Int) Char -> SetDash.Set(Int, Int) -> Int -> Int -> [Char] -> (Int,Int) -> Bool
visit_cell _ _ _ _ [] _  = True
visit_cell matrix visited m n (x:xs) (i,j) | i < 1 || i > m = False
                           | j < 1 || j > n = False
                           | SetDash.member (i,j) visited = False
```

```
                        | x /= (matrix ! (i,j)) = False
                        | otherwise = any (==True) children_visits
                        where
                          visited_dash = SetDash.insert (i,j) visited
                          visit_cell_dash = visit_cell matrix visited_dash m n xs
                          children_visits = map visit_cell_dash (get_nbrs i j)

word_exist_par :: Array (Int, Int) Char -> Int -> Int -> [Char] -> Bool
word_exist_par _ _ _ []     = True
word_exist_par matrix m n (x:xs) = any (==True) valid_visits
                    where
                      cells = [(i,j) | i <- [1..m], j <- [1..n]]
                      visit_cell_dash = visit_cell matrix SetDash.empty m n (x:xs)
                      valid_visits = map visit_cell_dash cells `using` parBuffer 1000 rseq
```

DictSearchParallel.hs

```
module DictSearchParallel (dict_search) where

import SearchParallel(word_exist_par)
import Data.Array
import Control.Parallel.Strategies
   ( parBuffer, using, rseq )

dict_search :: Array (Int, Int) Char -> Int -> Int -> [String] -> [Bool]
dict_search _ _ _ [] = []
dict_search matrix m n query_words = map (word_exist_par matrix m n) query_words `using`
parBuffer 10 rseq
```