COMS 4995 Parallel Functional Programming
Project Proposal-Word AutoComplete

Shengkai Li (sl4685), Wenqian Yan (wy2249)

November 23, 2020

1 Introduction

Word AutoComplete, or Word AutoSuggestion is a feature in which an application predicts the rest of a word a user is typing [1]. It is most commonly used in search engines, like Google, to suggest queries when users begin typing the first few letters of their search string.

These auto-suggestions should be as responsive as possible: they need to show up on the screen before users finish typing, otherwise the suggestion becomes pointless and useless. Hence, the speed of Word AutoSuggestion is crucial, which brought us to this project: speeding up word auto-suggestion with parallelism in Haskell.

We divided our work into three main steps: Word Cleanup, Word/N-Grams Count, and Word/N-Grams AutoSuggestion. At each step, we compare the performance of sequential implementation and parallel implementation to see how much we could speed up. Also, we analyze further on garbage collection and number of chunks running in parallel to optimize our parallel implementation. In summary, the performance of parallel implementation has been significantly improved, which is ***300 times*** faster than pure serial implementation.

2 Word Cleanup

Given a large enough text file as the dictionary of suggestion, we want to clean it up by discarding all non-alphabetic characters aside from whitespace and treating what's left as lowercase, and finally producing a list of cleaned words. The fully sequential implementation of word cleanup is shown below.

2.1 Sequential Word Cleanup

1. Map all characters to lowercase
2. Filter out words that are non-alphabetic

```
wordFilter :: [[Char]] -> [[Char]]
wordFilter lines =  map ((map toLower) . filter (\x -> isAlpha x || isSpace x)) lines
```

As shown in Figure 1, all the work of Word Cleanup was done on a single processor. The total runtime was 51.04s while 6.5% of that time was spent on garbage collection. This implementations' result should be the baseline for the parallel implementations to speed up.

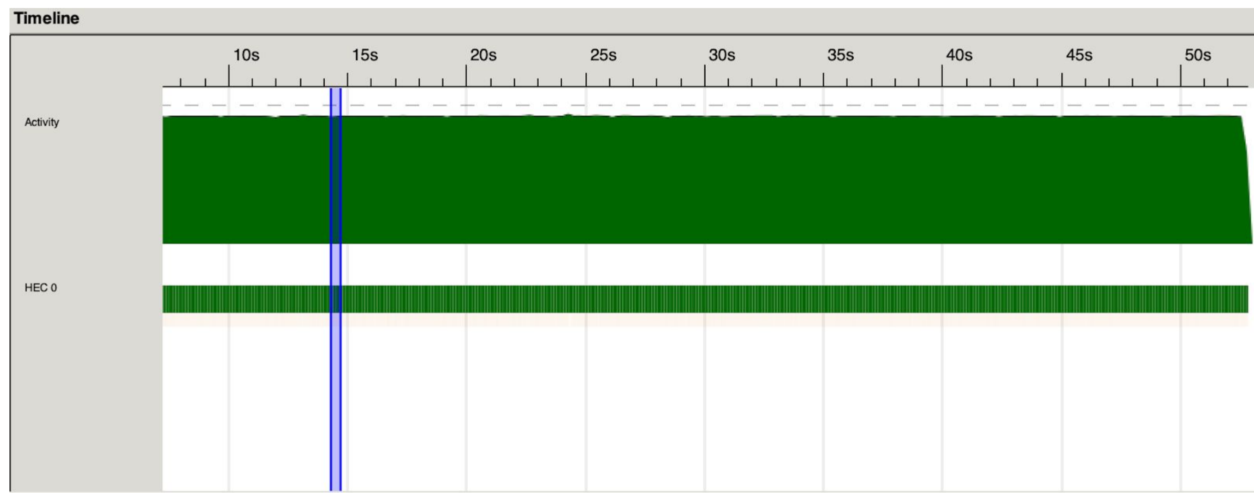

Figure 1: Timeline of Word Cleanup on a single processor

2.2 *parList* Parallel Word Cleanup

```
wordFilter :: [[Char]] -> [[Char]]
wordFilter lines =  map ((map toLower) . filter (\x -> isAlpha x || isSpace x)) lines
`using` parList rdeepseq
```

This is the first attempt at parallel implementing Word cleanup. Since the Word Cleanup is supposed to deal with phenomenally large datasets, this attempt tries to simultaneously clean up each line of that datasets. We run our implementations on a Unbuntu virtual machine with 1, 2, 4, 8 processor.

Cores	Time(s)	Speedup
1	160.99	0.37
2	116.44	0.43
4	82.42	0.61
8	86.55	0.589

Table 1: *parList* Parallel Performance

As shown in Table1, the parList parallel implementation does not speed up Word Cleanup at all but even result in worse performance. This is because it spent most of the time on garbage collection, about 50% of total run time, and "*parList* Strategy forces the whole spine of the list, preventing the program from streaming in constant space[2]".

2.3 *parBuffer* Parallel Word Cleanup

```
wordFilter :: [[Char]] -> [[Char]]
wordFilter lines =  map ((map toLower) . filter (\x -> isAlpha x || isSpace x)) lines
`using` parBuffer 4 rdeepseq
```

The parBuffer Parallel implementation is the second and real implementation. Since we don't want to force our program to load the entire file (because of memory consumption) and generate all the sparks at the beginning (because of spark pool overflow), the *parBuffer* strategy is the best choice even though we have to specify a particular value of buffer size[3].

As shown in Table2 and Figure 2, we successfully speedup our Word Cleanup and four processors have balance work.

Cores	Time(s)	Speedup
1	57.53	0.887
2	43.35	1.117
4	37.48	1.541
8	41.85	1.068

Table 2: *parBuffer* Parallel Performance

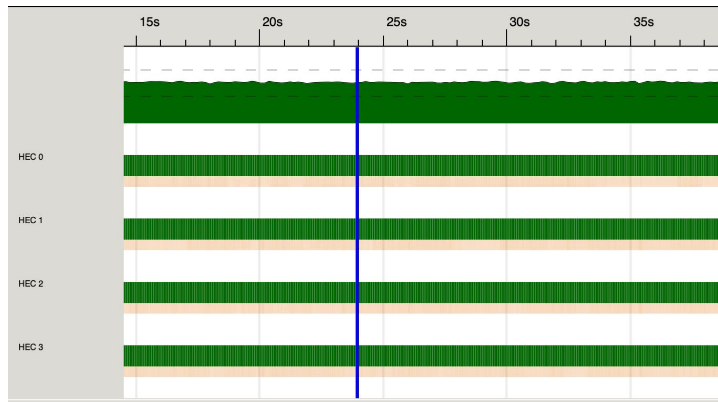

Figure 2: Timeline of events for parBuffer implementation using four cores

3 Word Count

Given the large cleaned words list, we want to generate (word, frequency) pairs to help us provide top N word suggestions based on frequency in the future.
An example of ***"Word Count***" result: [("haskell", 4995), ("plt", 4115)]

3.1 Sequential word count

1. Iterate through the word list to map each word to (word, 1)
2. Merge entries with the same word to (word, frequency) list

```
wordMapper :: [String] -> [(String, Int)]
wordMapper w = map (\x -> (x, 1)) w

wordReducer :: (Ord k, Num a) => [(k, a)] -> [(k, a)]
wordReducer l =  M.toList $ M.fromListWith (+) l
```

We tested our wordMapper sequential implementation on a 40MB text file to generate word count (word, frequency) pair list.

```
$ ./sequential ../../test/test.txt +RTS -N1  -s -ls

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

  INIT    time    0.001s  (  0.001s elapsed)
  MUT     time    8.177s  (  8.727s elapsed)
  GC      time    1.118s  (  1.132s elapsed)
  EXIT    time    0.001s  (  0.001s elapsed)
  Total   time    9.297s  (  9.861s elapsed)
```

```
Alloc rate    3,160,261,895 bytes per MUT second

Productivity  88.0% of total user, 88.5% of total elapsed
```

The total runtime to run wordMapper is 9.861s on a single processor. It’s not terrible with 11.5% of time spent on GC but there's still space to accelerate it. The complete sequential implementation on this 40MB text file is 22.681s and it takes wordReducer 12.304s running on a single processor, which is the benchmark for our parallelled wordReducer.

```
reduce takes12.304750443 seconds.

 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  (  0.001s elapsed)
 MUT     time   12.589s  ( 12.748s elapsed)
 GC      time    9.200s  (  9.927s elapsed)
 EXIT    time    0.000s  (  0.005s elapsed)
 Total   time   21.790s  ( 22.681s elapsed)
 Alloc rate    1,286,410,934 bytes per MUT second
 Productivity  57.8% of total user, 56.2% of total elapsed
```

3.2 mapper in parallel

3.2.1 Speed up wordMapper with parList rdeepseq

wordMapper, in the sequential implementation above, maps each word w in the list to (w,1) for future accumulation in reducer. The first intuition for speeding up word count is to run this map job parallelly for each list element. With this idea, we implemented a parallel function called ***`parWordMapper`*** by giving wordMapper, the map function in serial, a parallel strategy: ***parMap rdeepseq***.

parMap rdeepseq is to evaluate each list element in parallel with strategy fully evaluating then proceeding. With this strategy, we are able to generate (w,1) pairs parallely on each list element rather than slowly walking through the whole file one by one.

```
wordMapper :: [String] -> [(String, Int)]
wordMapper w = map (\x -> (x, 1)) w

parWordMapper :: [String] -> [(String, Int)]
parWordMapper w = wordMapper w `using` parList rdeepseq
```

Here is the output to run our sequential and parallel implementations on a Ubuntu virtual machine with 1, 2, 4, 6 processors over 9.861s by sequential wordMapper.

N	R1	R2	R3	Average
1 2 4 6	10.751s 12.861s 14.232s 15.794s	10.721s 13.171s 15.172s 16.613s	10.841s 13.611s 14.992s 16.422s	10.771s 13.214s 14.799s 16.276s

Table 3 Performance of parWordMapper with *parMap rdeepseq*

As table 3 shows, *parMap rdeepseq* does not speed up Word Count and all the tests take longer than 9.861s by sequential wordMapper. Thus, it is not the proper strategy for wordMapper.

3.2.2 Speed up wordMapper with parBuffer rdeepseq

Same with *WordCleanup*, parBuffer allows us not to load the entire list in, which creates overheads, and behaves like a circular buffer with a constant buffer size, consuming input and moving to the next. It's useful here as it does not need to know the entire list and just step over each element to map word to (word,1) tuple.

```
bufferSize :: Int
bufferSize = 4

wordMapper :: [String] -> [(String, Int)]
wordMapper w = map (\x -> (x, 1)) w

parWordMapper :: [String] -> [(String, Int)]
parWordMapper w = wordMapper w `using` parBuffer bufferSize rdeepseq
```

To find the proper buffer size for parBuffer, we ran tests with -N2 on test file (40MB), and the results in Table 4 compares them with 9.861s by sequential wordMapper and shows that it's the most optimal when `parBuffer 4 rdeepseq`.

BufferSize	Run time	Speed up
1 2 4 6 8 10	14.702s 10.822s 6.642s 6.882s 6.891s 6.942s	0.671 0.911 1.484 1.432 1.430 1.332

Table 4 *parBuffer rdeepseq* with different buffer size (+RTS -N2 -s -ls)

With the optimal buffer size 4 found, we continue to test the performance with machines with 1, 2, 4, 6 processors and their speed up to see which N is the most optimal one.

N	R1	R2	R3	Average	Avg Speed up
1	5.042s	5.381s	5.001s	5.141s	1.909
2	6.642s	6.712s	7.022s	6.882s	1.426
4	6.912s	7.323s	7.753s	7.329s	1.339
6	7.894s	7.813s	7.714s	7.807s	1.263

Table 5 Performance of parWordMapper with *parBuffer rdeepseq*

As table5 shows, *parBuffer rdeepseq* can at most speed up Word Count by twice when N=1, compared with 9.861s by the sequential implementation.

3.3 reducer in parallel

3.3.1 Speed up wordReducer with parList rdeepseq

wordReducer, in the sequential implementation, adds all (w,1) together for the same word w to get the final frequency count for each word w. The intuition for speeding up this part is to run this map job parallelly for each list element. So we implemented a parallel function called ***`parWordReducer`*** by giving wordReducer the parallel strategy: parList rdeepseq. To make it more efficient, we used *chunksOf* to split the large list into several chunks, each of size equals chunkSize, and then map wordReducer on each chunk to run parallely, and at last it concat results of all the chunks as output.

```
import Data.List.Split( chunksOf )

chunkSize :: Int
chunkSize = 100

wordReducer :: (Ord k, Num a) => [(k, a)] -> [(k, a)]
wordReducer l =  M.toList $ M.fromListWith (+) l

parWordReducer :: (Ord k, Num a, NFData k, NFData a) => [(k, a)] -> [(k, a)]
parWordReducer l = wordReducer $ concat ((map wordReducer l') `using` parList rdeepseq)
                    where l' = chunksOf chunkSize l
```

To find the proper chunk size to split the list, we ran tests with -N2 on test file (40MB), and the results in Table 6 compares them with 12.304s by sequential wordReducer and shows that the most optimal acceleration is 1.436 times faster when chunkSize=100.

ChunkSize	Run time	Speed up
20	9.280s	1.325
50	9.565s	1.286
100	8.571s	1.436
500	9.097s	1.352
1000	10.868s	1.132

Table 4 *parList rdeepseq* with different chunk size (+RTS -N2 -s -ls)

4 N-Grams Count

Given the large cleaned list, we want to generate (n-grams, frequency) pairs to provide top N phrase suggestions based on frequency in the dictionary.
An example of ‘***3-grams Count***’ result:
[(“haskell is great”, 10), (“best programming language”, 4995)]

4.1 Sequential n-grams count

```
mapper :: Int -> [String] -> [(String, Int)]
mapper n str
   | n <= length str = (unwords ngram, 1::Int) : mapper n str'
   | otherwise = []
   where ngram = take n str
         str'  = drop (1::Int) str

reducer :: (Ord k, Num a) => [(k, a)] -> [(k, a)]
reducer l = toList $ fromListWith (\num1 num2 -> num1 + num2) l
```

Sequential Implementation:

1. Break into a list of cleaned n-grams
2. Iterate through the word list to map each word to (ngram, 1)
3. Merge entries with the same word to (word, frequency) list

Because of our fully serial implementation, generating pairs of n-grams and frequency is significantly time/space consuming. Therefore, we switched to a smaller test dataset.

We tested our sequential implementation on a 1MB text file to generate two-grams.. As shown in Figure 3, the total runtime is 93.985s, with 0.24s on garbage collection, on a single processor.

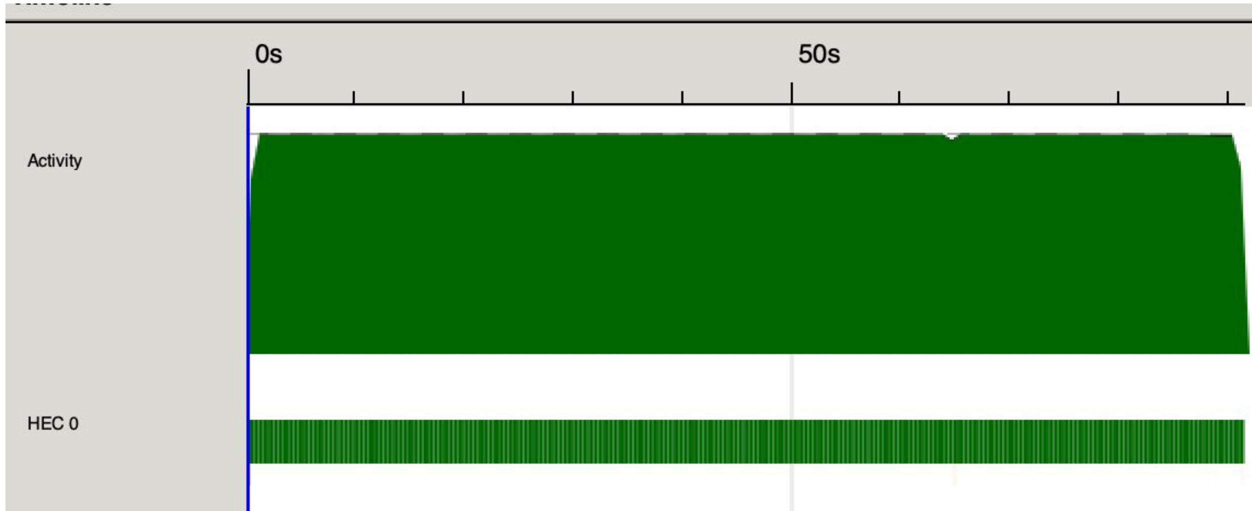

Figure 3: Timeline of events for sequential n-grams count

To accelerate the N-grams Count, we will focus on parallel implementation of both *mapper* and *reducer.*

4.1 reducer in parallel

```
par_reducer :: (Ord k, Num a, NFData k, NFData a) => [(k, a)] -> [(k, a)]
par_reducer l = reducer $ concat
              ((map reducer l') `using` parList rdeepseq)
              where l' = chunksOf 9 l
```

Based on our implementation of reducer, we implemented a function called `par_reducer` to help us run *reducer* in parallel. To achieve parallelism, *par_reducer* splits the input into several chunks, simultaneously calls *reducers* on each chunk, and then computes the final result of each chunk.

The core of *par_reducer* is to find out exactly how many chunks we should split. We did some experiments to explore the answer.

Chunks	Runtime	Garbage Collection
2	73.429	42
4	75.121	47
9	69.192	42
32	81.334	62
100	88.040	34

Table 5: Running on a 4-cores machine

The result in Table 3 is what we have expected. Generating too many sub-chunks makes our performance worse. This is because our *par_reducer* is not pure parallel. In the last step of *par_reducer,* it needs to call *reducer* to sum up all the results from sub-chunks, which is sequential work. The more sub chunks means the more sequential jobs the *par_reducer* needs to finish.

As a result, by running only the reducer in parallel, the best performance we could get is 69.120s. Compared with pure sequential implementation, we have successfully accelerated *25* seconds.

4.2 mapper in parallel

```
par_mapper :: Int -> [[String]] -> [(String, Int)]
par_mapper n l = concat (map (mapper n) l) `using` parList rdeepseq
```

Mapper in the sequential implementation did most of the duty work. To accelerate the n-grams count, it is crucial to run mapper in a good parallel mode. Based on our implementation of mapper, we implemented a parallel function called `par_mapper`. Instead of generating n-grams by keeping iterating the whole file, `par_mapper` simultaneously calls `mapper` on each line and then uses concat to group together the final result

4.2.1 par_mapper's drawback

First, we have to point out that there is potential drawback in par_mapper: It may miss some n grams. Consider the following example:

Input: “Hello world\nHaskell No”
Under The two-grams mode:
The expect output should be: (Hello world, 1), (word Haskell, 1), (Haskell No, 1)
The actually output will be: (Hello world, 1), (Haskell NO, 1)

This is because the nature of *par_mappe*r is to focus only on line and the connection words between two sentences will be discarded.

4.2.2 par_mapper evaluation

However, this potential drawback won't influence our functionality of Word AutoComplete. First, as long as the dataset is large enough, we can still give back the accurate suggestion. Second, *par_mapper* **significantly** improved the performance of n-grams count.

We run our implementations on a Unbuntu virtual machine with 1, 2, 4, 8 processor.

Cores	Time(s)	Speedup
1	0.599	155
2	0.426	218
4	0.428	217
8	0.463	200

Table 6

Combined *par_reducer* and *par_mapper* together (though par_mapper did most of the work), our parallel implementation is almost ***220*** times faster than pure sequential implementation. As shown Figure 4, all four processors work balanced between [0s, 0.25s]. And, as expected, between[0.25, 0.43], we need one single processor to sequentially generate the final result.

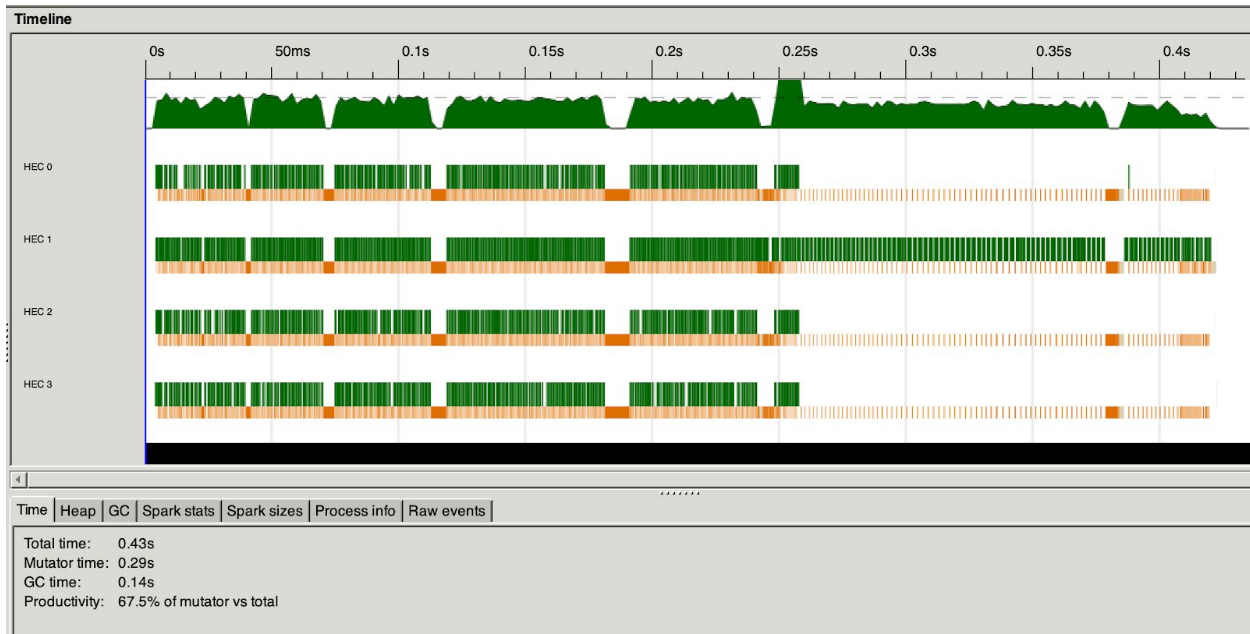

Figure 4: Timeline of events for parallel n-grams count

5 N-Grams/Word AutoComplete

This is the last step. Based on the result from Clean up and Word/N-Grams Count, we move further to implement our own version of **Word/N-Grams AutoComplete**. It accepts a word from the user's input and returns some `suggestion` to the user.

5.1 Sequential Implementation

- Filter along (word, frequency) list for words with matched prefix.
- Sort the filtered list by frequency.
- Find top N suggestions by taking head N of the list after filtering and sorting.

```
ngram_filer :: Eq a => [a] -> [([a], b)] -> [([a], b)]
ngram_filer str l = filter (\(x,_)->str `isPrefixOf` x) l

ngram_sort :: [(a, Int)] -> [(a, Int)]
ngram_sort = sortBy (\(_,x) (_, y) -> compare x (y::Int))
```

5.2 Parallel Implementation

We parallelized step1 to speed up searching lists. Similar to our parallel implementation of N-grams Count, we implemented a function called `par_ngram_filer` to help us run *filters* in parallel. To achieve parallelism, *par_suggestion_filer* splits the input into 9 chunks, simultaneously calls *filters* on each chunk, and then computes the final result of each chunk.

```
par_suggestion_filer :: (Eq a, NFData a, NFData b) => [a] -> [([a], b)] -> [([a], b)]
par_suggestion_filer str l = suggestion_filer str $ concat
                        ((map (suggestion_filer str) l') `using` parList rdeepseq)
                        where l' = chunksOf 9 l
```

5.3 Evaluation

We tested Parallel/Sequential implementation with a different size of test text file, on 4 core machine.

File Size (mb)	Sequential(s)	Parallel(s)	Speed Up
1	1.02	0.92	1.11
16	14.23	12.44	1.15
32	29.46	24.98	1.18
80	74.09	63.15	1.173

Table 7 Performance analysis of suggestion on Parallel/Sequential implementation (-N4)

As shown in Table 7, as the test file size becomes larger, the performance difference between Sequential and Parallel implementations becomes more clear.

6 Summary

Combining all the steps together, we can compare the total performance of word-auto completion by the factors of file size and cores running on. Below are two tables running one complete round on word completion and ngram completion without loop. (We only recorded the time for running sequential implementation on 1 MB file because it takes forever for larger files, while running parallel can reduce the time within one minute.)

File Size (mb)	Sequential	-N1	-N2	-N3	-N4	max speed up
1 16 32	1.011s 21.351s 43.371s	1.140s 14.911s 30.033s	1.311s 19.442s 42.255s	2.011s 20.232s 43.165s	2.131s 20.553s 22.007s	- 1.432 1.444

Table 7 Performance analysis of one round word-auto completion program

File Size (mb)	Sequential	-N1	-N2	-N3	-N4	max speed up
1 16 32	864.311s	3.061s 61.821s 69.831s	2.771s 48.611s 63.682s	2.891s 31.512s 63.622s	2.952s 33.432s 64.644s	311.9

Table 8 Performance analysis of one round ngram completion program

From the results, we can conclude that our parallel implementation is much more efficient both than the sequential one. We observed that for word-auto completion, the performance is best when N=1, and for ngram-autocomplete, the performance is best when N=3. This result is reasonable as the simple sequential implementation of word-auto completion does not take much time, while ngram is very time consuming to need more cores to get enough parallelization.

Also, from the table below, if there are too many cores running on, it may create too many unused or fizzled sparks and get wasted, while if the core is too few, it does not get enough parallelization which is also inefficient.

N	converted	overflowed	GC'd	fizzled
1 2 3 4	0 2848956 2937228 2998431	3278375 285891 292498 180452	878 128916 25268 80260	16388 31878 40647 36498

Table 9 spark stats for parallelization on 16MB test file

A Code Listing

main.hs

```
import NgramCount ( parNgramMapper, parNgramReducer )
import WordCount ( parWordMapper, parWordReducer )
import WordClean ( wordClean )
import Suggestion_Parallel ( par_suggestion_filer, suggestion_sort, suggestion_filer )
import System.Environment ( getArgs )
import Control.Parallel ( pseq )

enterLoop :: [([Char], Int)] -> IO b
enterLoop output = do
    putStrLn ""
    putStrLn "Please enter prefix of word(s):"
    str <- getLine
    putStrLn "Enter the number of top suggestions you want:"
    k <- getLine
    putStrLn "Here are the suggestions:"
    let results = take (read k::Int) $ suggestion_sort $ par_suggestion_filer str output
    mapM_ (\(a,_) -> putStrLn a) results
    enterLoop output

processMapReduce :: Int -> String -> [(String, Int)]
processMapReduce n content
    | n==1  = parWordReducer $ parWordMapper $ words $ unwords $ lines content
    | otherwise = parNgramReducer $ parNgramMapper n $ map words $ lines content

main :: IO ()
main = do
    [filename] <- getArgs
    putStrLn "Enter the number of words you want suggestion for (1 for a single word):"
    n <- getLine
    putStrLn "Loading Dictionary ..."
    content <- wordClean filename
    let output = processMapReduce (read n::Int) content
    writeFile "output.txt" (show output)
    putStrLn "Done loading!"
    enterLoop output
    --print $ head $ suggestion_sort $ par_suggestion_filer str output
    --print $ head $ suggestion_sort $ suggestion_filer str output
```

WordClean.hs

```
module WordClean
(
    wordFilter,
```

```
    wordClean
  ) where

import Data.Char ( isSpace, isAlpha, toLower )
import System.IO ( openFile, hGetContents, IOMode(ReadMode) )
import Control.Parallel.Strategies ( parBuffer, rdeepseq, using )

wordFilter :: [[Char]] -> [[Char]]
wordFilter lines =  map ((map toLower) . filter (\x -> isAlpha x || isSpace x)) lines `using` parBuffer
4 rdeepseq

wordClean :: FilePath -> IO String
wordClean input = do
    handle <- openFile input ReadMode
    contents <- fmap lines $ hGetContents handle
    let output = wordFilter contents
    return $ unlines output
    --writeFile "cleanOutput.txt" (unlines output)
```

WordCount.hs

```
module WordCount where
import WordClean(wordClean)
import qualified Data.Map as M
import Data.List.Split( chunksOf )
import Control.Parallel ( pseq )
import Control.Parallel.Strategies
    ( parList, rdeepseq, using, NFData )

chunkSize :: Int
chunkSize = 20

wordMapper :: [String] -> [(String, Int)]
wordMapper w = map (\x -> (x, 1)) w

parWordMapper :: [String] -> [(String, Int)]
parWordMapper w = concat ((map wordMapper w') `using` parList rdeepseq)
                where w' = chunksOf chunkSize w

wordReducer :: (Ord k, Num a) => [(k, a)] -> [(k, a)]
wordReducer l =  M.toList $ M.fromListWith (+) l

parWordReducer :: (Ord k, Num a, NFData k, NFData a) => [(k, a)] -> [(k, a)]
parWordReducer l = wordReducer l `using` parList rdeepseq
```

NgramCount.hs

```
module NgramCount where
import Data.Map (fromListWith, toList)
--import System.Environment(getArgs)
--import WordClean(wordClean)
```

```haskell
import Data.List.Split ( chunksOf )
import Control.Parallel.Strategies
    ( parList, rdeepseq, using, NFData )



ngramMapper :: Int -> [String] -> [(String, Int)]
ngramMapper n str
    | n <= length str = (unwords ngram, 1::Int) : ngramMapper n str'
    | otherwise = []
    where ngram = take n str
          str'  = drop (1::Int) str

parNgramMapper :: Int -> [[String]] -> [(String, Int)]
parNgramMapper n l = concat (map (ngramMapper n) l) `using` parList rdeepseq

ngramReducer :: (Ord k, Num a) => [(k, a)] -> [(k, a)]
ngramReducer l = toList $ fromListWith (\num1 num2 -> num1 + num2) l

parNgramReducer :: (Ord k, Num a, NFData k, NFData a) => [(k, a)] -> [(k, a)]
parNgramReducer l = ngramReducer $ concat
                ((map ngramReducer l') `using` parList rdeepseq)
                where l' = chunksOf 9 l
```

Suggestion.hs

```haskell
module Suggestion_Parallel where
import Data.List ( isPrefixOf, sortBy )
import Data.List.Split ( chunksOf )
import Control.Parallel.Strategies
    ( parList, rdeepseq, using, NFData )

suggestion_filer :: Eq a => [a] -> [([a], b)] -> [([a], b)]
suggestion_filer str l = filter (\(x,_)->str `isPrefixOf` x) l

par_suggestion_filer :: (Eq a, NFData a, NFData b) => [a] -> [([a], b)] -> [([a], b)]
par_suggestion_filer str l = suggestion_filer str $ concat
                         ((map (suggestion_filer str) l') `using` parList rdeepseq)
                         where l' = chunksOf 9 l

suggestion_sort :: [(a, Int)] -> [(a, Int)]
suggestion_sort = sortBy (\(_,x) (_, y) -> compare y (x::Int))
```