COMS 4995 Parallel Functional Programming
Project Proposal-Word AutoComplete

Shengkai Li (sl4685), Wenqian Yan (wy2249)

November 23, 2020

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.

Project Details:

We are planning to archive the following tasks for our AutoSuggestion program:

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.
An example of ***"Word Cleanup***" result: ["haskell", "plt", "programming"...]

- *Sequential Implementation*: (mostly same with our hw4)
  1. Map all characters to lowercase
  2. Filter out words that are non-alphabetic or whitespace
  3. Break into a list of cleaned words
  4. Break into a list of cleaned n-grams

- *Parallel Implementation*: we can run all the four steps parallely. Step 1 needs to finish to start step 2, and step 3 and step 4 can run together, hence we should choose specific parallel strategies accordingly.

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)]

- *Sequential Implementation*: (mostly same with our hw4)
  1. Iterate through the word list to map each word to (word, 1)
  2. Merge entries with the same word to (word, frequency) list

  As the text file is fairly large, it's time consuming without running it parallelly.

- *Parallel Implementation*: we formulate the MapReduce version of the sequential implementation above. Multiple mappers parallely receive word list as input and do step 1 to produce immediate key-value pairs (word, 1), and multiple reducers do step 2 to generate (word, frequency) list. Note that reducers have to wait for all mappers to finish, we would choose parallel strategy accordingly.

N-grams Count: Given the large cleaned n-grams 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)]

- *Sequential Implementation and Parallel Implementation* are similar with ***Word Count*** above.

Word AutoComplete: Based on the result from ***Word Count***, we move further to implement our own version of **Word AutoComplete**. It accepts a word from the user's input and return some `suggestion` to the user in the following three conditions:

1. If our dictionary contains the same word as the user's input, we return the same word.
2. If the prefix of any word in our dictionary does not match the user's input, we still return the same word. (i.e. we don't have any suggestion)
3. Otherwise, we ***search/traverse*** our list. Find out the word whose prefix matches the user's input and has the highest frequency.

- *Sequential Implementation*:
  1. Search along (word, frequency) list for words with matched prefix.
  2. Find top N suggestions by maintaining a max heap with length N when searching along (word, frequency) list.

- *Parallel Implementation*: we can parallelize step1 and step2 to speed up searching lists and maintaining the max heap. We need to choose a specific strategy so that each thread can manage one part of the list for efficiency.

N-grams AutoComplete: Similar to our ***Word AutoComplete***, It accepts a sentence L from the user's input, 0 < len(L) <= N-1, and tries to return some `suggestion`.

For example,

If the user's input is "Haskell is", then we want to suggest the user with "Haskell is *great*".

- *Sequential Implementation and Parallel Implementation* are similar with ***Word AutoComplete*** above.

***Ultimate Goal (What we are planning to do)*:**
(Word + N-grams AutoComplete): Our ultimate goal is to combine the functionalities of both ***Word AutoComplete*** and ***N-grams AutoComplete***. Our program is supposed to accept a sentence of any length from standard input. Then do Word AutoComplete on every word of that sentence and then finally do N-grams AutoComplete on the whole sentence.

For example:

Suppose we have a dictionary containing [("haskell is the best programming language" , 4995), ("haskell is the worst programming language", 4115)].

User input = "Hasl i th"

Our program is supposed to suggest to the user "Haskell is the best programming language". (Assume our dictionary also contains 'Haskell', 'is', 'the', 'best', 'programming', 'language' and they have the highest frequency).

Evaluation:

We will test our program on a Ubuntu 20.04 machine with 8 cores, and compare the performance of non-parallel and parallel. We would expect that parallelism with Haskell could greatly improve the speed in:

1. Parse/Generate (word, frequency) pairs
2. Search top N suggestions based on frequency in dictionary file