Parallel Functional Programming Final Project Proposal

MapReduce Word Frequency Search

Patricia Luc (pbl2116), Arush Sarda (as6785), Sarah Yang (sy3038)

November 27, 2023

Project Overview

For the final project, we propose an extension to the original MapReduce application of counting the frequencies of words from a large stream of text-based input. Given that this alone will heavily depend on I/O in terms of performance, we will augment the program to include an on-demand word search capability once the word counts have been calculated. Users can search for terms from the word counts taken from the text input. If a term cannot be found within the word frequencies, we will run a fuzzy match algorithm incorporating Levenshtein distance to find similar matching words. This project aims to turn the typical non-parallelizable, I/O bound word count problem into a parallelizable project of substance.

Background

Here is a high-level overview of how this project would be performed without parallelization:

- MapReduce Map Stage
  - Since this is a sequential approach, iterate over every input token within the text file and create key-value pairs of that token and the number 1.
- MapReduce Reduce Stage
  - Iterate over the key-value pairs and sum them up, producing the count of each word in the text
- Take in user input words, and search up their frequency in the map
  - If the word exists in the map, return the count
  - Otherwise, run a fuzzy search on every key inside the map, finding the Levenshtein distance between the input word and the key, and rank the most similar key words. Return the most similar word and its respective count within the map

Parallelization

The algorithm allows for multiple opportunities for parallelization. Firstly, the MapReduce word count algorithm can be parallelized as follows:

1. Given K cores, split the input file in K chunks, one assigned to each core
2. For each core:
   a. Complete the map stage of MapReduce on the given file chunk
   b. Complete the reduce stage of MapReduce
3. Aggregate reduce stage outputs for all K cores, store into one map

Secondly, the word search algorithm can also be parallelized in the case that the search word does not exist in the map. The idea is as follows:

1. Given K cores, split the word count map into K chunks, one assigned to each core
2. For each core:
    a. Iterate every word the map chunk, and calculate the Levenshtein distance between each word and the search word
3. Compare the intermediate closest words from each chunk, and return the word that is closest to the search word

References

[1]https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
[2] https://redis.com/blog/what-is-fuzzy-matching/
[3]https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0