Parallel Functional Programming Project Proposal

Daniel Scanteianu(UNI: dms2301) & Stanley Ye(UNI: yy2922)

1. Background

Searching a collection of documents for keywords is a well studied and frequently implemented problem. In order to search a collection of documents for one or more keywords, a metric must be established for how relevant each document is to the keywords specified. TFIDF is a widespread method to determine the relevance of words within a document. TFIDF stands for term frequency inverse document frequency. We will use TFIDF for keyword extraction and giving documents a relevance score.

We will build a system that loads and indexes a set of documents inputted as raw, unformatted text, and then supports interactive queries on top of the indexed documents.

2. The Algorithm

a. Data preparation:
   For each document, the count of the number of occurrences of each word in the document will be computed and stored at the document level. Then, for each word globally, a count will be computed of how many documents the word appears in. After this is done, at the document level, we will divide the count of occurrences of each word by the number of documents it appears in globally to compute TFIDF for the word for the document.
b. Searching:
   The user will enter a set of keywords. We will scan the set of documents. For each document, we assign a score, which starts at 0. For each keyword, if the keyword is present in the document, we add its tfidf to the score. At the end, we will have a score computed for each document. We can then return the top N most relevant documents by score.

3. Parallelism:

In the startup phase, indexing each document can be done either sequentially or in parallel, and the effects of indexing in parallel can be measured. A textual representation of the tfidf score of each keyword in each document can optionally be written to a file so that the indexing is a standalone component which can be measured in isolation.

During the query execution phase, when we compute the score for each document that can either be done in parallel (one spark per document/batch of documents) or sequentially. Because IO is limited, a script that measures when the query is sent and when the response is received should be able to pretty effectively measure the effects of parallelism on the actual search.

4. Testing Plan

Tests can be done across several dimensions (below are the x axes where time is the y axis):

- How big is the unit of work
    - Documents can be batched together into one spark, with the batch size being inversely proportional to the number of sparks
    - Each document is a spark and we can compare indexing a lot of small documents vs indexing a smaller set of bigger documents
- Test end-to-end sequentially vs end-to-end in parallel, and also test a system that just reads the files and skips the indexing part in order to figure out how much time is spent doing non parallelizable file IO (amdahl's law measurement)

Software design

Major Components:
Startup/Indexing:

- Given a folder or a list of files, reads all of them, and produces a list of strings
    - Likely has to be sequential due to the nature of reading data from disk read more here
- Given a list of strings, cleans each of the strings, and produces a list of keywords in each
    - Can be parallelized, per document
- Given a list of strings, computes a dictionary where the key is the word, and the value is the frequency
- Given a list of such dictionaries (one per document), compute a dictionary where the keys are all the keywords that exist in at least one document, and the values are the number of documents they appear in
    - If we start with one such dictionary per document, and this dictionary is computed by repeatedly merging two adjacent dictionaries, this can be parallelized, but there might be a more efficient way to do this
- Given the global keyword presence dictionary and the keyword frequency dictionary, compute tfidf for each keyword for each document

Query time:

- Given a search string, split it up into a list of keywords
    - Will have to be sequential
- Given a search string and a set of tfidf dictionaries (one per document) - compute the score for each document
    - Easy to parallelize
- Sort the documents by score
    - Probably easiest to do sequentially
- Display top n items to the user

Other Ideas and proposals:

Website graph builder/Baconator

Given a bunch of text/html files, find all hyperlinks in the text, and build a graph where nodes are pages, and links are edges.

Use this graph to do a 6 degrees of kevin bacon type algorithm (bfs) where it would print the shortest path from one page to another page

TFIDF

Extension:

Linking documents by keyword to form a knowledge graph - ie: keywords are nodes pick the top 5 scoring tfidf terms in each document, and make weighted edges from one keyword to another (where weight is sum of tfidfs of the two keywords across documents they appear in together) - potentially use this to suggest related articles.

(Related to this: sorting related problems such as counting sort, radix sort and bucket sort; maybe **deterministic selection algorithm**)

Interesting Problems in Spring 2020

1. random graphs defined over the natural numbers.
2. solve 2048-puzzle game (Maybe we can do something similar like min-max tree search (tictoc games))
3. freecell game (This is also interesting, and it uses parallel programming for searching.)
4. crossword solver (More like a searching problem.)
    a. This is my preferred backup problem - ds