Parallelizing A Search Engine

Daniel Scanteianu and Stanley Ye

Introduction and Background

Searching a collection of documents for keywords is a well studied and frequently implemented problem. We have implemented a search engine which indexes a collection of documents and then takes a set of keywords, and returns the most relevant documents in relation to those keywords.

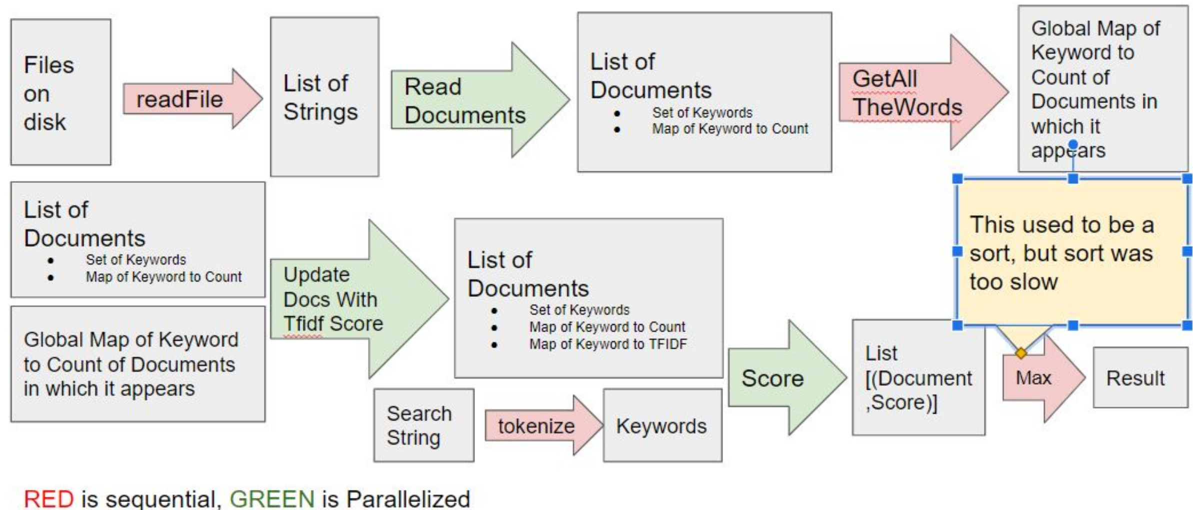
In order to search a collection of documents for one or more keywords, a metric must be established for how relevant each document is to the keywords specified.TFIDF is a widespread method to determine the relevance of words within a document. TFIDF stands for term frequency inverse document frequency. We designed a document indexing and search system using TFIDF for keyword extraction and giving documents a relevance score.

We built a system that loaded and indexed a set of documents inputted as raw, unformatted text, and then supported querying on top of the indexed documents.

Software design

The search engine pipeline consisted of two components - the ingestion and the search.

a. Ingestion:
   The documents were loaded from a user-specified file or directory. In the case of the file, each line was considered to be a distinct document. For each document, the count of the number of occurrences of each word in the document was computed and stored at the document level. Then, for each word globally, a count was computed of how many documents the word appears in. After this was done, at the document level, we divided the count of occurrences of each word by the number of documents it appeared in globally to compute TFIDF for the word for the document.
b. Searching:
   A user entered string was tokenized into a set of keywords. For each document, we assigned a score, which was the sum of the tfidf scores of the keywords that appeared both in the search string and in the document. Initially the documents were sorted by score and the top 5 were returned, however we then switched to a maximum as maximum is faster than sorting.



Experimental Design

In order to profile the search engine, we wanted to do a number of experiments to isolate the effects of parallelism, document size, and number of documents on performance.

We designed a series of experiments to try to measure these different factors in relative isolation.

For all of the experiments we performed, we used the tweets from a Kaggle dataset: https://www.kaggle.com/thoughtvector/customer-support-on-twitter. There were around 3 million tweets, and we reduced these tweets into datasets of 100,000, 500,000, 1,000,000, and 1,500,000 tweets (experimentally, we found that 1,500,000 tweets is the most that our software could handle on 16gb of ram before becoming IO bound due to swapping to and from disk). We took the 100,000 tweet dataset, and turned it into 12 very long documents of approximately equal length by concatenating tweets together in order to simulate "large" texts.

We tried to time the individual components (represented by arrows in the component design), starting specifically with the "search" components (reading, scoring, and max-ing keywords). We also tried to get end-to-end timings.

Tweets are a good data set because they're easy to obtain, fairly predictable in nature (ie: they're all approximately the same length, use the same kind of casual tone, and in the case of our data set, were overwhelmingly in English, meaning that there were no special characters to contend with, and they were all fair game from a search results perspective).

The 4 experiments that we did were:

1. Find the effects of parallelism on the search mechanism components
2. Find the effects of parallelism in combination with number of documents on the end-to-end timing of ingestion and search

3. Find the effects of parallelism in combination with size of documents on the end-to-end timing of ingestion and search
4. Find the effects of chunk size on performance

Experiment 1: Timing Individual Components

In order to try to evaluate the effects of parallelism on each of the components of the architecture (represented by arrows in the diagram), we tried to force Haskell to perform operations sequentially and get timestamps in between each operation. The below code sample illustrates our method: we tried to get a timestamp, perform each step of the search operation, and then get another timestamp. We printed the difference between the timestamps, which we expected to be indicative of the amount of time it took to perform each step. We tried seq, deepseq and no seq at all, and we found that unless we used deepseq, the difference between cpuTime0 and cpuTime1 and cpuTime2 was all 0 with all the time being measured between cpuTime2 and cpuTime3. We hoped that this design would give us a very granular measurement of the relationship between number of threads and component performance. We had a sequential version of the code (where the parMap below was replaced by a simple map), and we compared the sequential timings to the 2 thread and 4 thread timings of the parallel version.

```
searchAndSortPar:: String -> [Document] -> IO [(Document,Double)]
searchAndSortPar keywords docs = do
  cpuTime0 <- getCPUTime
  let kws = cpuTime0 `deepseq` tokenizeAndNormalize keywords
  cpuTime1 <- kws `deepseq` getCPUTime
  let docScores = cpuTime1  `deepseq` (parMap rseq (docScore kws) docs )
  cpuTime2 <- docScores `deepseq` getCPUTime
  let sortedScores = cpuTime2 `deepseq` sortDocsByScore docScores
  cpuTime3 <- sortedScores `deepseq` getCPUTime
```

Experiment 2 a and b - End to End Time and Multiple Threads

We wanted to measure the effects of parallelism on the end-to-end timing of the system (which included reading the documents, indexing them, and performing one search). We did some preliminary experiments to determine what we should measure (in terms of number of threads, number of documents, etc). We then created a Python script which would invoke the indexing program 10 times and time its execution each of those times, writing the results to a csv (the results were written to stdout and then piped into a csv). It repeated this for 1 to 5 threads (we determined in preliminary tests that above 5 threads, adding another thread meant adding more time to the execution, so we stopped at 5 because we wanted one run to illustrate that more threads than physical cores would always be slower than multithreading for our code). This script was used to test different scenarios (input documents changing and input documents of different sizes).

For these experiments we used parListChunk as our parallelization strategy (initial experiments showed that parMap with any number of threads was slower than regular map, likely due to the overhead of making sparks, as our map operations were relatively cheap, computationally

speaking). We used 32 chunks, as that seemed like a reasonable number (we expected to go up to 8 threads initially, so each thread would do 4 chunks of work on average; if we used fewer chunks, one chunk hanging would cause all the other threads to idle while it finished, and we figured that with 4 chunks per thread, if one chunk took twice as long as the others, another thread would be able to pick up the remaining chunks from the thread, leading to reduced idle time).

Experiment 2a: Relationship between Number of Documents and End-To-End time
We used our test harness to test the change in execution time in relation to change in number of documents. We used 100,000, 500,000, 1,000,000, and 1,500,000 tweets. The reason that 1.5 million was chosen is because we observed that for the whole corpus, the time was enormous (3 million documents took around 700 seconds), and further investigation showed that ram was filling up completely, likely causing more time to be spent swapping to and from disk. 1.5 million was the smallest number of tweets that filled up ram completely, so we used that to illustrate the effects of having a memory-bound computation.

Experiment 2b: Relationship between Document Size and End-To-End time
We used our test harness to test the change in execution time in relation to the size of the documents. We took our 100k tweet sample file and we turned it into 12 approximately equal sized documents by combining adjacent tweets.

Experiment 3: Relationship between Chunk Size and End-To-End Time
Since parMap failed our expectation when there were too many documents, we would like to explore how the number of chunks would affect the parallelism. We tested upto 4 threads with 4 chunks, 12 chunks, and 20 chunks, as well as 100k and 500k tweets. And the variations of chunks only applied in the “Score” step in the architecture diagram, and we only timed the “ Score and max” steps as our result data. For each unit experiment, we ran 10 times and recorded the average and standard deviation. Then, we could look at how the number of chunks would affect the performance of parallelism.

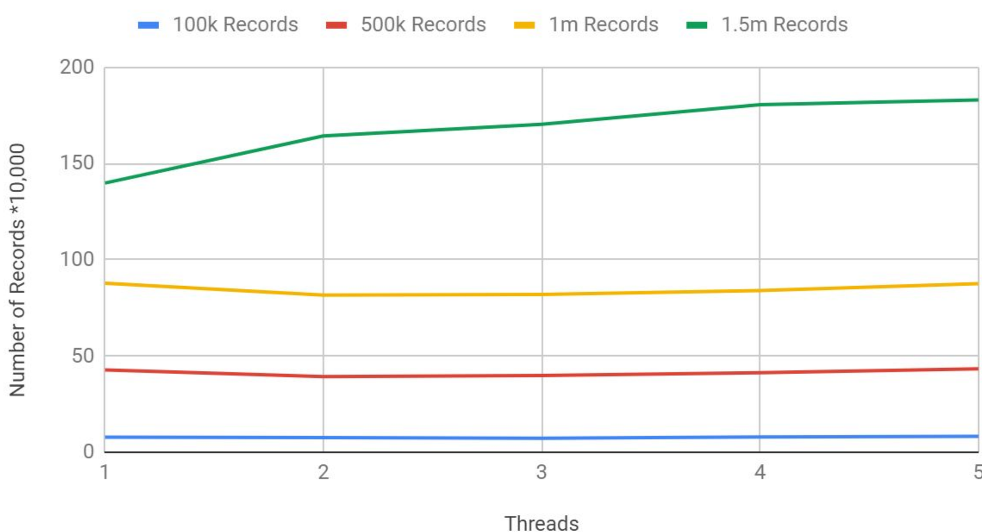
Experiment Results
Experiment 1: Search Timings
We performed experiment 1 only for the search component, which we expected to show some improvement when being parallelized. However, the results we observed were not in line with expectations: parallel scoring was always slower than single threaded scoring (we later ascertained that this might be due to parmap making one spark per list element, and in later experiments we used parListChunk to alleviate this problem, but this one was not rerun). Additionally, it is impossible for the keywords step (taking a string and turning it into a list of keywords) to have taken 0 time. Furthermore, the sorting step was supposed to always be sequential, so we really expected to see no variation in that timing. Finally, the numbers that were produced were often suspiciously similar (ie: scoring in parallel with 2 threads took the exact same amount of time as sorting with 4 threads and parallelism turned on). These are picoSeconds, according to the haskell getCpuTime documentation, so the results we saw were indicative of something being off. We also tried to `seq` instead of `deepseq` and no `seq` at all,

and in all cases but `deepseq`, sorting took all the time due to the laziness feature causing nothing at all to get evaluated until it was time to actually get a result.

Category	St - 2 thread	Par - 2 thread	St - 4 thread	Par - 4 thread
Keywords	0	0	0	0
Scoring	281250000000	312500000000	281250000000	343750000000
Sorting	968750000000	328125000000	1234375000000	312500000000

Experiment 2a: Number of Documents

We found that increasing the number of documents didn't have any particularly large effects on the speed up as more threads were added. We found the best improvement was around 10% for 500k documents when going from single to multithreaded execution. We found that in all cases when the number of threads exceeded the number of available physical cores (4 cores), execution slowed down and was slower than single threaded. We found that at 1.5 million records, the process became memory-bound (ram usage on windows task manager was observed to be at 99% and swapping constantly), and therefore, adding more threads added to overhead and slowed down the process. We found that 2 and 3 threads were generally comparable, and 4 threads was slower than 2 or 3 (but this may be due to the contention when using 4 cores with other things the OS is doing in the background).



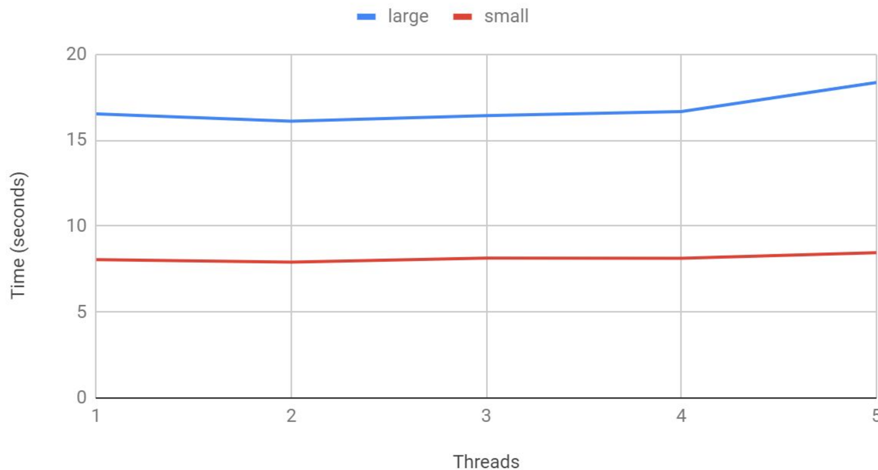
100k:	Average Time	Standard Deviation
1	7.820177237	0.5286586286
2	7.519708554	0.5395692695
3	7.275554551	0.04967302365
4	7.885595904	0.7468388326
5	8.196732601	0.6953339016
500k:		
1	42.76505513	0.8531633957
2	39.22405553	0.8187850185
3	39.78823646	0.5938956038
4	41.32797538	1.488843918
5	43.26393572	1.306905599
1m:		
1	87.73674538	0.890734873
2	81.66040908	2.508054366
3	81.96948939	2.096340273
4	83.95706103	1.807369928
5	87.60235129	1.696625691
1.5m:		
1	139.8346421	27.54136564
2	164.2952562	10.29333662
3	170.3952842	10.56777717
4	180.5431903	7.693357383
5	183.0082534	11.4533551

Experiment 2b: Document Size

We found that the same performance patterns occurred both in the large document and small document experiment - 2 and 3 threads were faster than single threaded for the large document, and the performance gains were small. Large Documents took about twice as long as small documents despite having the same amount of total text. This is likely due to usage of maps and sets where the total performance is O(nlogn) where n is the size of each document. With bigger documents, log n is bigger.

Threads	Large Document Avg Time	Large Document Time Stdev	Small Document Avg Time	Small Document TimeStdev
1	16.52573522	0.729064121	8.054771476	0.09631376774
2	16.10765388	0.5411509271	7.905438423	0.393094967
3	16.42805804	0.5279924703	8.136161831	0.5013071916

4	16.66502089	0.5292841538	8.133448256	0.4757304653
5	18.35853235	0.9809306468	8.453650342	0.3673195833



Experiment 3: Number of Chunks

Experiment result table:

	100k			500k		
	4 chunks	12 chunks	20 chunks	4 chunks	12 chunks	20 chunks
1 Thread	6.380	6.414	6.301	36.815	37.477	36.798
2 Threads	5.825	5.815	5.813	34.812	35.032	33.472
3 Threads	6.031	5.991	5.948	36.947	33.062	33.743
4 Threads	6.456	6.454	6.504	38.620	37.900	36.310

Our experiment 3 shows that the number of chunks do not have much impact on the parallelism performance on the 100k tweets. From the result data, 1 thead to 4 threads with 4 chunks, 12 chunks, and 20 chunks do not have much difference in terms of their performance based on the result data. Comparing the performance of different threads, it does match the previous experiments that the performances of 2 threads and 3 threads show some improvement than 1 thread and the performance of 4 threads are similar to 1 thread or slightly worse. In terms of 500k tweets, it shows that larger chunk splits would slightly improve the performance. For the same number of threads, more chunk splits could increase a little performance. Therefore, the conclusion for the experiment 3 is that the number of chunks with smaller document sizes do not heavily affect the performance; with larger document sizes, larger chunks would slightly improve the performance.

Discussion and conclusion

We were able to successfully implement a document indexing system which used tfidf to score document relevance based on keywords. We were able to demonstrate moderate performance gains by parallelizing, but we were not able to attribute the time spent during execution to any individual components. A future step would be to try to parallelize the search component's extraction of the top scoring documents, and to try to make a rest api or something similar that interactively serves up searches, so that we could time performance gains from just search which can be made to run entirely in parallel. We should also investigate what happens when we have many chunks (ie: close to or equal to 1 chunk per document) in terms of performance, and also try to get GC metrics. We found that there are limitations to the benefits that can be gained by parallelism, namely we found that having more threads than physical cores, or using more memory than the system has available will both lead to very poor performance.