Parallel Optimization of TF-IDF File Comparer

Luke Burford

12/13/18

CS475 - Parallel Programming

Introduction

For the project I will be working on parallelizing a C++ program that I worked on in my sophomore year. The program is meant to read text files, separate punctuation from meaningful words, resolve capitalization, stem the words using the Porter stemming algorithm[1] and a table of exceptions; and count the frequencies with which words and punctuation streams occur. From there using an algorithm known as TFIDF (Term Frequency Inverse Document Frequency)[2] which will run comparisons on word counts/ occurrences in and between files to calculate a matrix of similarity scores, as defined by the TFIDF algorithm. I believe there can be a large amount of improvement to be gained using OMP.

The Algorithm

The algorithm is relatively simple on the surface. Every word in a document gets a TFIDF score. The TFIDF score for a word in a document is the product of two numbers: $TFIDF(word, doc) = tf(word, doc) * idf(word)$. In this equation, $tf(word, doc)$ is simply the number of times the word appears in the document. The second term is the inverse document frequency, and it is defined in terms of N (the total number of documents) and n, the number of documents containing at least one instance of the word: $idf(word) = log10(N / n)$.

To measure the similarity between two documents, find the set of words that appear in both documents. For every word in the intersection of the documents, multiply the two TFIDF scores together, and sum those products. In other words:

$$Sim(doc1, doc2) = \sum_{w} TFIDF(w, doc1) * TFIDF(w, doc2)$$

Where the summation is over all words (including capitalized words, etc., but excluding punctuation) that are common to both documents. This is where a majority of the computational weight is in the program.

Program Analysis

When breaking up the program it can fall into three major functions: File loading, File Parsing, File Comparing. So there are 3 areas that can be analyzed for performance gains. There are two problem sizes that we can influence: number of files to compare, and size/ quantity of a file to compare.

Experiment Setup

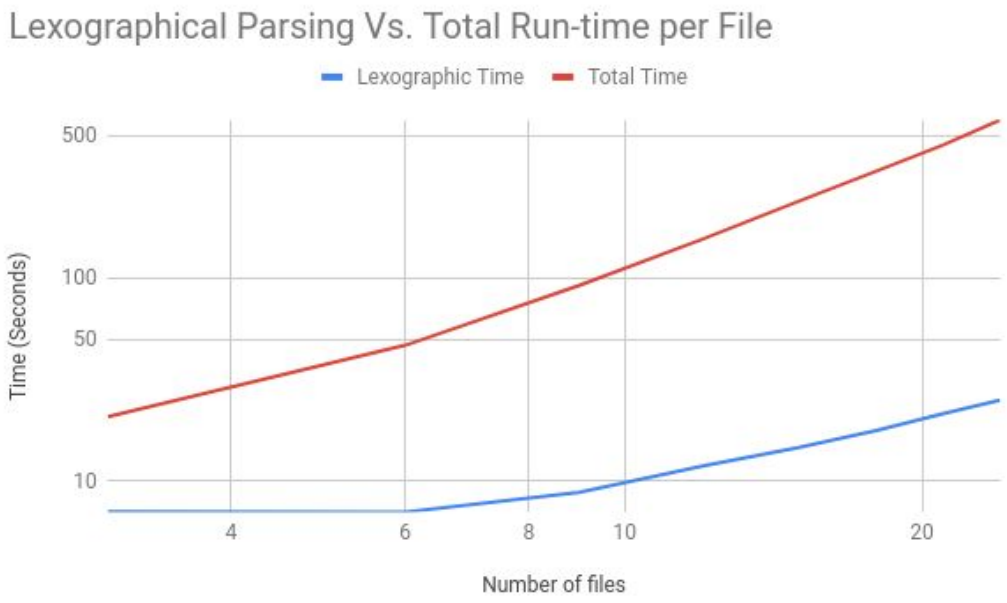
I ran tests against all of the with a large problem size. For file count I had it start with three files (the minimum number of files required to do a comparison). Each file had a consistent 8025 lines each. I then increased the number of files by three each test up to 24 files. For the file quantity tests, 3 files were consistently compared; but the size of the files increased by 8025 lines each iteration. The max size compared is 188,715 lines per file. I ran all of my tests using a python script. I first measured all of my total run times on the sequential program and from there ran a timing function on individual components to get an idea where the program was bottlenecking. I started by analyzing the run time for loading and parsing a file. All charts will be formatted logarithmically, in order to get a linear comparison. All of my tests were run on the fish lab machine which have a:

Intel Xeon e5 1650 V4

Cores	6
Threads	12
Theoretical GFLOPs	27.36 (4.56 per Core)
Max Memory Bandwidth	71.53 GBi/s
L1	6X32 kB (Data & Inst)
L2	6x256 kB
L3	15 mB (shared cache)

File Parser (Lexo.cpp / fDriver.cpp)

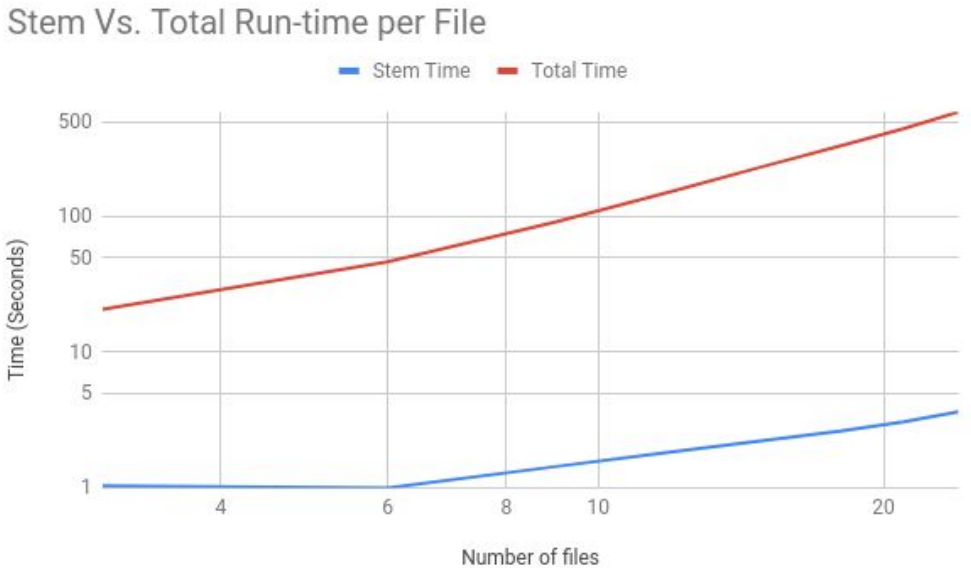
I decided to start analyzing the file parser due to the fact that the theoretical iteration space for this will have to iterate over every string in a given file, so the complexity compared to file size would be enormous. Surprisingly compared to execution time I was not entirely right.



Based on my benchmarking the file parsing only takes on averager 5.8% of the total execution time of the program. The reason for this is most likely to do with the C++ file parser library IFStream that the program uses being highly optimized itself. From here I tested the Stemming algorithm.

File Stemmer (StemExcep.cpp / Stem.cpp)

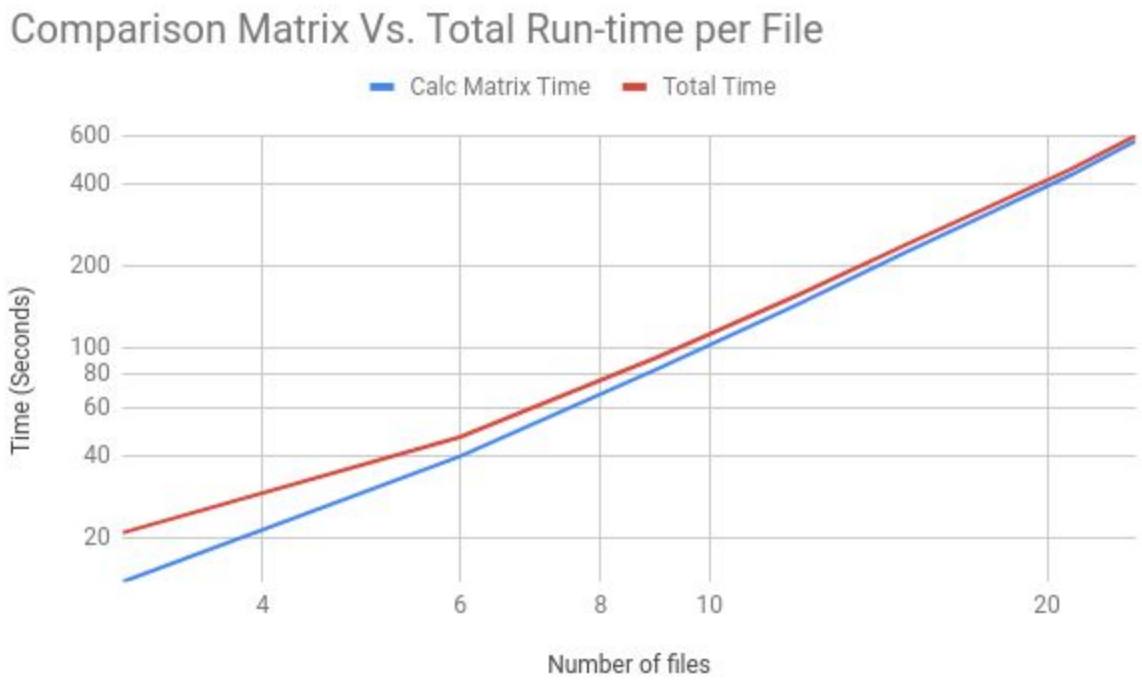
The file Stemmer uses the Porter stemming algorithm which will take a string and following a very precise set of steps will break a word down into its core components. Knowing this alone I already had a pretty good idea that this would have to remain sequential due to the nature of the algorithm.



Similar to the file parsers, the word stemming functions take up a very small fraction of time, just 0.85% total execution. At this point I knew that the file comparer function was what was eating most of the execution time.

File Frequency Comparer (fDriver.cpp)

This is built up of 5 functions each making up a 5 deep nested loop. Within these loops are multiple vector iterations and floating point arithmetic. I knew from this and my previous 2 tests that this is where the majority of the computation time was occuring.



Sure enough this is where almost all of the run-time was being used up. At this point, the focus moved to trying to optimize this portion of the code to get the greatest yields in performance.

Parallel Process

I started by trying to break up the large amounts of loops, vectors, and maps into easier to understand iteration spaces:

```
for(Document A)
  for(Document B)
   for(Word List C)
    for (Word List D)
     if(first character in word C < first character in word D)
       break;
     if(word C == word D)
```

```
        sum+= (count C in Doc * count C in other Docs) *
                  (count D in Doc * count D in other Docs);
return sum;
```

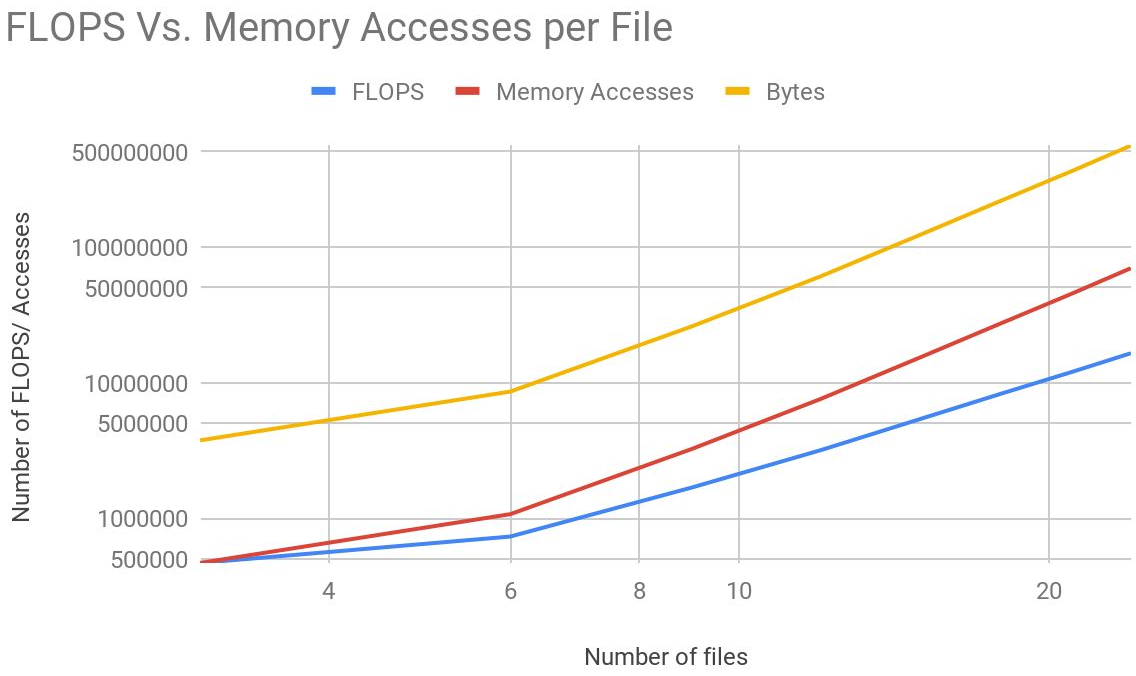
All of this is defined by the TFIDF algorithm:

$$Sim(doc1,\ doc2) = \sum_{w} TFIDF(w,\ doc1) * TFIDF(w,\ doc2)$$

$$TFIDF(word,\ doc) = tf(word,\ doc) * idf(word)$$

$$idf(word) = Log_{10}(\tfrac{N}{n}) * tf(word,\ doc) \quad \{N = total\ \#\ docs;\ n = \#\ docs\ with\ similarity\}$$

These iteration spaces are exponential with problem size. In order to get a good idea exactly what I'm dealing with, I have placed counters at points where floating-point operations occur and where the program loads/ writes to memory.

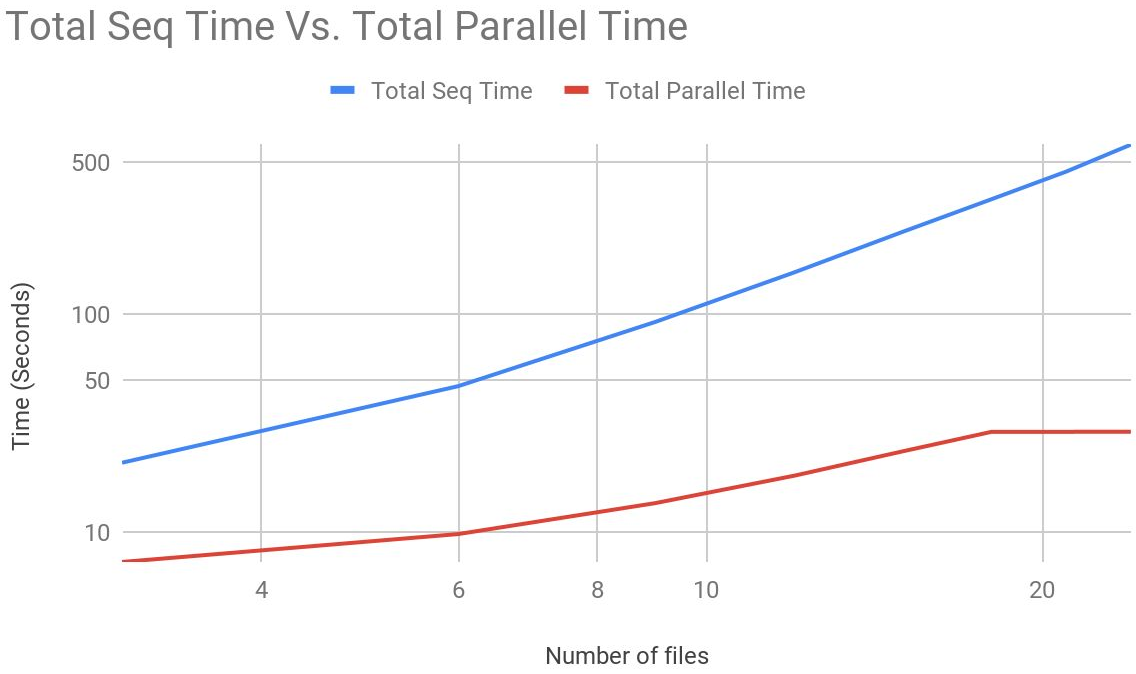


By quickly looking at these charts it's easy to see that this program is heavily memory bound. And this makes sense knowing that a single word has to be counted in a file: n^N and compared against every word in the other files: N^T, meaning we have to compare n^{N^T} strings against 1 string in a file. Unfortunately due to time constraints, complexity of the dependencies, and iteration space, I was unable to make much of a impact on the file access intensity. However, by using this analysis I was able to make a impact on improving performance with OMP.

I started trying to parallelize the outermost loops by giving a thread each a single document, which made very little difference in my initial tests due to the required dependencies in the the inner loops.

I then looked at the function calls being made within the loop bodies, I called them using tasks and with pruning at a certain depth to prevent overhead. And this made a significant difference. At the very bottom of the loop bodies when calculating IDF. I used multiple threads to compare the one word with the many documents and this also had a huge impact on performance.

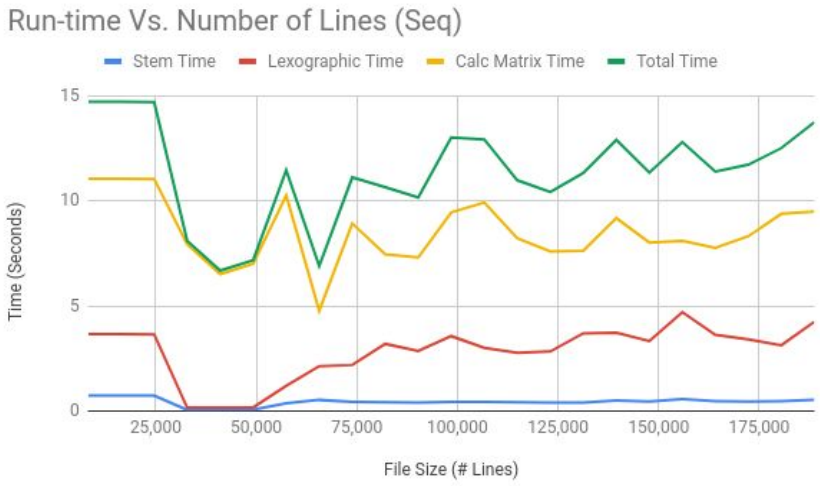
After that I began to test against number of threads and found through diminishing returns that an optimum thread count is 4.



Using the ISO-efficiency calculation, $\psi(4) = T(1) / T(4)$ we find with 4 threads there is a speed up of 12.3 which is a massive improvement. However looking ahead at our theoretical machine peek there is still a massive amount of improvement to be gained here since we are still going to be extremely memory bound.

Sources of Error

There were several sources of error in this project. The first, and largest is the inaccurate testing of the increasing length of files. Unfortunately due to the difficulty of managing and allocating the correct size of the files lead to some strange results:



This is likely due to a few factors. The first major one being that the length of the word, the amount of words per line, or the evenly wrapping of text was not even throughout. A potential solution to this would be to get a better automated way to take in a large amount of plaintext and evenly distribute them to the files accounting for word count not line count as I did. Due to the time constraint I was unable to perfect this. Although this data was not useless as it did help solidify the fact that the file comparison was the bottleneck of the program as a whole.

Another source of error was the inability to find a memory solution to the comparison matrix. With more time I know that it is obtainable, just not with the constraints with 1 person. Overall the improvement was still a huge success.

Conclusion

This project took a lot of work and was really interesting. All though I had a lot of shortcomings; I know that if I had another week or so to work on this and improve the memory bandwidth it could be much faster. Overall the speed up increase that I did find was significant and shows the potential improvement still yet to come.

Resources

1. http://snowball.tartarus.org/algorithms/porter/stemmer.html
2. https://en.wikipedia.org/wiki/Tf%E2%80%93idf