Multithreaded Text Analyzer in Java

Gavin Barber, Calvin Agar, Aribel Ruiz, and Amasha Rajaguru
Undergrad Students, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
{gavinb, calvinagar, aribelruiz, amasha667}@knights.ucf.edu

***Abstract*— This research paper discusses the effects of implementing a multithreaded text analysis program. Text analysis is the process of extracting meaning and insights from unstructured or semi-structured text data. This data can include a large variety of sources such as movie scripts, new articles, video transcripts, social media posts and more. Many businesses and individuals benefit from the use of preexisting text analysis software. In this project, we discuss our implementation of a multithreaded text analysis Java program that can read text data and return meaningful insights from the text. Our goal is to evaluate whether multithreading can improve the performance and efficiency of our program, enabling it to process larger volumes of text data quicker. Additionally, we aim to provide some unique analysis of the text data to distinguish our program from the many other text analysis software. This paper also explores the impact multithreading can have on the scalability of text analysis software.**

***Index Terms*— Multithreading, Text Analysis, Scalability, Parsing, Resource Utilization**

I. INTRODUCTION

Text analysis is a field that has existed for many years but in recent times has made many strides in advancements. These programs are able to extract useful information from large volumes of text data. Text analyzers can be used to understand a customer or reviewer's sentiment about a certain product, service, or movie. There are many practical applications of these programs. However, analyzing massive amounts of text can be computationally intensive and time-consuming. A solution to this issue is to implement a multi-threaded approach to the text analysis process. In this research paper, we explore the impact multi-threading has on text analysis and present a Java program that uses multi-threading to analyze large volumes of text data efficiently. Furthermore, we discuss design the design and implementation of our program, the challenges we encountered during development, and the results of our experiments.

A. Problem Statement

Our goal is to implement an efficient multi-threaded text analyzer in Java whose run time is faster than a single-threaded approach. Useful information such as words most commonly used can give insight into words that are used too often in a speech or whether certain dialogue is appropriate for a specific audience in a film's script. We aim to provide text statistics as an output for our text analyzer which includes the most common words, the number of unique words, and the number of sentences in a text.

As our program analyzes text, it should also gain statistics regarding inappropriate language predefined by the developers to give users greater insight as to whether the content is appropriate for certain ages. Our program should also take input from the user containing words to search for within the text to gain statistics on how many times those words appear and which are most common.

Once our program is complete, the results will be compared to a single-threaded approach to determine if a multi-threaded approach can produce the same results at a significantly faster rate. Using the run time results obtained from comparing a single-thread approach to approaches of varying thread counts, the most optimal number of threads when analyzing text will be determined. The total number of words within a text will also be counted and evaluated to determine how many words are required within a text for the most optimal performance for analyzing text using multi-threading.

II. MOTIVATION

One key motivation of having the text analyzer use multi-threading is that the program can analyze text more quickly and efficiently than a single-threaded program. This can be especially useful for analyzing large volumes of text or for real-time analysis, such as in social media monitoring.

Another motivation stems from the fact that the ability of getting the most common words in a piece of literature or text whilst ignoring words such as "and" is highly applicable in a variety of fields. The program can be used by researchers and scholars to quickly analyze texts and identify key concepts and themes. By filtering out common English words, the users can focus on the most important and relevant words in the text, which can help researchers to better understand the content and identify patterns. Students and educators could also use the program to analyze texts and identify key themes and concepts, which can help them to better understand the material.

Other resulting information from the program such as the number of sentences and unique words can help

content creators, such as writers and marketers, use the program to analyze their own writing or the writing of others. By identifying these statistics, they will be able to recognize if some words are overused in a piece of writing. Additionally, if the number of sentences in the text is lower than expected, that might give an insight into poor grammar usage. With the other aforementioned features as well, they can improve their own writing and better target their audience.

Overall, a multi-threaded text analysis program such as this seems to have many potential applications and benefits in a variety of fields.

III. RELATED WORK

Various tools specializing in text analysis, text analytics, and text mining are available online, such as application programming interfaces (APIs), paid downloadable software, and free web applications. Many advanced and pre-existing programs are heavily focused on text mining, the use of artificial intelligence technology that uses natural language processing (NLP) to transform large text into structured data suitable for analysis or machine learning algorithms [1]. As opposed to the use of artificial intelligence in text mining, text analysis involves the general use of a computer system to read and provide insight into the text.

Paid tools such as SpeakAI lean more towards the specialty of text mining as SpeakAI uses artificial intelligence to produce text analytics or patterns and trends that are statistically significant in text, and allows users to enter information to create custom data sets [2]. General text analysis is offered by different tools including MAXQDA, specifically, MAXDictio, which is designed to facilitate text analysis allowing for the search of relevant terms and word combinations [3].

Besides downloadable software, text Analysis tools offered online come in other forms such as MeaningCloud; a set of APIs offering text analytic functionalities within code [4]. The multi-threaded text analyzer we aim to develop will not have to be integrated into code or paid for, but rather a free program anyone can run and get text analysis from text right away.

Many text analyzers online also have specific niches, whether it be for business data or social media analysis. One text analyzer with a niche of analyzing text in resumes for hiring organizations is Daxtra [5]. Text analyzers with specific niches such as Daxtra search for data such as contact information that suits the context of the text it is parsing through.

Our goal is for the multi-threaded text analyzer to be used for various forms of text such as movie scripts, books, dialogue from podcasts, etc. As this program aims to provide better text context, we hope to search for specific data such as inappropriate language and provide general data analysis so that the program may be useful in multiple scenarios.

The preexisting software we found to be most similar to what we hope to accomplish is a free web application called Online Utility that provides the most frequent phrases and frequencies of words in a text [6]. It is undefined on Online-Utility.org what approach they used to analyze the inputted text, but our goal is for the multi-threaded approach to be able to analyze larger text in a shorter amount of time than that of the text analyzer found on Online-Utility.org [6].

IV. IMPLEMENTATION

For our multi-threaded implementation, we used Java and utilized the synchronization keyword at critical sections and the java.util.concurrent package for ConcurrentHashMaps and AtomicIntegers mainly. We also split the program into two different phases, with different threads operating at each phase.

A. Plan

At first, we ask the user for the name of the text file they would like to parse and analyze. Then, we ask the user if they would like to include a list of custom words to search for. These words will be handled differently when it comes to analyzing the file, which we will talk about later. A user can also decline to give a file containing custom words.

After this, we can start the main program and read the file that the user is trying to process. We will take all of the input in the file and put it into a list that the first set of threads can use to access a given word. Unfortunately, since file reading is done on the disk rather than the CPU and the disk does not support multi-threading, it is never faster to parallelize this part of the application.

Once we have a list of the words representing the file, we will start the ParserThreads, the first set of threads, which will use the list and work together to create a ConcurrentHashMap; with the key being a specific string from the file and the value being the number of times this word was found in the file. We have implemented our own Counter class which will be shared by all of the threads. This Counter is a custom atomic integer mutually exclusive by use of Java's synchronized operator, so this will prevent double counting.

The thread will take the current number in the shared Counter and use it as the index for retrieval from the array of strings. Once a thread has a string from the list, it will use a regex to decide if the string has any letters in it. If it does not, we will parse it as a number. If the string does have letters, we parse it as a word.

Different behavior is required for numbers and words because we want the most control over how punctuation is taken out of the input. For example, if a given string was '1.23' and we parsed words and numbers the same way, we would consider the decimal place to be where the string ends, removing the '.23'. However, since we are able to differentiate between this being a number, we will keep the full string and place it in the HashMap. The punctuation removal is done using an additional two

regices. For strings that are combinations of letters and numbers, we will always parse them as words.

Since multiple ParserThreads will need to be reading from and writing the HashMap at the same time, we used Java's ConcurrentHashMap. It is important to note that with Java's ConcurrentHashMap is that "even though all operations are thread-safe, retrieval operations do not entail locking". However, "an update operation for a given key bears a happens-before relation with any (non-null) retrieval for that key reporting the updated value."

Additionally, to help further protect concurrency for updates, the value in the ConcurrentHashMap is an AtomicInteger. Therefore, we feel comfortable under the assumption that we will not attempt to retrieve a value while it is trying to be updated. We have also implemented a single-threaded version of our program to test the parallel version again. Once the ParserThreads are finished executing and the HashMap contains all of the individual words in the file and their counts, we can start the analyzer threads. Quite a bit of setup is needed before we can begin.

To start off, we must put the HashMap of words and their counts into a data structure that is easily and consistently readable for the AnalyzerThreads. This involves creating an ArrayList and filling it with the HashMap's entry set. This gives us an array of entries (the word mapped to its count) allowing us to again utilize our Counter class for keeping track of which word a given thread is accessing. Then, we build the other data structures the threads need to operate on. This includes AtomicIntegers for the total count of words in the file and the number of times a word only appeared once in the file.

We also have a ConcurrentHashMap implementation, called CHMWrapper. The CHMWrapper contains a synchronized method called "check()". This method takes in a word's entry and will compare it with every other entry its HashMap contains. The method's goal is to maximize all of its values, so if it finds an entry with a lower count than what was passed in, it will replace that entry. This implementation was necessary because we needed to only allow one thread at a time to access this critical section, otherwise, race conditions are rampant since we're doing a considerable amount of work involving comparisons, removals, and updates.

We have two CHMWrappers currently, one for keeping track of the common words with high counts, and one for the uncommon words with high counts. We differentiate between common and uncommon words using a list of the one-hundred most common words in the English language, courtesy of Wikipedia [8]. Finally, we must do a bit of extra work if the user uploaded a file of custom words to search for. If they did, we read the file and create a ConcurrentHashMap for capturing all of the words we find. Note that here we do not use a CHMWrapper. It is unnecessary since we want to treat all of the custom words the user gives us with the same priority and not remove any for having a lower count than another. One other note, in the cases of the lists of common English words and the list of custom words, we use regular HashSets. A concurrent one is not needed because we are only doing retrievals from these two data structures.

All of these data structures are passed into the newly created AnalyzerThreads, and they can finally begin their work. The thread starts by retrieving the word entry from the array using the Counter as the index. Then it adds the entry's count to the total number of words and checks if its count is one. If it is one, the thread adds one to the count of words that only appear once. Then the thread checks if the word is found in the HashSet of common English words. If it is, the thread calls check() on the CHMWrapper for common English words. Otherwise, the thread calls check() on the CHMWrapper for uncommon words. Then, if the user chose to upload custom words to check for at the start of the program, the thread checks the custom words HashSet for the current word. If the custom word is within the custom words HashSet, the program places that entry into the custom words HashMap. Finally, the thread increments the Counter and moves on to the next entry.

Once the AnalyzerThreads are finished executing, we take all of the data structures they worked on and place them into a WordData object which contains some behavior for outputting the results.

The data we collect within WordData and report to the user is as follows: the filename that was analyzed, the run times for the parsing and analyzing phases of the program, the total sum of the run times for both paring and analyzing phases of the program, the total number of words, the total number of 'unique' words (words not in the top one-hundred most common English words), and the number of words with a count of one.

If the user chose to input a list of custom words to search for and analyze within the text, the report given to the user also includes the most common words within the text that are also within the custom words list.

Regardless if the user chooses to input a custom words list, the report includes a list of the top ten most frequent common English words. This section of the report also includes the total count of those ten words most frequent, what percentage of the file's words they take up, what the words are, and how many times each of those words appear in the text in decrement order. The ten most common unique words are also reported alongside the number of times they appear in the text; where the common unique words are words very common in the text provided but not within the list of most common words in the English language.

All of this data stored within WordData is finally output to a file through a function within the WordData class. Additionally, a single-threaded implementation runs after the multi-threaded implementation and outputs its data report to a separate file for comparison.

B. Anticipated and Encountered Challenges

One problem we encountered was attempting to read files using multi-threading. By comparing the speed to a single-threaded implementation, we were able to conclude that something was not behaving the way we had thought. Through further research, we discovered that file reading is done on the disk, not the CPU. The disk does not inherently allow for multi-threading, so threads end up fighting over each other on who gets to read the disk at any given moment, leading to a large bottleneck. Therefore, multi-threaded file reading is not and will never be faster than a single-threaded implementation.

Additionally, our AnalyzerThread implementation involves the threads accessing a single WordData object which the threads call methods on to update the data inside the object. Since we are using Java's synchronized operator, we discovered that our current implementation is quite slow. This is due to the fact that Java inherently bounds a single lock to the entire object. Therefore, only one thread is able to access the entire object at a single time, whereas we had operated under the assumption this would only apply to a given method inside the object.

Since many AnalyzerThreads are constantly fighting for control over the WordData object, this implementation is more than likely experiencing many bottlenecks. We circumvented this issue by building all of the data objects inside of the AnalyzerThreads themselves, rather than accessing them from another object.

C. Synchronization

The ParserThreads do not require much complexity with synchronization. They utilize Java's ConcurrentHashMap for thread-safe operations and a Counter class. We created our own Counter class which will atomically maintain the specific index either thread needs to use to access their data. The Counter contains one method called getAndIncrement() with a synchronized block for the critical sections.

The AnalyzerThreads, on the other hand, required a bit more effort. After doing further research and learning the old implementation was wrong, we spent some time going over iterations as to how we could solve this issue. We ran into many race conditions while trying to search for the lowest value in a HashMap and replace it with a higher value. However, we were able to use our knowledge of why our old implementation was slow to give us insight as to how we could solve this problem.

Since we knew that only one thread could access a single object with a synchronized block no matter how many different blocks there are, we used this to come up with the idea of extending the ConcurrentHashMap. The ConcurrentHashMap already contained so much of the behavior we wanted, but we just needed to add special functionality for our specific use case. It took some experimenting but in the end, we were able to consistently match the parallel output with the sequential output.

V. TESTING

During the production of the text-analyzer program, we tested using smaller text files we created as well as larger text files found online through free online book databases. These text files contained letters, numbers, symbols, and punctuation. The files were mainly used to test the functionality of our program as opposed to gathering data on the differences in run time between our single and multi-threaded approaches.

As for testing the efficiency of both the single-threaded and multi-threaded approach when gathering data, we used large text files generated using a Lorem ipsum generator online. This generator created large pieces of text containing letters and punctuation. The output of the Lorem ipsum generator was then transferred to a text file and passed into the text analyzer as input for testing the run time of the application for a single-threaded approach as well as varying multi-threaded approaches of different thread counts.

The thread counts we tested when gathering data were 1, 4, 8, 16, and 32 threads. As for the number of words within each text file that we analyzed, we used input text files of varying different sizes for each of the thread counts we tested. The word counts for the text files we tested when analyzing the speed of our program with different thread counts include 100,000 words, 200,000 words, 400,000 words, 800,000 words, and 1,600,000 words.

To test our multi-threaded text analyzer, we implemented a single-threaded version to verify the output and compare run times with. Comparing our multi-threaded approach with a single-threaded version was also very helpful for tracking down race conditions and debugging. Having a single-threaded version of our program to compare to the multi-threaded implementation allowed us to debug and discover why some of our implementations were slow.

Through testing, we found that our multi-threaded implementation using eight threads gave the lowest run time overall (Figure 1). This behavior was observed on multiple machines that we tested. Additionally, we measured the run time differences for the sequential implementation and the implementation using eight threads, since that was the consistently giving the fastest times.

We also tested the performance difference between a single-threaded approach and our multi-threaded approach on files containing different word counts (Figure 2). The performance was tested on files with varying word counts to check if the multi-threaded approach was faster overall or if the multi-threaded approach was only faster for files with larger word counts.

As you can see in Figure 2, the single-threaded output's run time increases significantly as the word count increases. When comparing these results to that of the multi-threaded approach, the multi-threaded approach with eight threads increases only marginally.

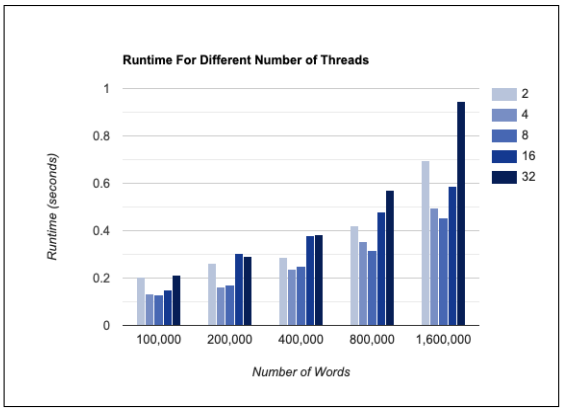

Fig. 1. How different numbers of threads impact runtime on increasing sizes of text data

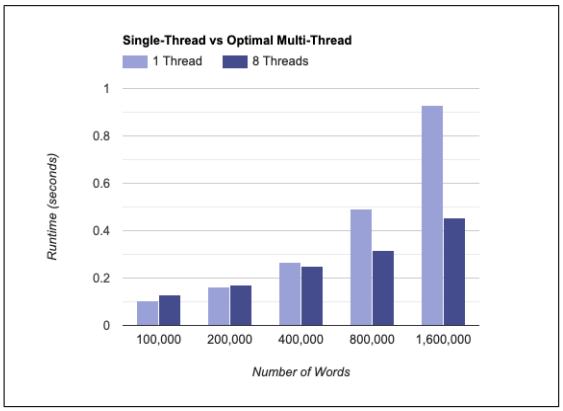

Fig. 2. Performance difference between our single-threaded implementation and optimal multi-threaded implementation.

A. Testing Environment

Our program was written in different environments among our team, however, all test data recorded was compiled and run using standard command line tools in a macOS environment to keep things consistent. These specific tests were conducted on a 2.3 GHz 8-Core Intel Core i9 with 16 GB of Memory. Each test was run multiple times to get the average amount of run time.

VI. EVALUATION

Our multi-threaded text analyzer can be much faster than a single-threaded implementation in some cases. A specific case where a multi-threaded implementation is seen to be more optimal than a single-threaded implementation is when a file with a large number of words must be parsed through. This is most likely due to there not being enough data to justify the overhead caused by the multi-threaded approach. It is important to note that the increase in speed is only really true for the parser phase when comparing both the single-threaded and multi-threaded approaches.

In regards to the approach when analyzing the text statistics, we observed that the single-threaded implementation was typically faster than the multi-threaded implementation. In our evaluation, we timed the parsing and analyzing phases of the program individually. When analyzing the parsing and analyzing phases separately, we were able to observe that the analyzing phase was slower most of the time for the multi-threaded approach. Through further inspection, we came to the conclusion that the analyzing phase was possibly slower within the multi-threaded approach due to the amount of overhead required in the multi-threaded analysis, and the low amounts of data it tends to receive.

We assume that if a file produced a very large number of entries, the input would be more viable to analyze in parallel within the multi-threaded implementation. In most cases, a file will not produce a large number of entries and this will not happen. This is unfortunate, however, we were still delighted to see large time differences with parsing.

In our evaluation, we also found that having too many threads in the text-analyzer program will cause too much contention, and having too few threads will cause the entire program to slow down. But overall, the multi-threaded text analyzer parsed through text files more efficiently using the multi-threaded implementation but analyzed text more efficiently with fewer threads.

A. Conclusion

This research paper presents the effects of implementing multi-threading in a text analyzer program in Java. Our results show that a multi-threaded implementation outperforms a single-threaded implementation at around 400,000 words and that 4 to 8 threads is the optimal number for achieving the best run time.

This research has practical applications in fields that require fast and efficient text analysis, such as natural language processing or data mining. However, our study is not without limitations, as it was conducted on a specific and limited data set and may not be generalizable to larger or more complex data sets. Future research could explore different thread pool configurations or evaluate the impact of different types of text analysis tasks. Overall, our study highlights the potential of multi-threading to improve the efficiency of text analysis programs in Java.

REFERENCES

[1] linguamatics.com. (2019). What Is Text Mining, Text Analytics and Natural Language Processing. [online]. Available: www.linguamatics.com/what-text-mining-text-analytics-and-natural-language-processing. [Accessed 9 Mar. 2023].

[2] Speak Ai. Free Text Analysis Tool. [online]. Available: speakai.co/tools/text-analysis-tool/. [Accessed 9 Mar. 2023].

[3] MAXQDA. Text Analysis Software | Powerful and Easy-To-Use. [online]. Available: https://www.maxqda.com/text-analysis-software. [Accessed 9 Mar. 2023].

[4] MeaningCloud. How Can You Use MeaningCloud?. [online]. Available: www.meaningcloud.com/how-can-you-use-meaningcloud. [Accessed 9 Mar. 2023].

[5] Daxtra. Parsing with Daxtra Capture. [online]. Available: info.daxtra.com/parsing-with-daxtra-capture. [Accessed 9 Mar. 2023].

[6] Online-Utility.org. (2020). Text Analyzer - Text Analysis Tool - Counts Frequencies of Words, Characters, Sentences and Syllables. [online]. Available: www.online-utility.org/text/analyzer.jsp. [Accessed 9 Mar. 2023].
[7] docs.oracle.com. ConcurrentHashMap (Java Platform SE 8). [online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html. [Accessed 9 Mar. 2023].
[8] Wikipedia. Most common words in English. [online]. Available: https://en.wikipedia.org/wiki/Most_common_words_in_English. [Accessed 31 Mar. 2023].

TABLE I

TOTAL RUNTIME OF SINGLED-THREADED VS. MULTI-THREADED IMPLEMENTATION

Number of Words	**Single-thread (seconds)**	**Multi-thread (seconds)**				
		Number of Threads				
		2	**4**	**8**	**16**	**32**
100,000	0.1033	0.2062	0.1331	0.1301	0.1483	0.2143
200,000	0.1611	0.2632	0.1626	0.1704	0.3027	0.2933
400,000	0.2669	0.2891	0.2379	0.2505	0.3811	0.3833
800,000	0.4907	0.4197	0.3541	0.3181	0.4797	0.5717
1,600,000	0.9311	0.6942	0.4938	0.4541	0.5855	0.9459