A Parallel Term-Frequency Inverse Document Frequency Implementation in C++

Parallel TF-IDF Classifier

Andrew Kelton
Undergrad Student,
Dept. Computer Science
University of Central Florida
Orlando, FL, USA
an597152@ucf.edu

***Abstract*—TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic that represents the significance of a word in a document relative to its corpus (collection of documents) it is included in. It is a widely used algorithm used for preprocessing in text classification. However, its normal, sequential, algorithm takes quite a long time to vectorize and compute TF-IDF scores across and entire corpus of documents. There are also not many TF-IDF implementations in C++, the most widely used TF-IDF implementations are in Python, while popular, is generally slower than C++ for computational tasks. In this project, I present a parallel TF-IDF vectorization, computation, and classification implementation written in C++. The goal of this project is to provide an efficient and quick TF-IDF algorithm, and to compare my parallel implementation to my sequential implementation. I was successful in creating the parallel TF-IDF vectorization, computation, and classification. Source code is included in this project, and the performance evaluation in Python is included as well.**

***Keywords*— C++, classification, computation, efficient, parallel, Python, quick, sequential, TF-IDF, vectorization**

I. Introduction

Natural language processing (NLP) is a vital field in today's world. One of the key techniques used in NLP is Term Frequency-Inverse Document Frequency (TF-IDF), which helps identify important words within a document, while filtering out less relevant words. Search engines such as Google utilize TF-IDF to rank web pages, filtering billions of somewhat relevant words to ensure users receive the most meaningful results. However, computing TF-IDF at this scale is extremely computationally expensive, especially when processing large datasets, such as internet search results.

A common approach to handling this challenge is to precompute and store TF-IDF scores in a database. However, maintaining such a vast database is resource intensive. An alternative solution is to compute TF-IDF dynamically when needed. Unfortunately, most existing implementations of TF-IDF are sequential and written in Python, which limits their efficiency and performance, especially for large-scale applications.

This project focuses on implementing a parallelized TF-IDF classifier in C++ to significantly improve computation speed and efficiency. It has been shown that C++ can be up to 100 times faster than Python for certain computational tasks due to its optimized memory management and lower-level control over hardware resources. By leveraging parallel computing, this implementation aims to further enhance TF-IDF performance by distributing the workload across multiple processing cores.

Beyond search engines, TF-IDF is widely used in spam detection, sentiment analysis, and document clustering, where faster computation can lead to real-time processing improvements. Traditional implementations suffer from bottlenecks in matrix operations, memory usage, and sequential execution, which can be mitigated using optimized C++ techniques. This project will integrate the Armadillo library, a highly efficient linear algebra library, to handle matrix operations and accelerate TF-IDF computation.

By implementing TF-IDF classification in C++ with parallelization, this project aims to demonstrate significant performance improvements over traditional approaches. The expected outcomes include faster execution times, efficient memory usage, and scalability for large datasets, making this approach highly suitable for real-world NLP applications

(Keeping these two paragraphs for later reference) The job of TF-IDF is to filter out these somewhat relevant words, whilst keeping the relevant ones accessible. However, keeping a database of TF-IDF scores for every item on the internet is extremely resource expensive. One solution is to compute the TF-IDF every time. This is also extremely costly, since most common TF-IDF implementations are sequential and in Python. In this project, I will focus on the implementation of a parallel TF-IDF in C++.

It has been shown that C++ can be up to one hundred times faster than Python when performing some computing tasks. Not only can the parallel TF-IDF be quicker and more efficient than traditional implementations of the TF-IDF classification, but the sequential implementation in C++ can also increase efficiency and performance times as compared to the traditional implementation.

A. Problem Statement

I have achieved the main goal for this project: Implement a parallel Term-Frequency Inverse Document Frequency

(TF-IDF) text classifier in C++ and benchmark it against my sequential implementation of a TF-IDF text classifier in C++. I will be using the sequential TF-IDF implementation as my control. As far as I know, there is no publicly available parallel TF-IDF implementation in C++. With this project I seek to be able to implement key components of the commonly used text vectorization and classification algorithm into an efficient parallelized, C++ implementation. Once I have equivalent implementations (sequential and parallel), I will run tests on the two different implementations to see if there are any performance, efficiency, accuracy, or precision differences between them. I will be looking mainly at speed and efficiency, and briefly discussing text classification accuracy and precision.

II. Motivation

In my research for this project, I could not find any parallel TF-IDF implementations for articles or text written in C++, either in literature or production. It surprised me that I could not find any strictly parallel TF-IDF implementations for text available in C++. While I would have appreciated the opportunity to test my parallel TF-IDF algorithm against others that are widely used, no such implementation appear to exist. The opportunity to review a similar implementation could have given insight on the key differences between a strictly parallel TF-IDF in C++ and user-level parallel TF-IDF techniques normally found in Python. Although this lack of existing work is disappointing, it confirms the novelty and significance of this project.

An explanation for this gap in C++ implementations, is the dominance of existing Python-based data science and machine learning libraries. Libraries such as Scikit-learn in Python have created specific functions that allow users to implement parallelism in their code, however, they do not provide a strictly parallel TF-IDF implementation. These implementations have often been criticized for not being thread-safe and not providing full control for concurrency. This requires users to explicitly optimize and control shared memory, while maintaining thread-safety. In contrast, C++ offers low-level access and control over parallel processing, making it a strong candidate for developing high-performance, large-scale text classification solutions.

The absence of scalable parallel TF-IDF implementations in C++ also suggests an opportunity to contribute a scalable, efficient solution that can be used freely by C++ programmers. This work aims to bridge that gap, demonstrating the feasibility and benefits of parallelizing TF-IDF in C++ for text vectorization and classification.

In the future, I plan to incorporate more functions that allow users to view more data resolved from the TF-IDF vectorization.

III. Related Work (NEED TO DO THIS)

NOT WRITTEN YET.

IV. Implementation

The following is the breakdown of the tasks I completed on this project:

- Write the C++ implementation for reading, splitting, and extracting document data from CSV files
- Implement tasks for preprocessing data
  - Skip stop words (common words such as *the*, *is*, etc. that do not contribute importance to a document)
  - Remove punctuation
  - Convert all letters to lower case
  - Stem words using the *Oleander Stemming Library*
- Implement the functions to vectorize a corpus of documents
  - `vectorize_corpus_threaded`
  - `vectorize_corpus_sequential`
- Implement modules containing classes and namespaces for the representations of:
  - Corpus
  - Document
  - Category
- Implement the functionality of each major TF-IDF function in C++
  - `tfidf_documents`
  - `tfidf_documents_seq`
  - `calculate_term_frequency_doc`
- Implement the functionality of major text classification functions in C++
  - `get_single_cat_par`
  - `get_single_cat_seq`
  - `get_important_terms`
  - `classify_text`
- Create a testing environment for extracting benchmarks, performance, accuracy, and precision results
  - Develop Python and Bash scripts to automate extraction comparison between the two implementations and generate human readable results.
- Compare benchmarks between the two implementations
  - Measure performance amongst sections
  - Classification accuracy and precision
  - Test multiple untrained corpuses with trained corpuses

A. Plan for Implementation

To summarize the above sections, my goal is to implement a parallel TF-IDF vectorizer and classifier for text processing in the C++ programming language. I plan to compare both my C++ parallel and sequential implementations. I am primarily interested in how my parallel implementation compares to the sequential implementation with regards to performance and accuracy, but I am also interested to see if any major changes to design between the two are essential. The goal is to provide a parallel C++ TF-IDF implementation for use in general NLP programming. My goal is to allow users to use my library without using C++'s parallel TF-IDF in `mlpack` or switching to Python. I first began by creating the structs and classes for the Corpus, Document, and Category objects and the descriptors for their tasks and uses. The following are essential user-facing operations required to use the TF-IDF implementation on a trained (category type is known) corpus:

- `read_csv_to_corpus`
- `vectorize_corpus_threaded/sequential`
- `corpus.tfidf_documents/_seq`
- `get_all_cat_par/seq`

After each function or section completes, I record performance benchmarks for each. I record the amount of time in milliseconds the operation took for both the sequential and parallel implementations. After completing these functions, the following functions are user-face operations required for untrained data, where categories are saved in a separate file:

- `vectorize_corpus_threaded/sequential`
- `corpus.tfidf_documents/_seq`
- `init_classification_par/seq`
- `print_classifications`

After each function or section completes, I record performance benchmarks for each. I record the amount of time in milliseconds the operation took for both the sequential and parallel implementations. After all functions complete, I record benchmarks for accuracy and precision of the classification of each document. I compare the execution times of each section and the classification's accuracy and performance between the parallel and sequential implementations.

B. Anticipated & Encountered Challenges

I anticipated the following major challenges with implementing this project: C++'s non-memory safe design, the lack of similar multi-threaded implementations, and my lack of experience with C++.

Since I used this project as an opportunity to learn C++, I anticipated some difficulty would come from learning the language, however, I was quite captivated by C++'s Java, C mixture. This is my first major project I have undergone in C++, and I have learned a lot about its object-oriented programming (OOP) capabilities. I became quite acquainted with the language's strengths, limitations, and code design.

C++'s design prioritizes performance and flexibility, but this comes at the cost of memory safety and built-in concurrency primitives. Though these traits allow fine-grained control over system resources, they also introduce challenges when implementing a multi-threaded system. A major issue I encountered was managing shared data across threads. Unlike higher-level languages with built-in memory safety mechanisms, C++ requires explicit handling of synchronization and resource management. Simply sharing a data structure across multiple threads is not straightforward, as concurrent access must be carefully controlled to prevent data races and undefined behavior.

One common approach is to use `std::mutex` to lock shared resources, ensuring thread safety at the cost of potential contention. However, using mutexes alone is not always ideal for high-performance applications, as they introduce blocking behavior. An alternative is `std::atomic`, which enables lock-free programming by allowing atomic operations on shared data. Despite these tools, I found that designing an efficient and safe multi-threaded system in C++ requires a deep understanding of low-level memory management and synchronization mechanisms. Thankfully, my extensive experience with the C programming language in systems programming, proved quite helpful when coming across sections that required meticulous memory management.

However, this is just one example of the increased complexity required to implement a parallel TF-IDF while adhering to C++'s memory management requirements. Encountering more situations like this was my greatest concern for this project, but I figured worst-case scenario I use a bit of C style programming in the C++ language. These tools are designed to avoid undesired results such as garbage values or invalid pointers.

C. More Challenges & Significant Changes to Design

Asdjnfjk;a

V. Testing

To accurately compare the results against the sequential C++ implementation, I decided to run the same trained and untrained documents on the parallel C++ implementation. For example, I start a counter for execution time of each section and the instant the function for each section returns, the counter stops, and execution time is reported in milliseconds. Below I will show the results when analyzing performance and accuracy.

A. Testing Environment

All code was written, compiled, and tested using C++ standard 17 in an Ubuntu Linux environment. This provided a standard across the board to avoid encountering OS specific bugs or delays in compilation and computation.

The main test files were written to be mostly one-to-one from each other, even though one is a parallel implementation while the other is a sequential implementation. The test is conducted by calling respective functions for each section in their main file. They both take in a file which is the trained data file.

The runner is a Makefile which compiles and executes the parallel implementation, then the sequential implementation. The command `make test` will compile and execute both the main test files with the preset trained data file and output the data into its respective files in the test-outputs folder.

B. THIS POINT ON IS PART OF A TEMPLATE

The template is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

1) For papers with more than six authors: Add author names horizontally, moving to a third row if needed for more than 8 authors.

2) For papers with less than six authors: To change the default, adjust the template as follows.

a) Selection: Highlight all author and affiliation lines.

b) Change number of columns: Select the Columns icon from the MS Word Standard toolbar and then select the correct number of columns from the selection palette.

c) Deletion: Delete the author and affiliation lines for the extra authors.

C. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced. Styles named "Heading 1", "Heading 2", "Heading 3", and "Heading 4" are prescribed.

D. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 1", even at the beginning of a sentence.

TABLE I. TABLE TYPE STYLES

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

We suggest that you use a text box to insert a graphic (which is ideally a 300 dpi TIFF or EPS file, with all fonts embedded) because, in an MSW document, this method is somewhat more stable than directly inserting a picture.

To have non-visible rules on your frame, use the MSWord "Format" pull-down menu, select Text Box > Colors and Lines to choose No Fill and No Line.

[a] Sample of a Table footnote. (*Table footnote*)

Fig. 1. Example of a figure caption. (*figure caption*)

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT *(Heading 5)*

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

The template will number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] K. Eves and J. Valasek, "Adaptive control for singularly perturbed systems examples," Code Ocean, Aug. 2023. [Online]. Available: https://codeocean.com/capsule/4989235/tree

[9] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," 2013, arXiv:1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114

[10] S. Liu, “Wi-Fi Energy Detection Testbed (12MTC),” 2023, gitHub repository. [Online]. Available: https://github.com/liustone99/Wi-Fi-Energy-Detection-Testbed-12MTC

[11] “Treatment episode data set: discharges (TEDS-D): concatenated, 2006 to 2009.” U.S. Department of Health and Human Services, Substance Abuse and Mental Health Services Administration, Office of Applied Studies, August, 2013, DOI:10.3886/ICPSR30122.v2

IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove template text from your paper may result in your paper not being published.