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

Andrew Kelton
Dept. of Computer Science
University of Central Florida
Orlando, 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 prepro-cessing in text classification. However, its normal, sequential, algorithm takes quite a long time to vectorize and compute TF-IDF scores across an entire corpus of documents. There are also not many TF-IDF implementations in C++, the most wide-ly used TF-IDF implementations are in Python, while popular for prototyping and data analysis, is generally slower than C++ for computational tasks. In this project, I present a paral-lel TF-IDF vectorization, computation, and classification im-plementation written in C++. The goal of this project is to provide an efficient and quick parallelized TF-IDF vectorizer and classifier in C++, 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.**

***Index Terms*—C++, classification, computation, efficient, parallel, Python, quick, sequential, Term Frequency-Inverse Document Frequency (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 pre-compute 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 [4], 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.

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.

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 appears 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 [4]. 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

As explained above, TF-IDF is a widely used technique to evaluate the importance of words in documents.

A. An Overview of TF-IDF

The following is the mathematical equation of the TF-IDF algorithm, where the TF-IDF score for word t in the document d is defined as:

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

Where

$$TF(t,d) = \frac{\text{number of times } t \text{ appears in } d}{\text{total number of terms in } d}$$

$$IDF(t) = \log\left(\frac{N}{1+\text{freq}(t,d)}\right)$$

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 modules containing classes and namespaces for the representations of:
  - TFIDF
  - Corpus
  - Document
  - Category
- 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 `english_stem` class from the Oleander Stemming Library [5]
- Implement the functions to vectorize a corpus of documents.
- Implement the functionality of each major TF-IDF function in C++.
- Implement the functionality of major text classification functions in C++.
- 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 test 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 sequential TF-IDF from mlpack [3] 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 the three main sections of the program that will differ significantly between the sequential and parallel implementation.

- Vectorization
- TF-IDF Computation
- Categorization

When using a trained corpus, I record performance benchmarks for each section. For example, the timer starts when the Vectorization section begins and ends when the Vectorization section completes. For an test corpus, I record performance benchmarks as a whole, including all sections and classification time in one recording. Execution times are recorded in milliseconds and begin and end recording in the same blocks in both implementations.

After all sections complete, including classification (for an test corpus), I record accuracy and precision of the classification for each document classified. I compare execution times of each section, classification accuracy, and classification precision between the parallel and sequential implementations.

B. Classification of Text

test text is classified using cosine similarity of the TF-IDF scores of the test text, d_i, and a category's combined TF-IDF scores, c_j, whose TF-IDF scores have already been calculated [1] and d_i is a document in the test corpus and c_j is the entire `TFIDF` of normalized TF-IDF scores of all documents in a `Category`. The cosine similarity approach used in this work is largely based on the methodology outlined by Park, Hong, and Kim in [2], due to the findings of cosine similarity's significance in using TF-IDF scores to classify text. The approach presented in this work differs from that of Park, Hong, and Kim in that it computes the cosine similarity between a single document and a combined TF-IDF score vector for a category, rather than computing cosine similarity across all documents. Park Hong and Kim's approach is given by:

$$\begin{aligned} \mathit{Sim}(doc1, doc2) &= \frac{doc1 \cdot doc2}{\|doc1\|\|doc2\|} \\ &= \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \end{aligned} \tag{1}$$

where A_i and B_i represent the components of vectors *doc1* and *doc2*, respectively [2].

While the cosine similarity equation used in this report is as follows:

$$\begin{aligned} \mathit{Sim}(d_i, c_j) &= \frac{d_i \cdot c_j}{\|d_i\|\|c_j\|} \\ &= \frac{\sum\limits_{t \in d_i \cap c_j} d_i(t) \cdot c_j(t)}{\sqrt{\sum\limits_{t \in d_i} d_i(t)^2} \cdot \sqrt{\sum\limits_{t \in c_j} c_j(t)^2}} \end{aligned} \tag{2}$$

where $d_i(t)$ and $c_j(t)$ denote the TF-IDF weights of term t in document d_i and category c_j, respectively. The summation $\sum_{t \in d}$ iterates over all terms t in the given document, and the intersection $d_i \cap c_j$ includes only the terms shared by both documents. The threshold $\varepsilon = 10^{-9}$ ensures stability by avoiding division by zero when either document has no weighted terms.

The methodology of classifying an test document is as follows:

$$\hat{y} = \arg\max_{1 \le j \le 5} \mathit{Sim}(d_i, c_j) \tag{3}$$

where it iterates through all 5 indices of `Category` object vector and saves the largest cosine similarity value each time, as that will be related to its classifed category. The equation iterates through 5 indices of the `Category` object vector since this project is currently limited to 5 classification categories that are stated in [6]. The following classification categories are supported in this project in sequential order:

1) Sports
2) Business
3) Politics
4) Tech
5) Entertainment

C. 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 resemblance of Java and C. 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. In the worst case, I relied on C-style programming techniques. These tools are designed to avoid undesired results such as garbage values or invalid pointers.

D. More Challenges & Changes to Design

When testing the code, I noticed it was quite difficult to understand exactly where errors where occurring, even after implementing `try: catch:` blocks within the source code. To mitigate these issues and increase ease of usability, I decided to create the `TFIDF` namespace and `TFIDF_` class.

The `TFIDF_` class is a user-side class that takes input on numerous variables, such as: number of threads to use, print output, print errors, file inputs, and file outputs, just to name a few. This greatly increased the code-readability, reusability, and effectiveness of recording metrics amongst multiple inputs. The class contains three functions:

- `process_trained_data`
- `process_un_trained_data`
- `process_all_data`

These functions allow the user to either process trained and test data separately, or process trained data then process test data by only calling one function. When testing the code with multiple datasets and varying thread counts, `process_all_data` is called to provide a standard for code usage and recording of benchmarks.

Another major issue I became aware of when first testing the program, was the extreme inaccuracy of category classification when utilizing multiple threads. This issue was mainly due to the fact I was not ensuring all threads maintained the same data across all threads. I would lock the `vector` that contains the final list of `Category` objects when pushing a new `Category` but would not lock or ensure proper synchronization when categorizing data. Therefore, `Category` objects contained incorrect document indexes, since their chosen index had changed due to another thread modifying the `vector` of documents. I was able to fix this issue by locking the entire program when a thread is actively creating a `Category` object. However, this solution is not efficient and allows for long-blocking periods, making it essentially sequential with a minor decrease in performance, due to thread creation, deletion, and idle-waiting. In the future, I hope to implement a queue and worker threads to efficiently dequeue *Document* objects and classify them to their correct category without inefficient blocking or waiting.

E. Synchronization

The approach for my implementation requires support of locking mechanisms and atomic operations. To avoid data races, whenever pushing or emplacing to a `TFIDF`, I lock the `TFIDF` and ensure only one thread updates the shared `TFIDF` at a time. This design guarantees only one thread updates the `TFIDF` at a time, while also ensuring a short critical section to avoid long blocking. Atomic operations are used when classifying our test data and incrementing the total and correct count of classifications. This approach guarantees all threads contain the same data throughout the entire TF-IDF process.

My implementation also ensures proper thread balancing, avoiding scenarios where some threads have completed their tasks, while others have just begun. This is achieved by splitting the dataset into roughly equal chunks based on the number of available threads we are testing with as seen in Algorithm 1 . Each thread is assigned a portion of data to: vectorize, compute TF-IDF scores, and classify test data.

Additionally, thread synchronization is cautiously managed to ensure threads are not being blocked by long critical sections. The task distribution is designed so that no thread holds up others for extended periods of time. Immediately after updating a critical section that uses locks, threads resume their tasks and remain in sync containing the same data. This approach not only ensures efficient workload, but also efficient use of resources, minimizing idle time and maintaining overall performance.

V. Testing

To accurately compare the results against the sequential C++ implementation, I used the same original dataset [6] and split it into 80% training data and 20% testing data. This ensured consistent input across all versions of the program. As discussed in Section IV, I measured execution time in milliseconds and recorded the classification accuracy and precision. Below I will present the results of performance, accuracy, and precision analysis.

A. Testing Environment

All code was written, compiled, and tested using C++17 on the University of Central Florida's high-performance Secure Shell Protocol (SSH) server, *eustis3*. Eustis3 runs Ubuntu 22.04 LTS with Linux kernel version 5.15.0 and is equipped with dual-socket Intel$^{®}$ Xeon$^{®}$ Silver 4216 CPUs, totaling 64 threads (2 sockets $\times$ 16 cores $\times$ 2 threads per core). The system provides 187 GiB of RAM, of which approximately 58 GiB were free and 75 GiB available during testing. This setup offered sufficient parallel processing capabilities for benchmarking the proposed algorithm while maintaining a consistent Linux-based development environment to minimize OS-specific inconsistencies.

The main test files were written to be mostly one-to-one from each other. The only differences in the main test files are the inputs when creating the `TFIDF_` object. The tests are conducted by inputting the respective number of threads for the `TFIDF_` object to handle. All testing files use the same datasets: *dataset-1*, *dataset-2*, and *dataset-3*, which consist of collections of categorized BBC News articles of varying lengths and complexities. These datasets were sourced from [6]

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.

To run multiple tests of all numbers of threads and the sequential test, I used a Bash script. The script runs each testing scenario ten times before testing the next scenario. This allows shorter testing scenarios to run and complete before testing longer scenarios, such as the sequential implementation, which may take upwards of twenty minutes to complete per test. Figure 1 is an example of what a block in the Bash script looks like when testing the code with two threads. This script was used to gather ten iterations of each test across the three datasets.

```
NUM_ITERATIONS=10

for ((i=1; i<=$NUM_ITERATIONS; i++)); do
    make 2-test
done
```

Fig. 1: Example block of bash testing script to test 2 threads.

VI. Evaluation

Once both the sequential and parallel TF-IDF implementations were finished in C++, I could analyze the differences in performance, accuracy, and precision between the two. For this paper, I will be using the results from dataset-3, which is the largest dataset used and the most likely to be used in a real-world scenario.

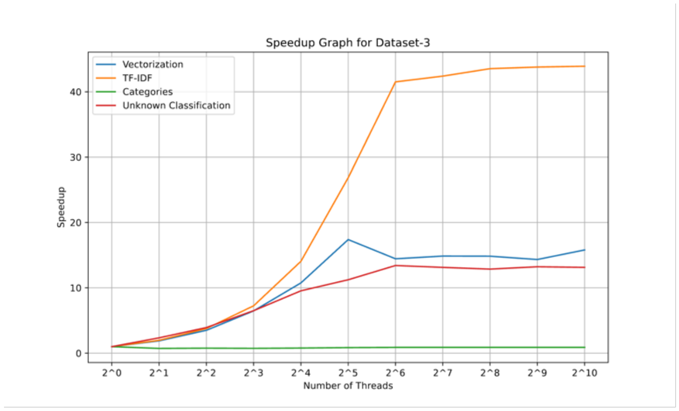

Fig. 2: Speedup results from 1–1024 threads.

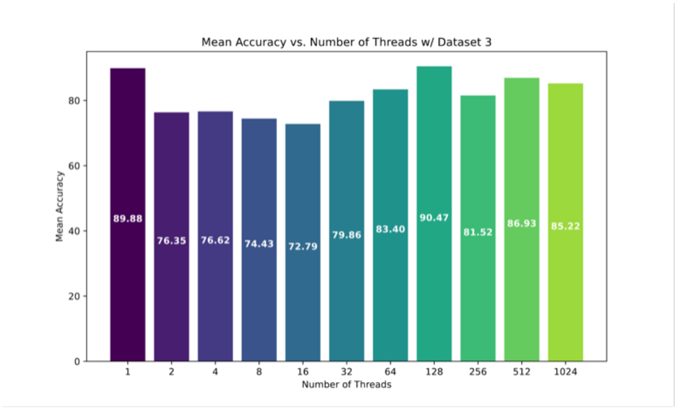

Fig. 3: Classification accuracy results from 1–1024 threads.

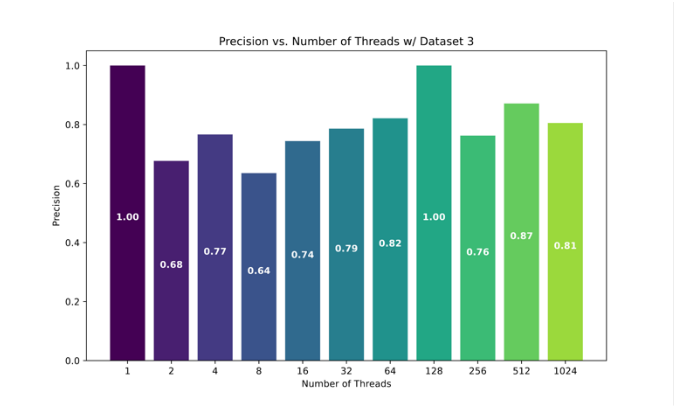

Fig. 4: Classification precision results from 1–1024 threads.

In figure 2, there is a dramatic change in performance from 1 to 64 threads, and then it fizzles out, remaining close to or worse than before. However, the Categories performance remains similar amongst all threads. This is due to the way the Categories are updated between the sequential and parallel implementations. As stated earlier, the parallel implementation for the Categories section is essentially sequential by locking the entire corpus object before completing categorization tasks. In the future, I hope to fix this issue by implementation a conccurent queue for categorization tasks.

Figure 3 illustrates the mean classification accuracy of 10 iterations across all tests. As we can see, there is a dramatic drop in accuracy between 2 and 32 threads. This may suggest an issue with data races when classifying test data, as trained data values and objects across all threads remains the same. However, the test utilizing 128 threads maintains the highest accuracy percentage of 90.47%, followed by the sequential implementation with an accuracy percentage of 89.88%.

Figure 4 illustrates the classification precision of 10 iterations across all tests. To clarify what precision is specifically for this paper, precision is the consistency of classification results across multiple runs. Therefore I am recording the ability to reproduce the same results across 10 iterations of the same input data. As we can see, there is once again a dramatic drop in precision between 2 and 32 threads. This only confirms my suspicions of data races occuring, but being more prevalent between tests with threads 2-32. If there was no data races, the precision would be 1.00 or close to 1.00, as the data given to my cosine similarity function would be the same every time. However, besides the drastic differences in precision between threads 2-32, we can see the sequential test and the test with 128 threads have a precision of 1.00. These tests generate the same results every single time across 10 iterations.

It is also worth noting that the sequential implementation, on average across all 10 tests, took approximately 7.8 minutes to complete all tasks, while the implementation utilizing 128 threads took an average of 2.3 minutes to complete all tasks. These two configurations were chosen for direct comparison because they were the only ones that consistently achieved a precision of 1.00, indicating deterministic behavior and the absence of data races. Furthermore, both tests yielded the highest classification accuracies across all tests, with the sequential version achieving 89.88% and the 128-thread version achieving a slightly higher accuracy of 90.47%. This comparison demonstrates that the sequential implementation is about 3.4 times slower than my parallel implementation with 128 threads, which not only maintained precision but marginally improved classification accuracy.

A. Conclusion

I present a parallel TF-IDF implementation in C++. I discovered significant performance improvements in the parallel implementation of the sequential implementation. While I also discovered some data race concerns amongst a certain number of threads, I also discovered utilizing 128 threads was optimal for performance, accuracy, and precision. My implementation uses both atomic operations and locking mechanisms, such as mutex locks. It allows for the classification of test data, provided there was trained data given. I compared the parallel implementation to my sequential implementation using custom test scripts and found the optimal number of threads across all tasks for performance, classification accuracy, and precision was 128 threads. I plan to further develop this project and incorporate more functions and usages to further expand C++'s usage in modern NLP.

References

[1] G. Juraev and O. Bozorov, "Using TF-IDF in text classification," AIP Conference Proceedings, vol. 2789, pp. 050017—050017, Jan. 2023, https:://doi.org/10.1063/5.0145520.

[2] K. Park, J. S. Hong, and W. Kim, "A Methodology Combining Cosine Similarity with Classifier for Text Classification," Applied Artificial Intelligence, vol. 34, no. 5, pp. 396–411, Feb. 2020, https://doi.org/10.1080/08839514.2020.1723868.

[3] R. R. Curtin *et al.*, "mlpack 4: a fast, header-only C++ machine learning library," *Journal of Open Source Software*, vol. 8, no. 85, p. 5011, 2023, https://doi.org/10.21105/joss.05011.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] B. Madden, "Oleander Stemming Library," May 19, 2024. Accessed Feb. 11, 2025. [Online]. Available: https://github.com/Blake-Madden/OleanderStemmingLibrary.

[6] B. Bose, "BBC News Classification," 2019. Accessed Jan. 2025. [Online]. Available: https://kaggle.com/competitions/learn-ai-bbc.

Algorithm 1 loads x documents to a thread

```
1: n_docs_th = n_docs / num_ths
2: n_last = n_docs % n_docs_th
3: n_last += n_docs_th
4: if $n\_docs\_th == 1$ then
5:     num_ths_used = n_docs
6: else
7:     num_ths_used = num_ths
8: end if
9: for $i = 0$ to $n\_docs$ step $n\_docs\_th$ do
10:     create_thread(function()
11:     if $i = n\_docs - n\_docs\_th$ and $n\_last > 0$ then
12:         for $x = 0$ to $n\_last$ do
13:             if $x + i < n\_docs$ then
14:                 function($documents[x + i]$)
15:             end if
16:         end for
17:     else
18:         for $x = 0$ to $n\_docs\_th$ do
19:             if $x + i < n\_docs$ then
20:                 function($documents[x + i]$)
21:             end if
22:         end for
23:     end if
24:     )                                   ▷ End of lambda function
25: end for
```