Word Frequency Calculator in Rust

By Eva Gao and Ariel Traver

Problem Description:

Word frequency counters are one of the most common questions in software engineering interviews. Potential candidates are challenged to calculate the frequency of each unique word in a stream of text. Although the problem seems simple at first glance, many different solutions may arise. The simplest approach is to loop through the whole file whenever a new word is encountered, tallying its occurrences. This approach would be extremely inefficient when applied to a large text file. A smarter approach would use only one for-loop and a hashmap or hash table to save words and their counts. However, many computers possess multiple processors, and time may also be wasted waiting for I/O requests to complete. Therefore, the most efficient solution is to divide up the work across several threads. We choose to implement a Word Statistic Calculator in Rust, because Rust prevents concurrency bugs such as data races while providing powerful low-level control.

Why Rust:

Rust's strict type system equips it with failproof security features while still allowing low-level data manipulation. Mutability rules and sizing laws help prevent memory corruption and runtime errors [2]. Additionally, Rust enforces firm ownership rules to avoid data races and excess copying [2]. Rust does not allow a non-static data or data without locks to be shared by the thread. Each scope owns a value and once the ownership has been transferred into another scope or thread, the variable cannot be used again. The borrowing feature allows simple primitives to be shared, but it will not accept values too large to pass without being copied. Furthermore, shared data must live longer than the threads to ensure that no threads outlive the stack frame that called them and access invalid sections of memory. Finally, Rust also ensures that the function calling the thread waits for the thread to complete before terminating. All of these rules prevent data races from occurring so that the result is valid. The Rust compiler provided helpful suggestions, which helped our team program multiple threads.

Design:

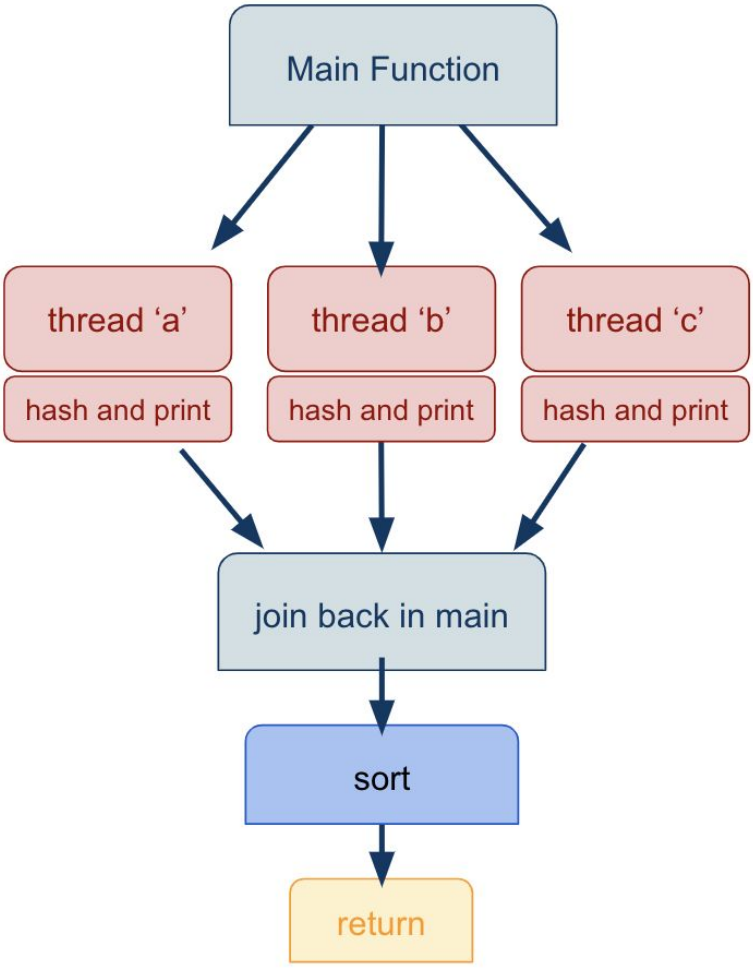
When approaching this problem, we first had to decide how to divide the work into threads. We considered splitting the file into sections, but we realized that data races might occur if two threads attempted to save the same word into a shared table. We thus settled on giving each thread a range of starting letters. This prevents two threads from ever hashing the same string.

Our function first loads the file contents into a static string, cleaning out any punctuation marks or newlines along the way. It then runs a while-loop across ranges of letters in the alphabet, using a u8 number to represent the unicode character of each one. The while-loop spawns a thread for each range, which has its own hashmap. Within each thread, there is also a for-loop iterating over the text. The thread only stores values in the hashmap when the unicode value of the first character is within its unique range. Once the for-loop over the text is complete,

the thread stores its findings in a vector of tuples shared between all threads. This vector is sorted and printed once all threads are complete.

The limitations of our program are that it can only process a single text file at time and also that our program does not check the validity of the words. Any string of non-punctuation characters count as a word, so our program's accuracy and meaningfulness depends heavily on the accuracy of the input text file. Another limitation is that our program does not include words which begin with accented letters such as ‘å’, hyphens, number, or non-english characters. To solve this problem, we decided to design an option that allows a user to include these if he/she wants. This extra work will be appended to the first thread.

Figure 1: Flowchart of the steps in Word Frequency Calculator



Implementation:

We implemented four helper functions in addition to the main. The main function calls the first function prepare_buff first saves the contents of a single text file as a string and then calls remove_chars() to replace the hard-coded punctuations with spaces. We choose to hard code the punctuations, because we only want to replace a limited subset of punctuations.

Figure 2 Code Example for replace_chars()

```
/// replace_chars
/// Cleans out pesky punctuation and newlines
/// # Arguments
///     * buff: the String to be cleaned
/// # Returns
///     * newbuff: a copy of the original without punctuation except spaces
fn replace_chars(buff:String)->String {
    let v = vec![',','\"','.','!','?','(',')','{','}',':',';','. ',', ','\\', '/','_','"','"', '|', '+', '-'];
    let mut new_buff = String::new();
    let iterator = buff.chars();
    for ch in iterator {
        if v.contains(&ch) {
            new_buff.push(' '); //don't push punctuation in
        }
        else if ch == '\n' {
            new_buff.push(' ');
        }
        else {
            new_buff.push(ch);
        }
```

The function prepare_buff then saves the cleaned string into a special static variable called TEXT_LOCK. TEXT_LOCK is a lazy_static combined with read-write lock (Fig 3). Because lazy_static variables create global static variables that can be initialized at run-time and a read-write lock allows for multiple readers, TEXT_LOCK increases CPU efficiency as no thread has to wait for the other thread to finish [7,8].

Figure 3: Code example for static variables

```
// creates a global instance of the text which can be shared with threads
lazy_static! {
    pub static ref TEXT_LOCK: RwLock<String> = RwLock::new(String::new());
    pub static ref RESULT_VECTOR: Mutex<Vec<(String, i32)>> = Mutex::new(vec![]);
}
```

After preparing the text, the main function calls calculate_word_count to calculate word count by spawning threads. Calculate_word_count takes a parameter range to determine the range of starting letters assigned to each thread. Instead of normal Rust threads, we use “scoped threads” from the package Crossbeam, which are guaranteed to terminate at the end of a scope we specify. Inside the scope, a for-loop calls the threads using an ARC counter i, which is cloned such that each thread can take its own copy. We initially delegated one thread for every letter, but this generated 26 separate threads which ended up harming performance on our dual and four-core processors. Additionally, because the distribution of starting letters in English is fairly uneven, this is not very efficient resource use as some threads will likely be idle. We discovered that two threads worked best on the dual-core processor and four on the quad-core. This number can be changed using the “range” parameter passed into calculate_word_count.

Figure 4: Code Example of calculate_word_count

```
/// calculate_word_count
/// Spawns threads for each range of starting letters
/// Each thread has its own hash table.
/// Threads print out their results once their calculations are complete
/// # Arguments
///     * range0: the range of u8 (unicode) starting letters per thread
fn calculate_word_count(range0:u8, mut extras:bool){
    let mut i:u8 = 97; //u8 for the character 'a'
    crossbeam::scope(|s| { //threads guaranteed to join before this scope ends
    while i <= 122 { //u8 for the character 'z'
        let icopy = Arc::new(i); //u
        let range = Arc::new(range0);
        let extras_not_done = Arc::new(extras);
        s.spawn(move |_| { //spawn a thread and move variables in
            let mut hmap = HashMap::<String,i32,>::new(); //new map for thread
            //let mut hmap = FnvHashMap::with_hasher(Default::default());
            let buff = TEXT_LOCK.read().unwrap(); // acquire the lock for reading
            let cursor = buff.split_whitespace(); //break by spaces
            for current in cursor {
                let first = current.chars().next().unwrap() as u8;
                //the extras_not_done determines whether random non-letters should be
ted
                if (first >= *icopy && first < *icopy + *range )
                   | (*extras_not_done && (first < 97 || first > 122))
                { //only look at certain letters
                    if hmap.contains_key::<str>(&current) {
```

For each thread, the program initializes a HashMap<String, i32> to store words and frequency counts. We originally initialized a vector of HashMaps for each thread in the calculate_word_count function; however, Rust's ownership laws prevented other threads from using the shared vector once the ownership was passed to the first thread. The thread then iterates through every word in the text string, split by spaces using the iterator String:split_whitespace(). Before the thread terminates, we write the key and value pair as a tuple into another lazy_static mutex containing a vector. After all of the threads are finished, we call another function, analyze_results, which unlocks the vector and sorts it by String in alphabetical order using Rust's built-in lambda function. Finally, the sorted list is printed to the screen. We recognize that sorting will cause extra overhead, so our tests are done on a version of our code that only prints the results. Because of Rust's strict ownership rules, we were considering writing the HashMap to a file for each thread; however, we decided against this approach because file writing would involve many I/O to the disk which would increase the overhead.

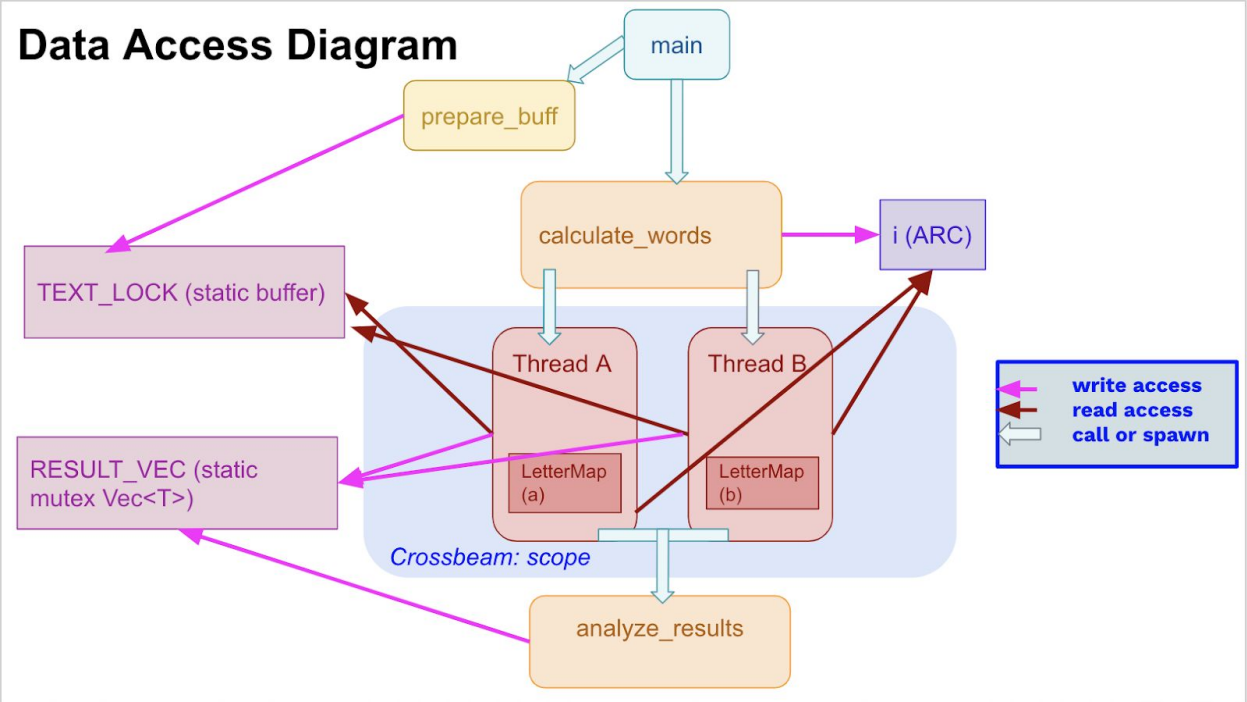
To account for words which begin with special characters, we included a boolean parameter called "extras" in calculate_word_count. This option allows words beginning with hyphens, numbers, and other non-standard characters to be included in the results. The additional work is performed by the first thread. After the first iteration of the while loop, the boolean "extras" is set to false so that these words never appear twice.

In addition to the above function, we also created a non-threaded version of the word frequency calculator called "calculate_word_single." This function uses a single for-loop to store each word in its own location in a HashMap. Calculate_word_single serves as a useful metric to see how well our function outperforms a simpler approach.

Our implementation is similar to that of another multi-threaded word statistics "quanteda" package in R. This package is considered to be faster and more efficient than other Python or R

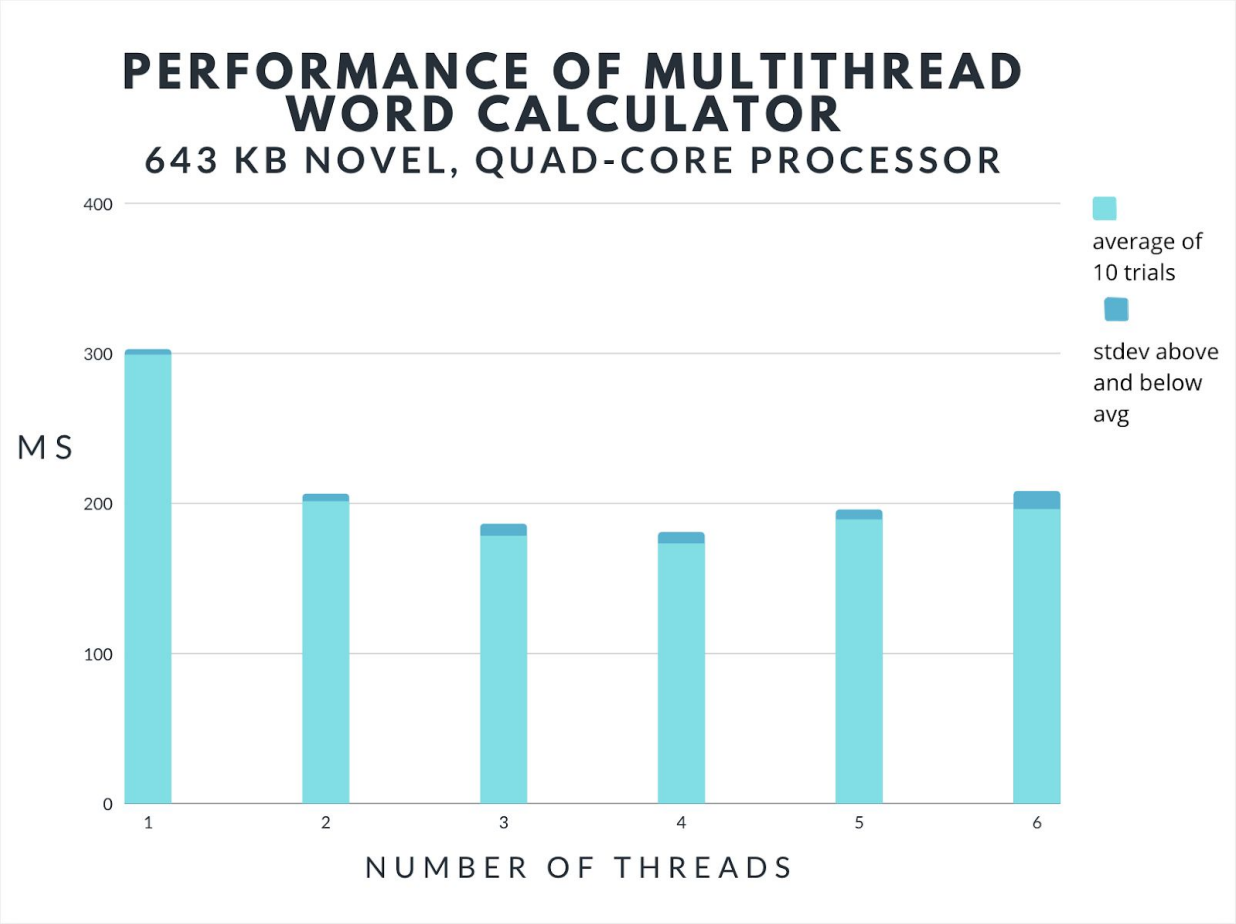
packages because of its extensive multithreading in C++ [1]. The package also tokenizes the strings and saves the words in a hashmap, which greatly increases performance even in a laptop computer [1]. “Quanteda” uses the number of available cores for the number of threads to maximize performance [1]. Quanteda could serve as a way to compare C++ concurrency and Rust concurrency in the future.

Figure 5: Data Access Diagram



Effectiveness and Accuracy:

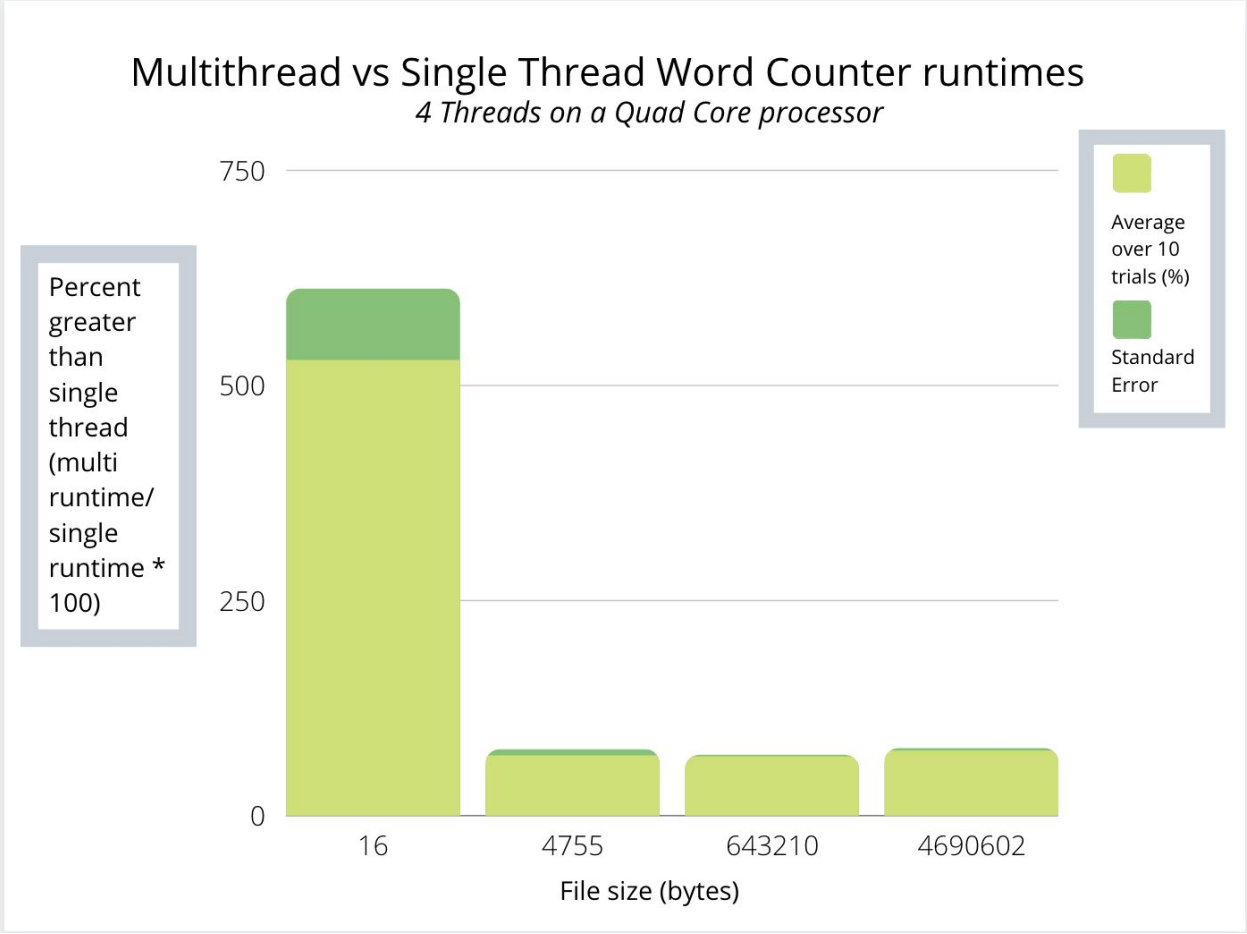
Figure 6: Performance as measured by thread number



First, we wanted to find the optimum thread number for best performance. We ran ten trials for each number of threads on a Macbook Pro with a quad-core Intel processor. The trials all counted words in the same 643 KB text file, which was a human-readable novel. These trials were run on a version of our code without extra features and involve each thread printing the result.

On the quad-core processor, the optimal number of threads was around 4, although there was no statistically significant difference between the performance of 4 and 3 threads (Fig 6). For subsequent tests, we set the number of threads to be four.

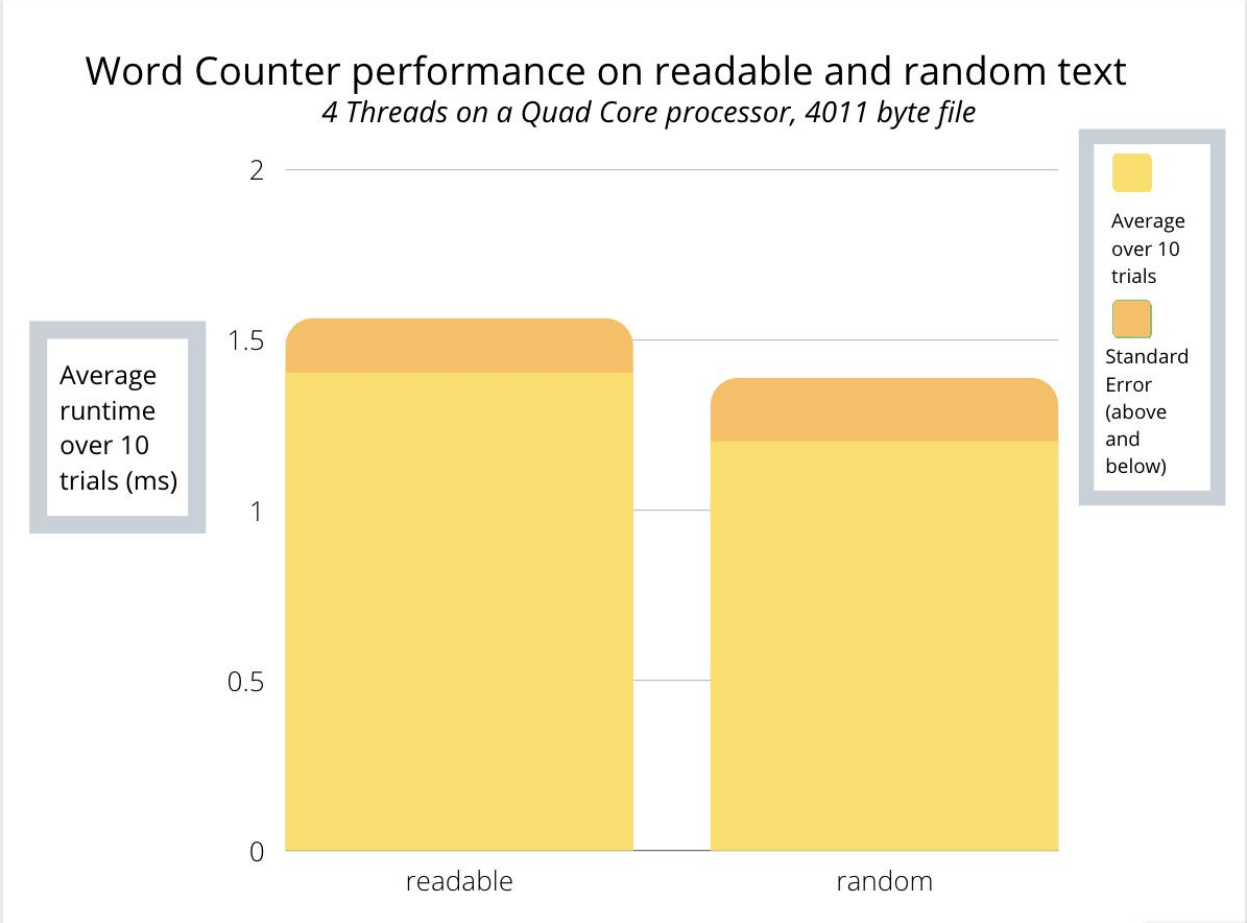
Figure 7: Multi-thread counter outperforms single on file sizes greater than 4755 bytes



After finding the optimal thread number, we compared the performance of our four threaded program to the performance of our single-threaded program on different file sizes. We conducted 4 trials each on four different file sizes and measured the percentage ratio of the multithreaded program to its single-threaded counterpart. All trials used 4 threads and took place on a Macbook Pro with a quad-core Intel processor. The files were all human-readable texts with coherent sentences (a poem, a manual page, the novel Moby Dick, and the Bible combined with Moby Dick).

On very small files at most 16 bytes in size, we conclude that the single-threaded version performs much better (Fig 7) . However, by 4,750 bytes, the multi-threaded version outpaces its counterpart (Fig 7). There does not appear to be a trend past that point. More tests may be performed to determine the exact threshold where these two metrics cross.

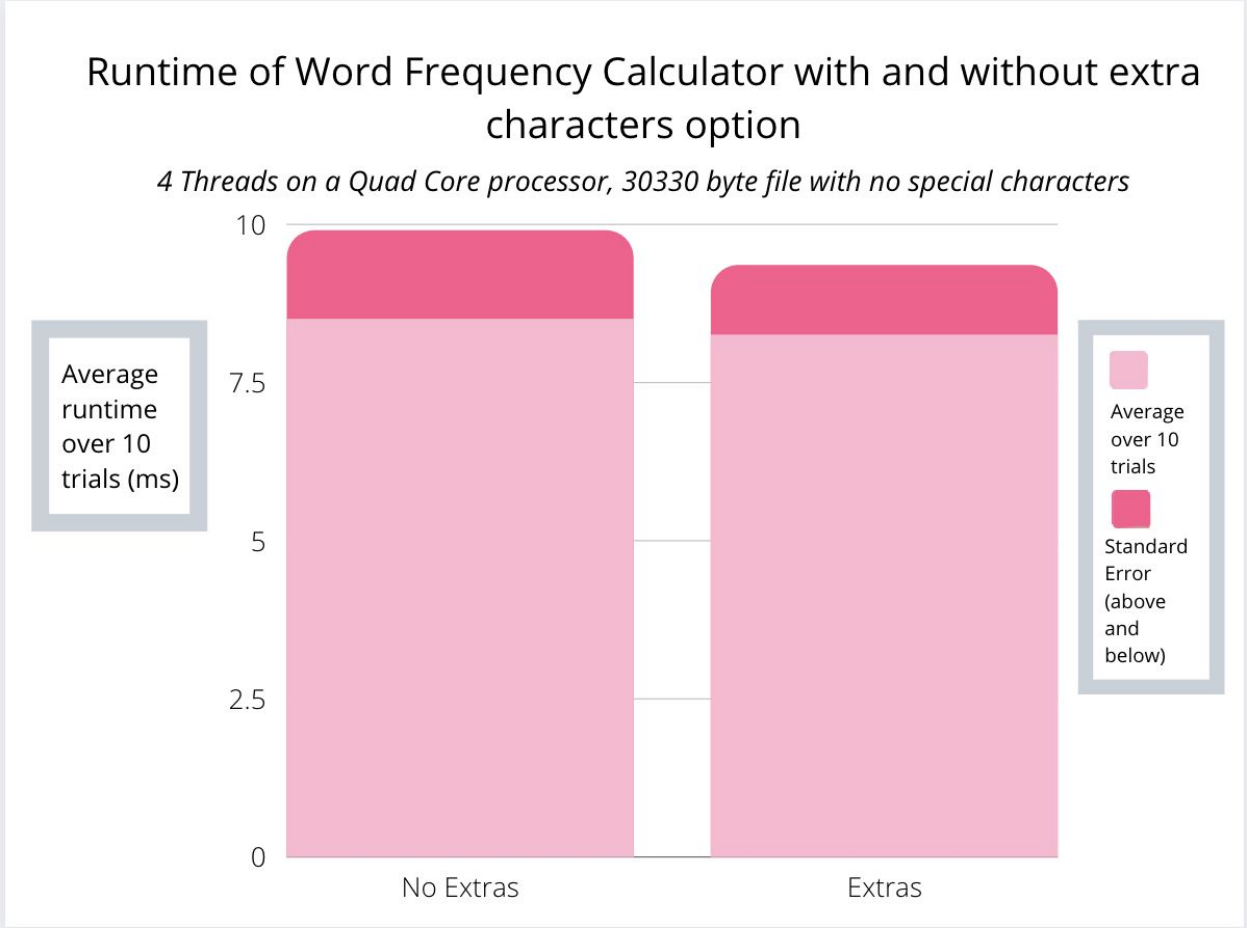
Figure 8: Similar performance on random and readable files



We also tested our word calculator on random and readable text files. We conducted ten trials on two files with equal size (4011 bytes). One file consisted of “lorem ipsum” randomly generated readable words. The other was a portion of *random.txt* from the Canterbury Corpus, a random collection of nonsensical strings.

We conducted a chi-square test on the two data sets and determined that the results are inconclusive ($p = 1$) . The standard deviations are fairly low, however, suggesting the data was fairly accurate (Fig 8). If the difference were true, it may be the case that the readable text performed worse because there were fewer spaces in the random text, meaning fewer words to iterate through and store in the hash table.

Figure 9:No significant performance overhead from extra characters option



As our final test, we compared the runtime of our calculator with and without the extra characters option to measure the overhead of implementing this add-on. We performed ten trials on two identical 30330 byte files consisting of random, 8-character long strings made of only lowercase letters. The version without the extra characters option included no variables or conditionals related to this feature at all.

We can draw no conclusion that the extra option introduces significant runtime overhead. The average runtime for the extra characters option was slightly lower than that without the option (Fig 9). However, both values were within an insignificant range of each other. We predict these results may differ for smaller file sizes, where initialization steps outweigh the cost of the word search itself.

Conclusion:

Rust serves as a capable platform for designing multithreaded programs. Our multi-threaded word counter runs faster than the non-threaded version when applied to medium to large text files. Using ranges of starting letters to divide the workload, we concluded that three to four threads worked best on a quad-core computer. However, having more threads does not equate to greater speed as performance degrades past the optimal number. This may be caused by resource contention as the number of threads exceeds the number of cores, forcing them to share common memory resources. Furthermore, unpublished findings between our quad-core and dual-core computers suggest that the optimal threshold for thread number settles around the number of processors in a device. This discovery further supports that design choices of quanteda. With these findings in mind, if a user wishes to implement this tool, they may also choose to include words that start with characters outside the standard English alphabet. This tool will generate little overhead while providing a more accurate and useful product overall.

References

[1] Benoit, Kenneth, Kohei Watanabe, Haiyan Wang, Paul Nulty, Adam Obeng, Stefan Müller, and Akitaka Matsuo. 2018 quanteda: An R package for the quantitative analysis of textual data. *Journal of Open Source Software*, (October 2018), 3(30), 774. https://doi.org/10.21105/joss.00774.

[2] Guillane Gomez. 2018. Rust Programming By Example. (March 2001). Retrieved December 5, 2020, from https://doc.rust-lang.org/rust-by-example/

[3] Rcpp Parallel. Retrieved December 15, 2020 from https://rcppcore.github.io/RcppParallel/

[4] Sal Khan. 2018. “Chi Squared Statistics for Hypothesis Testing.” Retrieved December 14, 2020, from https://www.khanacademy.org/math/ap-statistics/chi-square-tests/chi-square-goodness-fit/v/chi-square-statistic

[5] Steve Klabnik and Carol Nichols. August, 2018. *The Rust Programming Language.* No Starch Press, Inc. Retrieved December 5, 2020 from https://doc.rust-lang.org/book/title-page.html.

[6] Shepmaster. 2015. “How do I create mutable singleton.” (September 2015). Retrieved December 10, 2020 from https://stackoverflow.com/questions/27791532/how-do-i-create-a-global-mutable-singleton

[7] Stack Overflow. Retrieved December 5, 2020 from https://stackoverflow.com

[8] lazy_static-Rust Retrieved from December 10, 2020 from https://docs.rs/lazy_static/1.4.0/lazy_static/