Cacey Ostic and Nick Petrilli
Parallel Processing
Project 1

Project Overview

Main Project File

The Project1.cpp class is where the main function of our program lives. Above main, we have defined our other functions that are under main and have created a global variable of a HashMap. The main function starts by prompting the user to enter a filename and a number of threads, and using cin to read their answers into the program. The filename is used to open the file given that the correct path has been entered (or in the same directory as Project). After checking if the file can be opened, we then count the number of newline characters for the total number of lines in the file. Using that number of lines, we create an array to hold all of the lines in the file. At this point, we need to calculate how many lines each thread is going to process. Dividing the totalLines by the numThreads will give us the amount of linesPerThread. For cases of uneven integer division, we use the modulus operator to obtain the remainder, and evenly handle the remaining lines by adjusting the endLine of each thread. After calculating each thread's starting and end line (start and end elements of the linesArr), the amount of numThreads are spawned calling the countFrequency function.

The countFrequency function has parameters of the linesArr, start line, end line, and thread number. To start, each thread creates a new hashmap called a threadmap. As described below, our HashMap contains a string key, and an integer value of its count. Then, we loop through the starting and ending lines of the array to find the spaces in the file. When a space is found, we create a substring up to the space, indicating a potential word was found. Now the processString function is called, sending in the potential word to check if it is valid. The processString function checks if each character is a lowercase letter and converts to uppercase, ignoring all special characters besides apostrophes and hyphens. After being processed to confirm the string is a word we want to keep track of, we check if our threadmap already contains that word. If the threadmap contains the word, we increment the count by one, and if the word isn't already in the map then we put it in, with a starting value of 1. When a word is deemed valid and put into the threadmap, the word is erased from the line and the line is reprocessed until it is empty. Since we are finding words based on trailing spaces, the last word in the line won't have a trailing space so we make sure to process the last word by using what is remaining in the line. After all words in the starting to ending lines of that thread are finished, we merge the (smaller) threadmaps into the larger 'words' hashmap. This change to have different hashmaps per thread helped improve our runtime because it allows each thread to handle their own map, and not worry about waiting for locks in the larger map. The merging of each threadmap to the larger map happens at the end of the threads scope, so it has already finished the actions of adding to the map.

After the countFrequency function is executed by each thread, their scope has ended and each of the threads are joined together. At this point all threadmaps have been merged into the main ‘words’ hashmap and need to be sorted. The merge sort function is called, which sorts the hashmap in decreasing order based on the word’s value. After sorting the words and their frequencies are written to the output file. Our first sorting implementation was selection sort, but was changed to merge sort for better time complexity (O(nlogn) compared to O(n^2)).

Throughout the main function we track the runtime of each section (and total) of our code, which is the file reading, word count, and sorting. This was done by using the time_point of the chrono::system_clock, marking the time at the start of the operation and at the end and subtracting the difference. The total runtime is what is used in the tables below. Memory consumption was obtained by using an external python program to monitor the program.

HashMap

The HashMap class is the data structure used to store the word frequencies of the file. It is a custom-made thread-safe map that uses hashing to find values. The main structure used within the class is an array of KeyValue objects. These KeyValue objects, as described in the next section, contain the key-value pairs of strings to integer counts. Each KeyValue object in the array is indexed by the hash of the string key. The array is resized once a specified threshold of the total capacity is reached. This ensures that large amounts of unique words can be stored in the map.

A shared mutex object is used to keep the map thread-safe. Shared mutex has two types of locks: exclusive and shared. The exclusive lock works similarly to a normal mutex. Only one thread may obtain the exclusive lock at a time. This works well for writing operations that edit the map’s array, such as putting in a new value or resizing. The shared lock, on the other hand, can be obtained by many threads at the same time. This works well for read operations like finding or retrieving values. This also works for incrementing specific word counts because thread-safety of the value is handled within the KeyValue class. Additionally, when a thread obtains a write lock, no other thread may obtain a read lock. Similarly, when at least one thread obtains a read lock, no thread can obtain a write lock. This ensures that the array is modified if and only if there are no threads reading the array.

The important member variables and functions are described in more detail below.

Member Variables

- int INITIAL_CAPACITY = 8
    - An integer constant that defines the initial capacity of the array
- int LOAD_FACTOR = 0.75
    - A float constant that determines when the array needs to be resized

- int size
    - The current amount of values in the array
- int capacity
    - The current full size of the array
- KeyValue* array
    - The map of KeyValue pairs
- std::shared_mutex mutex
    - The shared mutex object used for locking critical functions

Private Functions

- void checkResize()
    - This method checks the current size against the load factor. If the size exceeds the load factor, the array size is doubled and all values are rehashed and copied into the new array.
- int hash(std::string)
    - This function implements the hashing of strings. It takes each character in the string and multiplies its ASCII value by an increasing power of a prime number. In this case, the prime number chosen was 31. Then, the modulo operator is used to ensure the hash is brought within the bounds of the map's current capacity.
- int find(std::string)
    - This is one of the most important functions in the class and it serves two purposes. It hashes the string and tries to find it within the map. It handles collisions linearly, meaning if the index at the hash it calculates is already used by another string, it searches forward one place in the map until either the string or an empty space is found. If the string is found, then the index of that place is returned. If an empty space is found, the string is not yet placed in the map. In this case, the index of the empty space is returned as a negative number to communicate that the index is empty rather than found.

Public Functions

- void put(std::string, int)
    - The function attempts to put a new string into the map. If the string already exists, it is incremented instead.
    - This function uses an exclusive (write) lock.
- int get(std::string)
    - The function attempts to find a string in the map. If the string is found, its value is returned. Otherwise, 0 is returned.
    - This function uses a shared (read) lock.
- void increment(std::string)

    - This function attempts to increment the specified value. If the value is not found, the put function is called instead.
    - The function uses a shared (read) lock.
- bool contains(std::string)
    - The function attempts to find a string in the map. If the string is found, it returns true, otherwise false.
    - The function uses a shared (read) lock.
- void addTo(std::string, int)
    - The function attempts to add a specified value to an existing string in the map. If the string is not found, the put function is called instead.
    - The function uses a shared (read) lock.
- KeyValue* getAll()
    - This function traverses the entire map and creates a sequential array of all values. This is used for sorting and printing later in the program.

KeyValue

KeyValue is a small object that acts as a holder for the key-value pairs in HashMap. It contains standard accessors and mutators for both its key and value.

Member Variables

- std::string key
- std::atomic<int> value
    - The value of the pair is atomic to ensure that editing its values is thread-safe. Using atomic over mutex improves runtime.

Public Functions

- Standard accessors and mutators
    - getKey(), getValue(), setKey(std::string), setValue(int)
- void increment()
    - Increments the value by 1. This function exists to make the code more understandable in the HashMap class.

Test Results

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	constitution.txt	26 KB	4503	14.42 MB	0.0067007
2					0.0061212

4					0.0051689
8					0.0055229
12					0.0082456
16					0.0058346
24					0.0110729

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	bible.txt	4,332 KB	853,632	15.11 MB	0.251366
2					0.168128
4					0.146302
8					0.206224
12					0.156501
16					0.11512
24					0.0951076

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	big.txt	12,925 KB	2,231,692	15.14 MB	1.21651
2					1.45206
4					0.581993
8					0.316373
12					0.598773
16					1.27184
24					1.61473

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	verybig.txt	29,210 KB	5,124,675	15.15 MB	3.69576
2					2.3562
4					16.5716
8					13.1414
12					13.8329
16					13.2985
24					8.50777

Num Threads	File Name	File Size	Word Count	Memory	Runtime (s)
1	oneWord.txt	23,555 KB	6,000,000	15.18 MB	0.641207
2					0.459928
4					0.396143
8					0.315945
12					0.40418
16					0.351265
24					0.361441

- Both the constitution.txt and bible.txt have shown improvements in runtime when increasing the number of threads. These files were relatively small compared to the rest, and plateaued at around 8 to 12 threads but not much faster than the single threaded runtime.
- Big.txt is a little more interesting, a larger file that improved runtime up until it peaked at 8 threads. Each run increasing the threads to 12, 16 and 24 increased the runtime rather than decreasing it. This could be due to the increased context switching between the threads, which would include saving/restoring the thread's state and updating memory.
- Verybig.txt is the largest file with lots of different words. This had more interesting results where, starting with 4 threads, the time increased. This may be due to the final merge between all of the threads taking much longer than the processing of the file.

- OneWord.txt seemed to perform the same as big.txt. Increasing the number of threads resulted in decreasing the runtime and peaked at 8 threads. After 8 threads, the runtime increased.