COMP3003 Assignment 1
Report

Bharath Sukesh
19982634
16 Sep, 2021

1 Approach

Starting Threads

My thread creation is contained within my SearchController.java class, which contains the majority of the code associated with Threading. How I go about starting the threads, is that I create a SearchController object in the UI class named FSUserInterface, and I start threads in the constructor of the SearchController class. If SearchController demands access to the GUI Thread, then it can use Platform.runLater() to manipulate the GUI. This means that threading responsibilities are handled by one class (SearchController) in isolation, whilst the GUI thread is only responsible for GUI-related tasks. This setup means that to any class outside SearchController, the threading related logic is abstracted away entirely.

Thread Communication & Resource sharing

The SearchController constructor is where I initialize all my threads. I used a thread pool as per the assignment specification to manage tasks assigned to us. Specifically, I was using a thread pool for the filter() method which deals primarily with iterating through a list (full of strings containing paths for each non-empty file), converting the strings contained within each file into a character array, and then using that as parameters for my method that calculates the similarity between each file. The comparison method depending on the number of files, can be a computationally intensive task, meaning that it would be known as a CPU-bound task. These types of tasks are limited by the power of the CPU. As for the file reading to convert into a character array, this involves *both* I/O and CPU processing. As for writing each comparison to a file, this is obviously an I/O bound task. As for tasks outside of the thread pool, I decided to place the file writing method call / responsibility to be executed in a normal thread, as this is a primarily I/O Bound task, i.e., writing to the results.csv file in the local project directory on disk, for each completed comparison. I believe it'd have been detrimental to place this task inside the thread pool as for example, if you had a folder with 50 files to be cross compared, then there would be 1225 comparisons made, and 1225 writes made to the results.csv file. With this many writes for only 50 files, giving the thread pool responsibility to coordinate that much I/O work would be a waste of resources. As for the thread pool itself, I restricted the number of available threads to be the same as the number of CPU threads available. This program is a mix of I/O and CPU-based tasks, with a bit more emphasis on I/O bound tasks with file reading and writing, meaning it doesn't really make sense to have a huge excess of Threads in the Thread pool than there are the number of CPU Threads available. It's important to note that this isn't to say it *won't* work with any more threads, but I think it won't achieve much of a boost in efficiency doing so.

How do the threads communicate? **How do the threads share resources (if they do at all) without incurring race conditions or deadlocks**?

My main form of resource sharing among threads was by the use of blocking queues. Of the many types of blocking queue available, I felt that the LinkedBlockingQueue was the best fit for my purpose of storing information. Whilst this was required of us to use in this program, I felt this was an efficient way of handling any ongoing communication between two different threads. Initially I had in mind to use a queue for storing the filtered files (before comparisons were made) much like practical 04, however I realised upon some difficulties with making my comparisons asymmetric with the LCS algorithm, i.e., avoiding redundant comparisons to be made (e.g., File A with File B, then File B with File A). This led me to have to change my approach, and so I figured that it would be better off to use a LinkedList there instead. My use of the queue turned out to be more fitting to the problem, as I am using it to store file paths which are *put()* into the queue in the search() method, and taken out in the toFile() method. The benefits of using a queue here was that the search() method that finds all non-empty files aka the Producer in my case, may speed up or slow down, for example depending on the size of the file, how many strings/contents there are within said file. If there was ever a case where the producer sped up so much that we would like to have a buffer of tasks submitted, we can achieve this with a blocking queue, meaning that the consumer method in toFile() doesn't directly impact and

constrain the pace at which the producer works. Whilst blocking queues automatically handle atomicity among threads, any shared resources that don't involve the queue would need to be protected from deadlocks and race conditions. An example of this is my compareTally integer variable, which gets incremented whenever a comparison is completed between one file and all other coexisting files in the selected folder. compareTally is essential for calculating the current progress of comparisons completed, and is a variable accessed many times by many threads. To combat this need for atomicity I used a mutex.

My approach for closing down threads was via the stop button. I built upon the provided demo code, meaning that the existing demo code which ran a method upon clicking the Stop button; I modified this stop button to call a method in SearchController to stop all running threads and tasks. I used Thread.interrupt() as a means for stopping the file searching and writing threads, which throws InterruptedException, which I catch in relevant parts of my code. For this reason, whilst the threads are running, my toFile() method which contains a queue.take() method would block (due to the nature of Blocking Queues) when the queue is empty i.e. no more comparisons are performed/need to be written to file, thereby meaning that it would block before the PrintWriter is in use again. This is another benefit of handing my blocking queue the responsibility of managing file writing, as it means that the PW would not be running endlessly even when it's not doing anything.

As for the thread pools, I used shutdownNow as a means of terminating all running tasks within the pool immediately, instead of shutdown(); the main difference being that shutdownNow() will attempt to terminate previously submitted and waiting tasks by interrupting their relevant threads before the thread pool itself terminates.

2 Scalability

2.1

NFR's:

i) Execution pace: The program must be fast such that it should finish comparisons between 50 files under 5KB, at a maximum of 2 seconds.

ii) File Quantity Capability: The program must be capable of handling comparisons between at least 500 files under 5KB, at a maximum of 5 seconds.

iii) File Size Capability: The program must be capable of reading in and operating with 500 files between 50 and 100KB in size, without memory failure.

The first two Non-functional requirements ensure that the program works at a steady pace with for example 50 input files of small size (<5KB), and also that it utilizes multithreading by not slowing down *significantly* when a larger number of input files (500) are provided. The third non-functional requirement suggests for the program to be capable of handling *a large amount of data*, and while "large" is subjective, I have stated 500 files between 50 and 100KB in size, so as to ensure that the program does not struggle once scaled to bigger sets of input data.

Problems:

A large amount of input data would cause some problems that would need to be dealt with, with regards to scalability. With the way the program is currently laid out, I read in all files in a specified directory into a Linked List of Strings, containing the file path/directory to each and every non-empty text file. The comparisons only get performed once the list gets populated, meaning that if you provide a folder that has a very large number of files to read in, then this may take a long time. A similar issue can occur if the user provides files that contain

a very large number of contents, i.e., if the file is a text file containing ~ 500 lines. This would hamper the performance, as the current method of operation is that each file's contents is read into a Character array, to which then each character array gets compared with one another. In both cases mentioned above, the I/O work of reading in files can cause a potential bottleneck in the program.

Speaking of bottlenecks, this brings me to my next problem in that the above issues can cause memory errors, if the files are too big, or if there are too few Threads (or Memory allocated) for the computer to cope with the demands of the task. This may end up causing an OutOfMemoryError exception to be thrown at runtime. Secondly, another potential issue that may arise as a result of dealing with a very large amount of data is the GUI Thread freezing because of a large load mounted onto the CPU for large computationally intensive tasks e.g., comparing two files with a large number of lines, or freeze because of a lack of memory.

Lastly, a problem that would need to be sorted would be the inefficiencies of the provided LCS Algorithm. We make a string object for every line of text in each file, which is then converted to a Character array, meaning the String object is momentarily forgotten about/discarded. These strings accumulate in Java's so called "garbage collector", where it finds unused objects like the above-mentioned strings, and attempts to free them from memory i.e. clean up and *collect the garbage*. This is a periodic process that takes time, as it needs to be able to find said objects, and in our program's case, when we have so many strings (for each line of each file), this can produce a slowdown in performance.

2.2

There are a few architectural approaches that could be applied to deal with the aforementioned problems such as making the application utilise Frameworks to gain a better program performance with a larger input data set. The program already uses the JavaFX framework for the GUI, however utilising other frameworks for I/O operations such as reading and writing files would result in a better program performance with a larger data set. Secondly, carefully choosing frameworks and efficient API's will allow for the program to survive the issues revolving around scalability, as the newer Framework would adapt better to larger volume of input data. The aim of this would be to avoid the problems raised previously with memory issues, crashes and freezes, as choosing an appropriate Framework would allow for a better program architecture and therein a program that adapts to higher volume of input data. A defined architecture would resolve issues like "reading in all files *then only comparing* amongst them", as the design for the program would be more defined and pose less a risk for architectural flaws. As for the flaws in our algorithmic choice for file comparison, I would investigate and choose an algorithm that would perform far better in a worst-case time complexity. Also, I would make sure to alter the algorithm so that it can avoid the mentioned "garbage collection" problem as much as possible. An optimal algorithm would positively impact the performance as the volume of input data increases, which I believe would make the program more scalable.

Lastly, specifically for the scenario provided to us where multiple users would run the program at different points in time; due to the nature of the program, I assume this involves multiple users connecting to a centralized server so that they can access the same file system, and thereby use the program all in a similar manner. With a system like this at hand I believe the Client-Server architecture would be the most ideal setup. This would mean a centralized server would host the file system and any user from anywhere, can access these files and perform comparisons. These types of networks are also scalable in that increasing the size of the main server would improve the network's sustainability without much more hassle. In saying that, there are some potential performance issues that arise with a Client-Server model. If you have a very large number of clients, then for example if hardware at the centralized server is outdated or performs poorly, then the network can get congested, and when under heavy load, its performance can deteriorate or worse, result in crashes. There is also the architectural issue with this that if the centralized server were to fail, any client requests to the Server would fail and thus would render the network ineffective.