Mutli-Threaded Word Counter

(1.1) General Perspective – Resource Sharing:

Multi-threading in general has a better performance than a single thread as they share the resources like data structures, memory, etc.

When we consider our word frequency counter application, we can find the same as Multi thread counter has better performance in a way as they share the thread safe data structures such as **ArrayBlockingQueue** to store the hyperlinks as we are progressing and **Concurrent HashMap** to store the count of words in the hyperlink and the corresponding hyperlink.

Program Perspective – Processing of Hyperlinks:

Description of Thread-safe data structures and resource sharing in program:

No	Thread -safe Data Structures	Name	Purpose
1.	ArrayBlockingQueue	**queue**	To store the urls as and when processing and remove them when they are getting visited.
2.	ArrayBlockingQueue	**queuecopy**	To store the urls in the queue whenever new hyperlinks are there and here it is not removed anytime. It is taken as a Buffer Queue for **comparison to add unique and unvisited urls anytime**.
3.	Vector	**visited**	To store the list of urls visited. Removed from queue and added to visited on visiting the url.
4.	Concurrent HashMap	**results**	To store and display urls and word occurrences.

In case of a single threaded application,

1. We can only take one hyperlink at a time and control the process of that hyperlink from Queue to Visited vector.
2. In Visited vector, we compare if we haven't visited the url previously, we add it to the visited vector
3. Then, the same thread finds the count of the word in visited url added and finally displays the result count in each url.
4. On hitting any of the conditions, the printStatistics method gets notified and displays the result.

Multi-Thread Counter:

Whereas in a multi-thread application that we designed for the word counter, creates multiple thread at a time with the Execute method in the Executor Service.

Then the first two above steps can be done by any thread running in parallel i.e., multiple threads can at a time check the Queue removes it from main queue, compare it with the visited and if and only if the visited is not having the url, we can add it to visited. **This can be done my many threads in parallel reducing time and increasing performance.**

Following which we must restrict the process of getting the content from the corresponding url and adding the count to be done by single thread as multiple threads in case performing this operation unnecessarily is a waste of processing time and memory. Hence, we restrict this to single thread and finally output the result by any thread encountering the condition.

(1.2) Crawling Strategy Used:

I have used BFS approach to efficiently make use of the ArrayBlockingQueue in multithreading. The BlockingQueue has the capability of controlling the threads trying to take or insert the elements into the queue. So, when a thread tries to put elements in excess to the capacity of the queue, it will wait for the queue to become available and when a thread tries to take an element from the queue when it is empty, it gets blocked until there is an element to take.

I have used **put** and **take** methods in **ArrayBlockingQueue** to insert and remove elements accordingly.

Program Implementation Brief according to BFS:

Step 1: Enqueue seedURL to Q

Initially the first url is the seedurl which is given as input to the queue in the main and the same url is being added to queuecopy which I used as buffer to compare the urls anytime before adding.

Step 2: Start a new WebAnalyser thread to process URL

I have called the find method from main to process the url.

Step 3: Dequeue a URL from Q

Before processing the current url, I have removed the url from the queue so that one stage queue will become empty satisfying the while condition. I have removed from the queue but not queuecopy to keep track of urls getting added to queue everytime so that we will not add again and again to avoid queue becoming too full.

Step 4: Add url to Visited

I have processed the url and added the url to visited before I get the content and hyperlinks. Visited keeps track of processed urls. Also, Visited is a vector to ensure thread-safety and I made a check to ensure only unique url goes in everytime since multiple threads may process the urls have a chance to add it in.

Step 5: If url contains keyword then add it to results

I have added the url to results when the keyword exists in the url by ensuring a condition if **count** > 0, then put to the results Map.

Step 6: For each hyperlink on the url, enqueue to Q

I added the urls using advanced for loop and processed them until the queue is empty.

(1.3) According to my implementation, all the pages will be traversed at a time since I have a loop controlling the thread count which is of limit assigned to 100.