**Advanced Enterprise Java**

**Project-3**

Submission by Swapnil Patel (1966690)

https://github.com/swap1210/ Adv-java-assignment3/

Table of Contents

[How to Execute: 3](#_Toc130335244)

[Driver Class: 3](#_Toc130335245)

[Q1. QuickSort class 4](#_Toc130335246)

[Q2. ParallelQuickSort class 5](#_Toc130335247)

[Q3.ParallelQuickSortThreadPool class 6](#_Toc130335248)

[Q4. Analysis 7](#_Toc130335249)

# How to Execute:

To execute just run the Driver.java file inside src folder.

> javac asg3/Driver.java && java asg3/Driver

Expected output:

A picture containing text, newspaper, plaque

Description automatically generated

The outcome can be pasted in excel column 1 and the provided excel will automatically separate $ delimited values and plot a chart against it.

It shows for various array sizes how long each sort algorithm ran.

# Driver Class:

It performs benchmark tests on three different implementations of the Quick Sort algorithm. Here's an explanation of what the code does:

* This code is an implementation of a test driver that performs a benchmark test on three different types of quicksort algorithms: Quick Sort, Parallel Quick Sort, and Thread Pooled Parallel Quick Sort. The code generates arrays of different types (integer, float, and character) of unique and random data to be sorted, with each array of different sizes (100, 500, 1000, 2500, and 5000). For each array, the three sorting algorithms are executed, and the execution time is measured in milliseconds. The results of each test are stored in a list of lists called testResults, and then printed out to the console.
* The main() method is the entry point for the program. It first initializes the testResults list and creates a header row that describes the columns in the test result table. It then proceeds to test the three sorting algorithms on arrays of different types and sizes. Each test is performed by calling the performThreeTest() method and passing in an array of test data. The results of each test are stored in a List<String> called result and added to testResults list.
* The performThreeTest() method is responsible for executing the three sorting algorithms and measuring the execution time. It takes an array of type K, which is a generic type that extends the Comparable interface. The method creates instances of the QuickSort, ParallelQuickSortThreadPool, and ParallelQuickSort classes, which are the implementations of the three sorting algorithms. It then adds the test data to each of the sorting algorithm's input lists. Finally, the method executes each of the three sorting algorithms and measures the execution time using the System.nanoTime() method. The execution time of each algorithm is added to the result list, and the list is added to the testResults list.
* The fillUniqueIntArray(), fillUniqueFloatArray(), and fillCharArray() methods are helper methods that generate arrays of unique random data of a specified size for integer, float, and character data types, respectively.

# Q1. QuickSort class

This is an implementation of the QuickSort algorithm in Java, using generics to sort any type of Comparable objects. Here is a brief overview of the code:

* The QuickSort class is defined with a type parameter T that extends the Comparable interface, indicating that the objects to be sorted must implement the compareTo method.
* The class has a field arr that is an ArrayList of objects of type T, representing the array to be sorted.
* There is a constructor that creates a new empty ArrayList.
* The swap method takes two indices i and j and swaps the elements at those positions in the array.
* The partition method takes two indices low and high that represent the range of the subarray to be partitioned. It selects the last element of the subarray as the pivot, and then moves all elements smaller than the pivot to the left of it and all elements larger than the pivot to the right of it. It returns the index of the pivot after the partitioning is done.
* The perform method is the public method that performs the sorting. It simply calls the private perform method with the range of the whole array.
* The private perform method takes two indices low and high that represent the range of the subarray to be sorted. It first checks if the subarray has more than one element, and if so, it partitions the subarray around a pivot element and then recursively sorts the two resulting subarrays on either side of the pivot.

To do temp test:

> javac asg3/Q1Driver.java && java asg3/Q1Driver

Text

Description automatically generated

# Q2. ParallelQuickSort class

This code implements a parallel version of quicksort algorithm in Java, which utilizes the Fork-Join framework to take advantage of multi-core processors. The algorithm partitions the array around a randomly selected pivot element, and recursively sorts the sub-arrays using parallel threads.

* The class ParallelQuickSort extends RecursiveTask<Integer> and implements the MySorters interface. It contains an arr field of type List<T> which represents the array to be sorted, and start and end fields which define the range of the current sub-array to be sorted.
* The perform() method initializes the ForkJoinPool and starts the sorting process by invoking the compute() method of an instance of ParallelQuickSort on the range [0, n-1], where n is the size of the input array.
* The compute() method is the heart of the parallel sorting algorithm, which is implemented recursively as a fork-join operation. The method first checks if the current range of the array is less than or equal to 1, and if so, returns null (base case). Otherwise, it selects a random pivot element, partitions the array around it, and recursively sorts the two sub-arrays using two separate threads. The left sub-problem is forked and the right sub-problem is computed by the current thread. Then the current thread waits until the left thread completes using the join() method. Finally, the method returns null to indicate that no value is to be returned.
* The partition() method is used to partition the array around the randomly selected pivot element. The method selects a random index between start and end and swaps that element with the last element of the array. Then, it iterates over the sub-array from start to end-1, swapping elements as necessary so that elements smaller than the pivot are to the left of it, and elements greater than or equal to it are to the right. Finally, it swaps the pivot with the first element of the right sub-array and returns the index of the pivot.

To do temp test:

> javac asg3/Q2Driver.java && java asg3/Q2Driver

Text

Description automatically generated

# Q3.ParallelQuickSortThreadPool class

This is a Java implementation of parallel quicksort using a thread pool. The algorithm splits the input list into sub-lists that are sorted by separate threads.

The ParallelQuickSortThreadPool class implements the MySorters interface, which defines a perform() method that sorts a list. The input list is initialized as a synchronized list to avoid thread synchronization issues. The N\_THREADS constant is set to the number of available processors, and the FALLBACK constant is used to determine when to fall back to sequential quicksort. The thread pool is initialized with a fixed number of threads equal to N\_THREADS.

The QuicksortRunnable class is an inner class that implements the Runnable interface, which defines a run() method that performs the sorting. The values list is the list to be sorted, and the left and right indices specify the range of the list to be sorted. The count parameter is an AtomicInteger that keeps track of the number of threads currently executing.

The quicksort() method sorts the sub-list using the quicksort algorithm. If the number of threads currently executing is greater than or equal to FALLBACK \* N\_THREADS, the sub-list is sorted using sequential quicksort. Otherwise, the sub-list is divided into two sub-lists that are sorted by separate threads. If the number of threads is not enough, the count is incremented and new QuicksortRunnable objects are created and executed by the thread pool.

The partition() method partitions the sub-list into two parts: elements less than the pivot and elements greater than or equal to the pivot. The swap() method is used to swap elements.

Finally, the perform() method executes the QuicksortRunnable object on the thread pool and waits until all threads finish.

To do temp test:

> javac asg3/Q3Driver.java && java asg3/Q3Driver

Text

Description automatically generated

# Q4. Analysis

How to interpret the output:

There are benchmarks for each array be it integer, float, or character.

Pasting this code in excel and delimiting by $ we’ll get a table like this:

Text

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Quick Sort | Parallel Quick Sort | Thread Pooled Parallel Quick Sort |
| 100 | 2.18875 | 17.27075 | 5.512459 |
| 500 | 0.8045 | 30.964083 | 28.282291 |
| 1000 | 4.490667 | 15.286917 | 10.225042 |
| 2500 | 1.393209 | 11.154417 | 12.982333 |
| 5000 | 3.172708 | 11.83075 | 10.272708 |

Chart, line chart

Description automatically generated

Similarly for float values:

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Quick Sort | Parallel Quick Sort | Thread Pooled Parallel Quick Sort |
| 100 | 2.920875 | 1.333917 | 5.19325 |
| 500 | 0.458042 | 2.041166 | 15.193 |
| 1000 | 0.935958 | 10.120083 | 3.045083 |
| 2500 | 4.250084 | 8.307417 | 8.819833 |
| 5000 | 4.561958 | 12.64525 | 14.350541 |

Chart, line chart

Description automatically generated

Similarly for character values:

Text

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Quick Sort | Parallel Quick Sort | Thread Pooled Parallel Quick Sort |
| 100 | 1.422084 | 6.323666 | 4.508125 |
| 500 | 1.277 | 3.938791 | 7.337167 |
| 1000 | 1.403625 | 7.084875 | 9.998042 |
| 2500 | 3.629625 | 25.902708 | 14.203959 |
| 5000 | 2.819833 | 53.424792 | 28.838167 |

Chart, line chart

Description automatically generated