CS3354 – Fall 2017 – Assignment 6

Due date: Monday, Dec. 4, 2017 at 11:55 pm.

Goal:

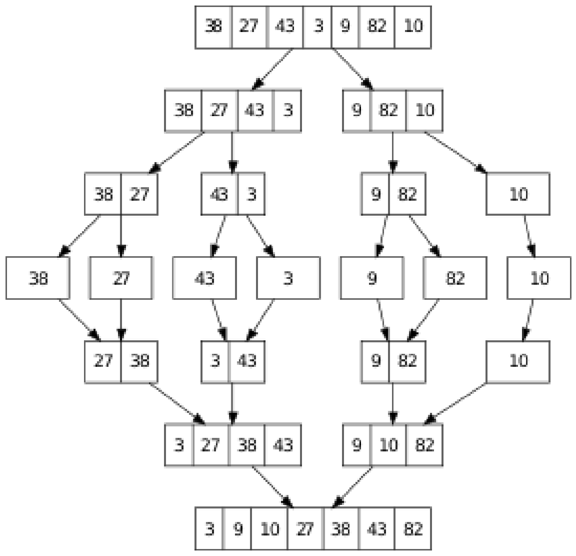
The goal of this assignment is to help students understand the concepts of Java Concurrency and Multi-threaded programming.

Description:

The attached source code shows an example of a program that can sort a set of numbers using the top-down MergeSort[1] algorithm. In the given implementation, the program uses a single thread to sort the whole input. MergeSort is a sorting algorithm that can be easily parallelized due its divide-and-conquer nature[2]. On CPUs with multiple cores, multiple threads can speed up the sorting time, by splitting the work.

MergeSort is a divide-and-conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

The figure below shows how an array of size 7 is split into two halves and it continues like that until each subarray is of size one, at which point it starts merging.



1 https://en.wikipedia.org/wiki/Merge_sort

2 https://en.wikipedia.org/wiki/Merge_sort#Parallel_merge_sort

The following is a pseudocode of the MergeSort algorithm:

```
mergeSort(arr[], l,  r)
if r > l
   Find the middle point to divide the array into two halves:
      middle m = (l+r)/2
   Call mergeSort for first half:
      Call mergeSort(arr, l, m)
   Call mergeSort for second half:
      Call mergeSort(arr, m+1, r)
   Merge the two halves sorted in step 2 and 3:
      Call merge(arr, l, m, r)
```

The pseudocode above can be modified to allow mergeSort(arr, l, m) and mergeSort(arr, m+1, r) to run in parallel.

```
parallelMergeSort(arr[], l,  r)
if r > l
   if (avalableTreads <= 1)
      mergeSort(arr[], l,  r) // Call original non-paralel mergeSort
   else
      Find the middle point to divide the array into two halves:
         middle m = (l+r)/2

      Call parallelMergeSort for first half in a new thread:
         Call parallelMergeSort(arr, l, m) in new thread

      Call parallelMergeSort for second half in a new thread:
         Call parallelMergeSort(arr, m+1, r) in new thread

      Wait for threads to finish

      Merge the two halves sorted in previous steps:
         Call merge(arr, l, m, r)
```

Note that the pseudocode above uses the variable avalableTreads to decide whether to allocate a new thread or not based on the number of threads that has been made available to the program. That means that every time a new thread is created, the variable avalableTreads should be updated to reflect the number of threads that are still available.

Tasks:

1. Modify the MergeSorter class to convert it into a parallelized MergeSort that uses multiple threads to perform the sorting task. The maximum number of threads to be used should be passed as an argument to the sorting method. Your program should not allow more than the specified maximum number of threads to run in parallel. Call the modified class ParallelMergeSorter.

2. Modify the SortTester class provided to run some sorting simulations using the parallelized ParallelMergeSorter, to assess the possible speedup from using multiple threads, as opposed to a single thread. Call the modified class ParallelSortTester. Each experiment should time the sorting speed with different sizes of random number arrays, starting from an array of size 1000 and doubling the size of the array at each round, for 15 rounds (last round array size should be 16384000). Run the experiment with different number of threads starting from one thread and doubling the number of threads, up to the number of CPU cores available in your system. The number of cores available in your computer can be obtained from Java using the following statement Runtime.getRuntime().availableProcessors();.
   <u>Copy the output results and paste them into text file</u>. **Submit your output together with your code.** At the end of this document you can see the output of an example run on my computer.

Notes:

1. For task 2 you will need to generate arrays with randomly distributed numbers (integers or doubles).

2. To make sure that your parallelized MergeSort works correctly, use the method isSorted, of the given class SortTester which takes an array as an input argument and returns true or false, depending on if the input array is sorted or not.

Logistics:

This assignment can be done and submitted individually by each student.
Submit your answer in a single file (assign6_xxxxxx.zip). The xxxxxx is your TX State NetID. Submit an electronic copy only, using the Assignments tool on the TRACS website for this class. **Do NOT include executable or .class files in your submission.**

Sample output of ParallelSortTester:

```
1 threads:
    1000 elements  =>     5 ms
    2000 elements  =>     6 ms
    4000 elements  =>    13 ms
    8000 elements  =>    26 ms
   16000 elements  =>     5 ms
   32000 elements  =>    10 ms
   64000 elements  =>    14 ms
  128000 elements  =>    23 ms
  256000 elements  =>    50 ms
  512000 elements  =>   107 ms
 1024000 elements  =>   245 ms
 2048000 elements  =>   541 ms
```

```
   4096000 elements  =>    1243 ms

2 threads:
      1000 elements  =>      1 ms
      2000 elements  =>      1 ms
      4000 elements  =>      1 ms
      8000 elements  =>      2 ms
     16000 elements  =>      4 ms
     32000 elements  =>      6 ms
     64000 elements  =>      7 ms
    128000 elements  =>     16 ms
    256000 elements  =>     41 ms
    512000 elements  =>    105 ms
   1024000 elements  =>    155 ms
   2048000 elements  =>    360 ms
   4096000 elements  =>    803 ms

4 threads:
      1000 elements  =>      1 ms
      2000 elements  =>      1 ms
      4000 elements  =>      1 ms
      8000 elements  =>      1 ms
     16000 elements  =>      3 ms
     32000 elements  =>      6 ms
     64000 elements  =>     66 ms
    128000 elements  =>     12 ms
    256000 elements  =>     33 ms
    512000 elements  =>     54 ms
   1024000 elements  =>    131 ms
   2048000 elements  =>    270 ms
   4096000 elements  =>    690 ms

8 threads:
      1000 elements  =>      1 ms
      2000 elements  =>      1 ms
      4000 elements  =>     64 ms
      8000 elements  =>      1 ms
     16000 elements  =>      2 ms
     32000 elements  =>      3 ms
     64000 elements  =>      7 ms
    128000 elements  =>     18 ms
    256000 elements  =>     22 ms
    512000 elements  =>     43 ms
   1024000 elements  =>    104 ms
   2048000 elements  =>    250 ms
   4096000 elements  =>    537 ms
```