Assignment 5

TASK

Please see the presentation on *Assignment on Parallel Sorting* under the *Exams. etc.* module.
Your task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You will consider two different schemes for deciding whether to sort in parallel.

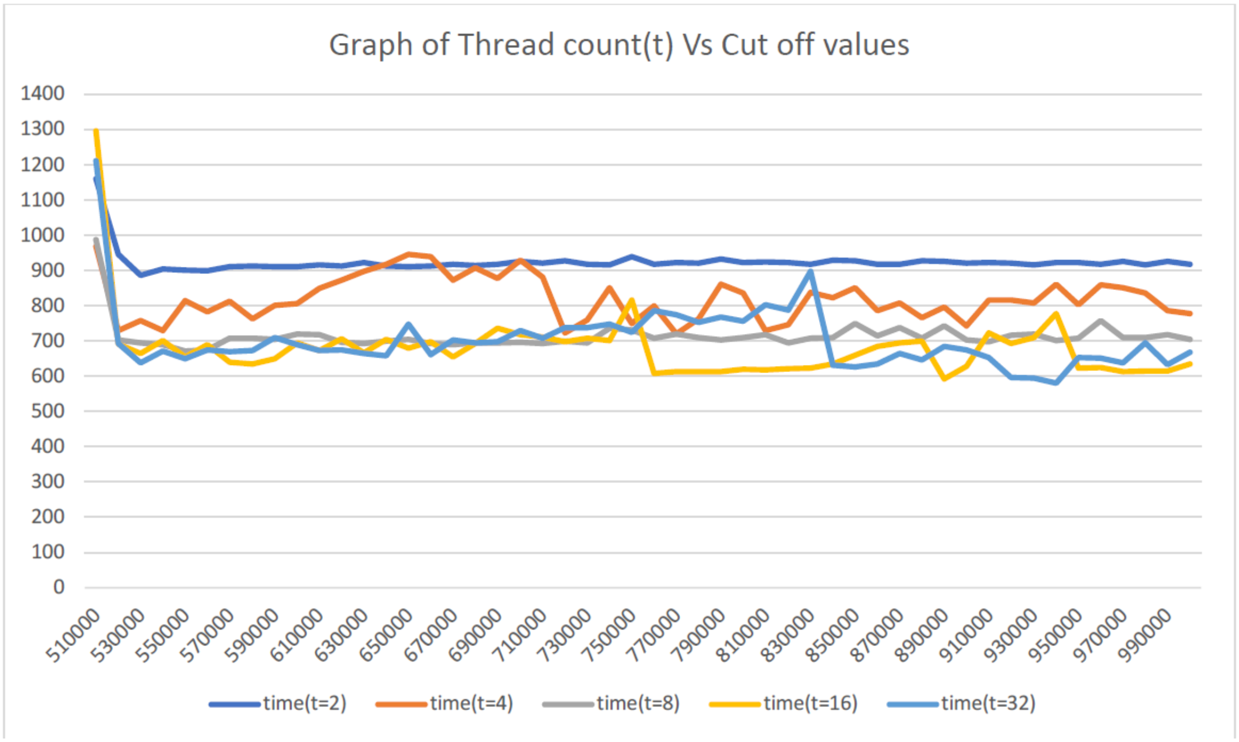
1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. Recursion depth or the number of available threads. Using this determination, you might decide on an ideal number (*t*) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of *lg t* is reached).
3. An appropriate combination of these.

There is a *Main* class and the *ParSort* class in the *sort.par* package of the INFO6205 repository.
The *Main* class can be used as is but the *ParSort* class needs to be implemented where you see "TODO..." [it turns out that these TODOs are already implemented].
Unless you have a good reason not to, you should just go along with the Java8-style future implementations provided for you in the class repository.
You must prepare a report that shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cutoff schemes.

OUTPUT:

The first change I made to ParSort.java is to introduce thread counts using ForkJoinPool.
This made the number of threads used more manageable in the code.
The next thing I did, was introduced numerous variations of thread-count and input size values.
I also added a GetMin.py which would print out the minimum values from all CSV files to give a holistic view of the best cut off value for the given algorithm.

RESULT:
Graphical representation of evidence:



Analyzing the output & evidence:

Input Data Range: 2000000 to 5000000

Cutoff Range tried: starting from 510000 to with an interval of 100000

Considered Thread Count Values - 2, 4, 8,16 and 32

From the above experiment its clear that cutoff value: 630000 with thread count of 8 performs well as N – number of input element increases. The algorithms perform best with these set of values.

Observations for rejecting other thread count values:

1. When thread count value = 32, the graph is flat. That means even with less number of input elements the sorting algorithm is going to take the same amount of time as it would do with a higher value of the input elements.
2. When thread count value = 16, the graph is not that flat. The execution time peaks and falls at random places. It performs poorly with less values of cutoff.
3. When thread count value = 4, The execution time peaks and falls at random places. It performs poorly with most values of cutoff.