# Program Structures and Algorithms
Fall 2022(SEC 06)

NAME: Olasunkanmi Olayinka
NUID: 001512266

## 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.

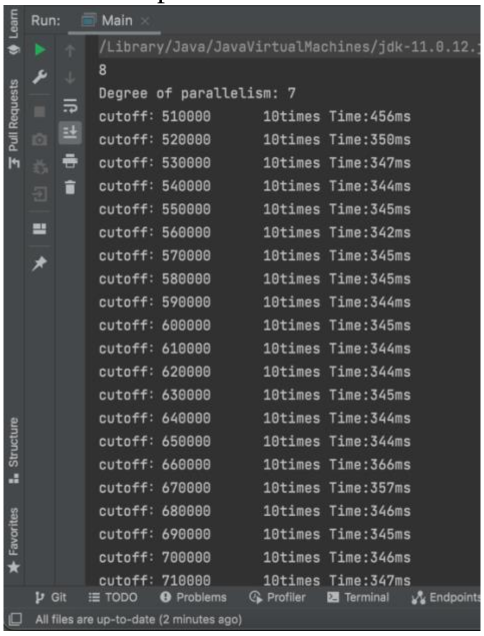
1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. 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.

## Relationship Conclusion:

- With the increase of available threads, the performance increased, hence the time (t) taken to sort decreases with increasing thread (R)
- Combining parallel sorting and cutoff should have a better performance for merge sort
- Given the number of threads (N) available for a CPU, while sorting in parallel N number of threads will work fastest

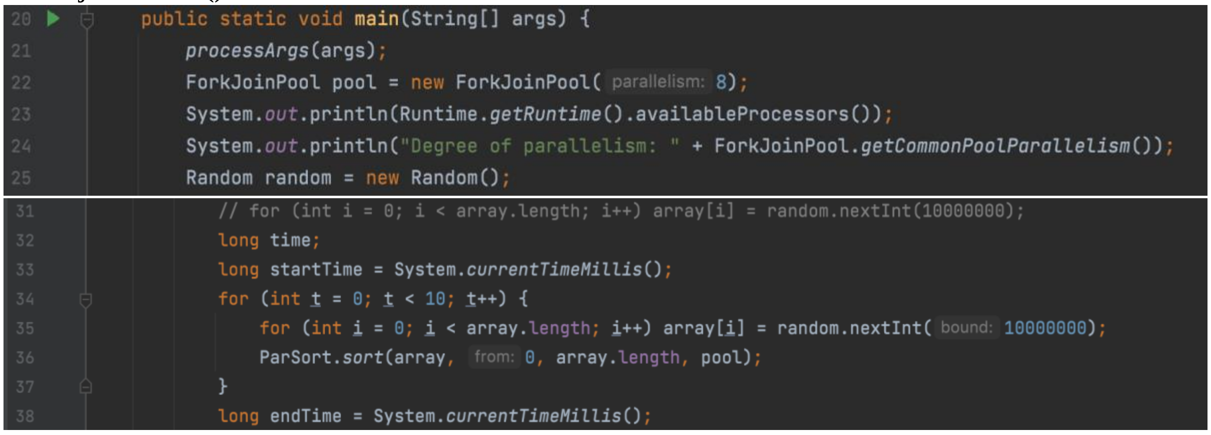
## Evidence to support that conclusion:

- Console output



```
/Library/Java/JavaVirtualMachines/jdk-11.0.12.
8
Degree of parallelism: 7
cutoff: 510000      10times Time:456ms
cutoff: 520000      10times Time:350ms
cutoff: 530000      10times Time:347ms
cutoff: 540000      10times Time:344ms
cutoff: 550000      10times Time:345ms
cutoff: 560000      10times Time:342ms
cutoff: 570000      10times Time:345ms
cutoff: 580000      10times Time:345ms
cutoff: 590000      10times Time:344ms
cutoff: 600000      10times Time:345ms
cutoff: 610000      10times Time:344ms
cutoff: 620000      10times Time:344ms
cutoff: 630000      10times Time:345ms
cutoff: 640000      10times Time:344ms
cutoff: 650000      10times Time:344ms
cutoff: 660000      10times Time:366ms
cutoff: 670000      10times Time:357ms
cutoff: 680000      10times Time:346ms
cutoff: 690000      10times Time:345ms
cutoff: 700000      10times Time:346ms
cutoff: 710000      10times Time:347ms
```

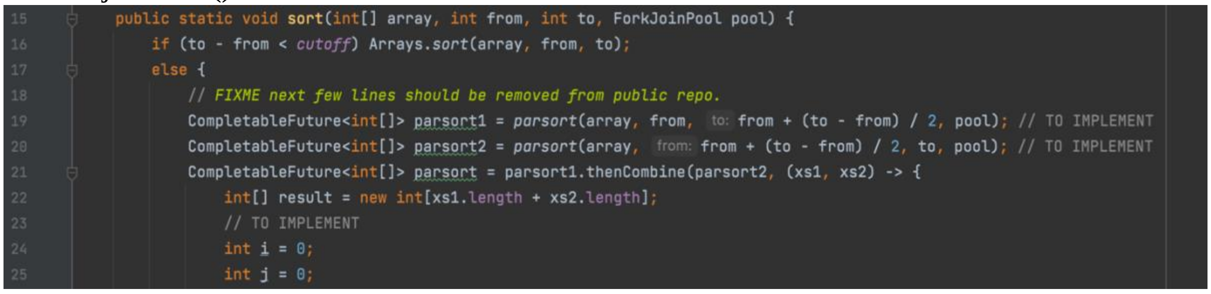
- Main.java main() method



```java
    public static void main(String[] args) {
        processArgs(args);
        ForkJoinPool pool = new ForkJoinPool( parallelism: 8);
        System.out.println(Runtime.getRuntime().availableProcessors());
        System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
        Random random = new Random();
```

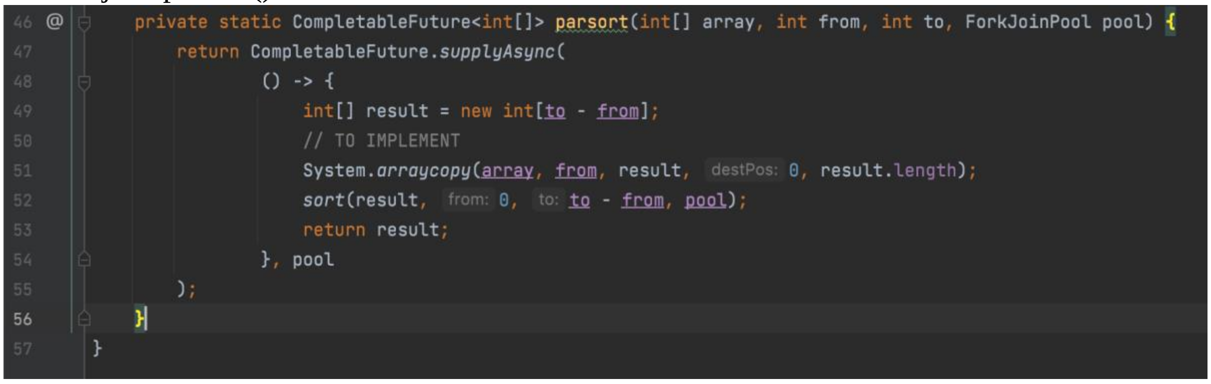
```java
        // for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);
        long time;
        long startTime = System.currentTimeMillis();
        for (int t = 0; t < 10; t++) {
            for (int i = 0; i < array.length; i++) array[i] = random.nextInt( bound: 10000000);
            ParSort.sort(array, from: 0, array.length, pool);
        }
        long endTime = System.currentTimeMillis();
```

- Parsort.java sort() method



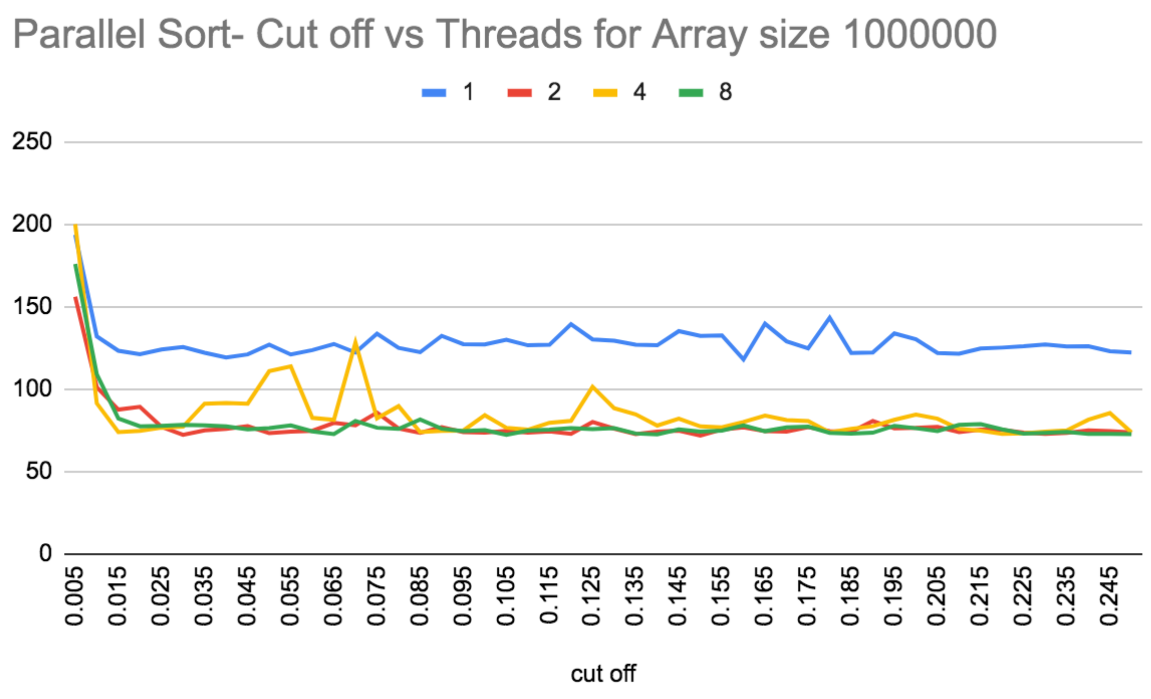
```java
    public static void sort(int[] array, int from, int to, ForkJoinPool pool) {
        if (to - from < cutoff) Arrays.sort(array, from, to);
        else {
            // FIXME next few lines should be removed from public repo.
            CompletableFuture<int[]> parsort1 = parsort(array, from, to: from + (to - from) / 2, pool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, from: from + (to - from) / 2, to, pool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {
                int[] result = new int[xs1.length + xs2.length];
                // TO IMPLEMENT
                int i = 0;
                int j = 0;
```

- Parsort.java parsort() method

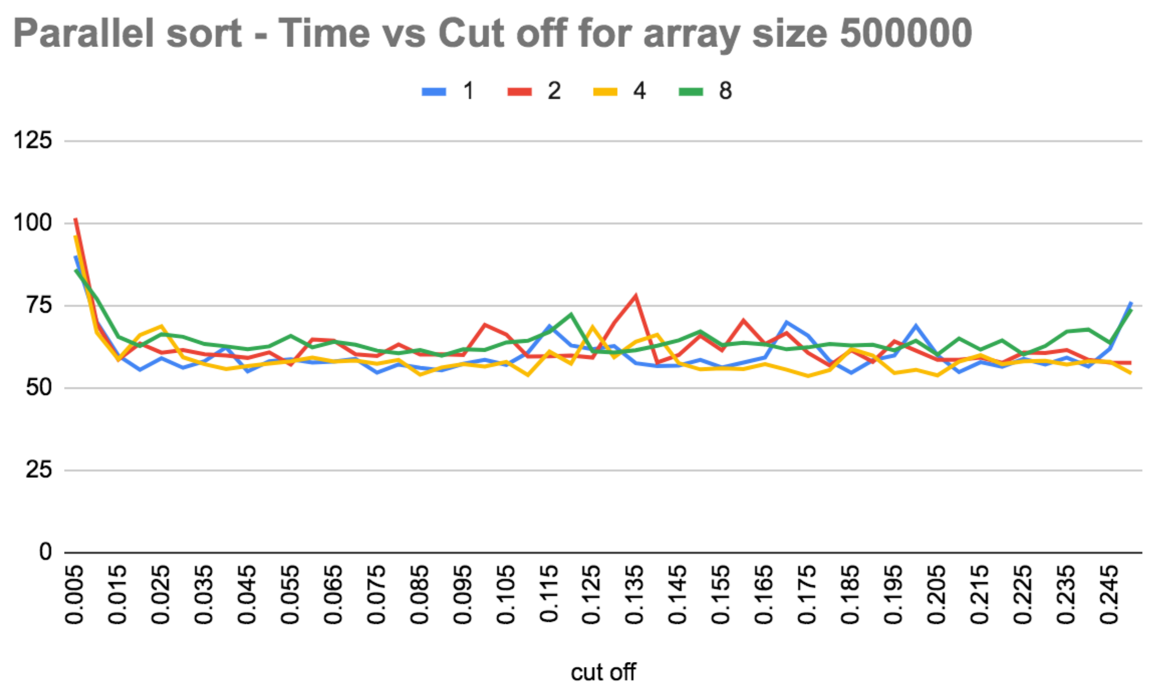


```java
    private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool pool) {
        return CompletableFuture.supplyAsync(
                () -> {
                    int[] result = new int[to - from];
                    // TO IMPLEMENT
                    System.arraycopy(array, from, result, destPos: 0, result.length);
                    sort(result, from: 0, to: to - from, pool);
                    return result;
                }, pool
        );
    }
}
```

## Graphical Representation:

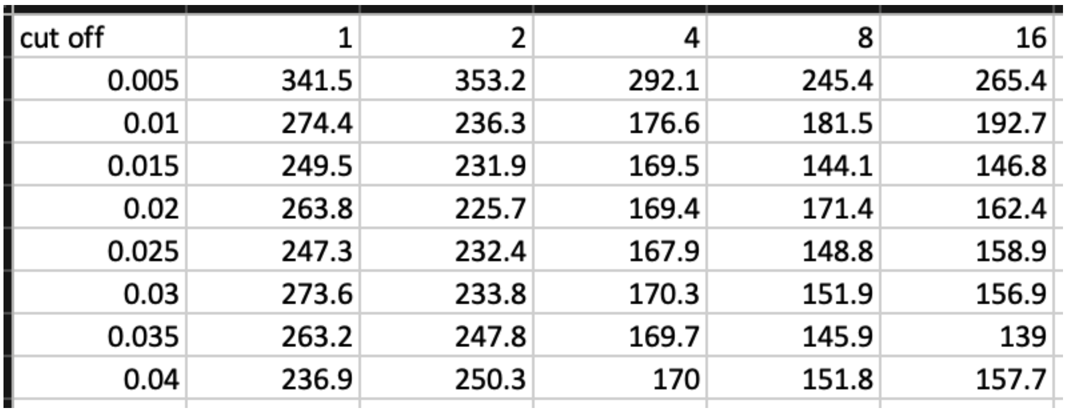


The time to sort is halved for all the thread counts except for 1



With a 500000 sized array, the time to sort for all thread counts is fairly similar

Tabular representation



| cut off | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 0.005 | 341.5 | 353.2 | 292.1 | 245.4 | 265.4 |
| 0.01 | 274.4 | 236.3 | 176.6 | 181.5 | 192.7 |
| 0.015 | 249.5 | 231.9 | 169.5 | 144.1 | 146.8 |
| 0.02 | 263.8 | 225.7 | 169.4 | 171.4 | 162.4 |
| 0.025 | 247.3 | 232.4 | 167.9 | 148.8 | 158.9 |
| 0.03 | 273.6 | 233.8 | 170.3 | 151.9 | 156.9 |
| 0.035 | 263.2 | 247.8 | 169.7 | 145.9 | 139 |
| 0.04 | 236.9 | 250.3 | 170 | 151.8 | 157.7 |

- The single thread was always the slowest for all array sizes
- When the thread is greater than 8 the time does not decrease because the system used is a 4 core 8 thread CPU
- Combining the parallel sort and cutoff shows better performance, as comparing the result of 1 thread to 8 threads, the 8 thread performs better