Program Structures and Algorithms
Spring 2023(SEC –03)

NAME: Pranav Khismatrao
NUID: 002746375

Assignment - 5

Task:

1. 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.
2. 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.
3. 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).
4. 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

Relationship Conclusion:

The observation suggest that there are several factors that can influence the time required to sort an array using parallel merge sort.

First, increasing the cutoff value can lead to a decrease in the time needed to sort the array, as shown in graph.

Second, increasing the number of threads used in the sorting process can also reduce the time required to sort the array.

However, thirdly, as the size of the array increases, the time required to sort the array also increases. This is due to the time complexity of parallel merge sort, which is O(nlogn).

Therefore, to optimize the performance of parallel merge sort, it is important to carefully consider these factors and to adjust the cutoff value and number of threads used to ensure efficient sorting for arrays of various sizes.

Evidence to support that conclusion:

Please find the below observations:

1. Array Size : 100000

Cut Off	Time ms (Thread 2)	Thrread 4	Thread 8	Thread 16	Thread 32	Thread 64	Thread 128
5,10,000	685	551	329	298	277	193	181

5,50,000	196	182	63	66	61	61	66
6,00,000	188	187	65	70	62	60	66
6,50,000	192	186	59	61	58	62	64
7,00,000	187	183	58	64	59	59	61
7,50,000	193	188	60	61	58	61	62
8,00,000	191	185	59	70	57	63	60
8,50,000	188	192	58	63	60	64	61
9,00,000	185	183	59	106	62	59	60
9,50,000	187	186	61	64	60	66	62
10,00,000	185	186	56	64	71	64	62

2. Array Size : 200000

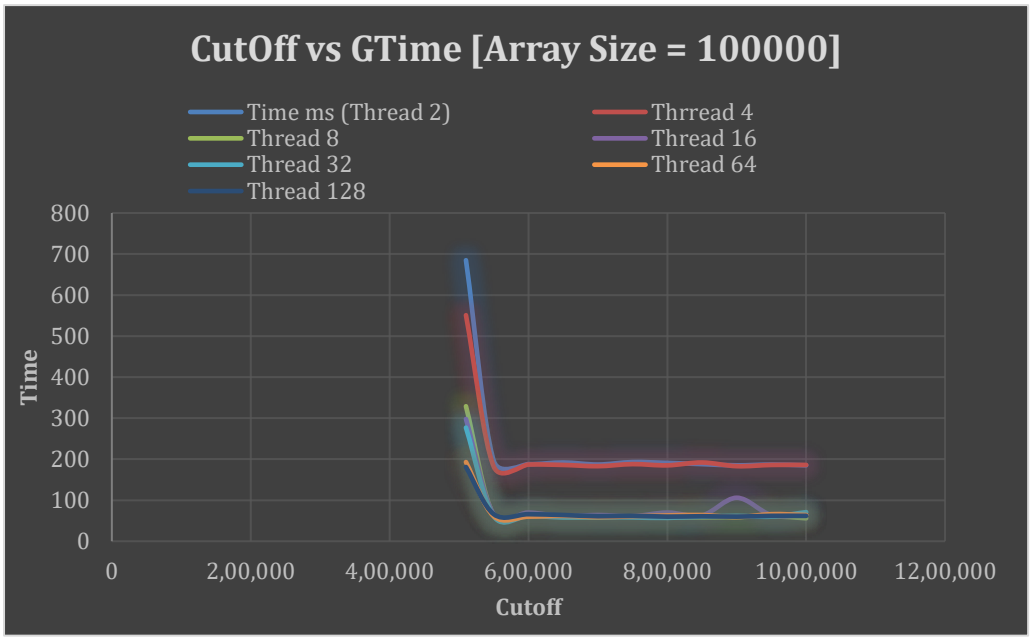
Cut Off	Time ms (Thread 2)	Thread 4	Thread 8	Thread 16	Thread 32	Thread 64	Thread 128
5,10,000	356	366	440	408	316	424	344
5,50,000	129	129	131	137	126	137	134
6,00,000	121	124	124	126	119	139	126
6,50,000	124	125	127	124	123	134	124
7,00,000	124	126	121	127	127	133	122
7,50,000	129	122	124	125	124	139	124
8,00,000	124	121	135	120	123	129	122
8,50,000	124	122	124	127	123	132	122
9,00,000	125	120	121	127	123	128	130
9,50,000	121	126	121	125	126	147	126
10,00,000	122	124	128	128	124	128	128

3. Array Size : 300000

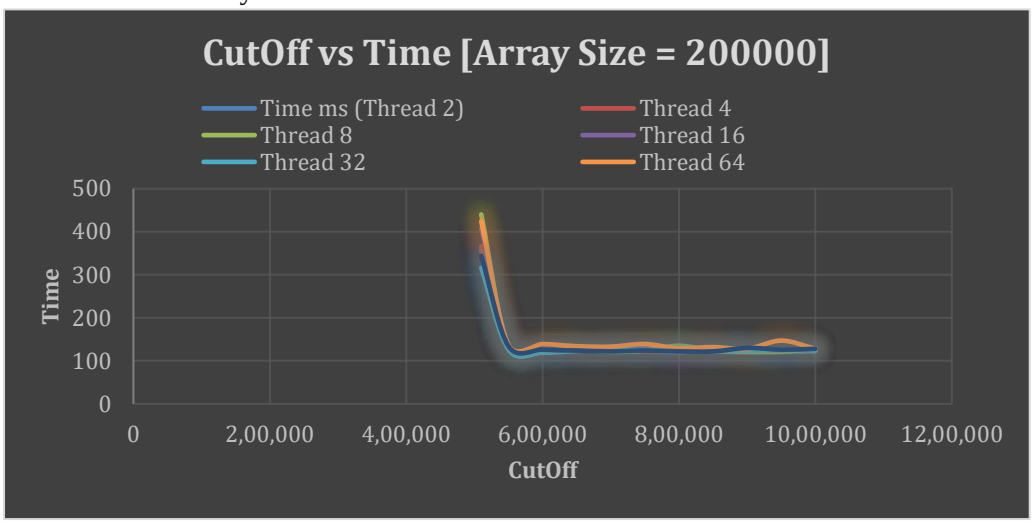
Cut Off	Time ms (Thread 2)	Thread 4	Thread 8	Thread 16	Thread 32	Thread 64	Thread 128
5,10,000	468	357	364	349	406	382	333
5,50,000	190	208	195	198	197	194	197
6,00,000	186	189	194	193	191	186	189
6,50,000	203	191	194	187	191	183	188
7,00,000	185	187	193	193	191	186	187
7,50,000	191	190	189	187	190	185	194
8,00,000	188	190	199	190	191	183	189
8,50,000	224	194	189	183	189	186	188
9,00,000	188	187	186	191	191	186	185
9,50,000	189	196	188	188	189	187	190
10,00,000	217	190	188	188	191	188	186

Graphical Representation:

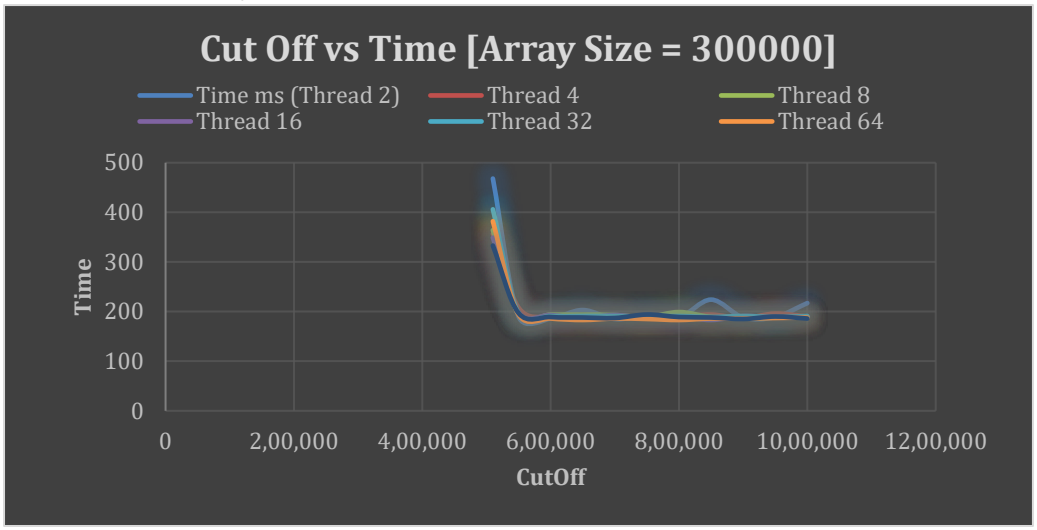
1. CutOff Vs Time for Array Size 10000:



2. CutOff vs Time Array Size 200000:



3. CutOff vs Time Array Size 300000



Unit Test Screenshots:

No Unit test in this Assignment

Code Snippets:

1. Added executor thread function to calculate relationship I changed the nThreads number to calculate observations and their relationships

```
2 usages  Pranav014 *
private static CompletableFuture<int[]> parsort(int[] array, int from, int to) {
    Executor executor = Executors.newFixedThreadPool( nThreads: 128);
    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);
                return result;
            }
    , executor);
}
}
```

2. Sort Method:

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


        CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {
            int[] result = new int[xs1.length + xs2.length];
            // TO IMPLEMENT
            int i = 0;
            int j = 0;
            for (int k = 0; k < result.length; k++) {
                if (i >= xs1.length) {
                    result[k] = xs2[j++];
                } else if (j >= xs2.length) {
                    result[k] = xs1[i++];
                } else if (xs2[j] < xs1[i]) {
                    result[k] = xs2[j++];
                } else {
                    result[k] = xs1[i++];
                }
            }
            return result;
        });
        parsort.whenComplete((result, throwable) -> System.arraycopy(result, srcPos: 0, array, from, result.length));
        System.out.println("# threads: "+ ForkJoinPool.commonPool().getRunningThreadCount());
        parsort.join();
    }
```

3. Main Method: (I changed array size here to determine observations for various cases depending ion array size here)

```java
Pranav014 *
public class Main {

    Pranav014 *
    public static void main(String[] args) {
        processArgs(args);
        System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
            Random random = new Random();
            int[] array = new int[300000];
            ArrayList<Long> timeList = new ArrayList<>();
            for (int j = 50; j < 100; j++) {
                ParSort.cutoff = 10000 * (j + 1);
                // 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);
                }
                long endTime = System.currentTimeMillis();
                time = (endTime - startTime);
                timeList.add(time);

//              System.out.println("cutoff: " + (ParSort.cutoff) + "\t\t10times Time:" + time + "ms");
                System.out.println(time);

            }
            try {
                FileOutputStream fis = new FileOutputStream( name: "./src/result.csv");
                OutputStreamWriter isr = new OutputStreamWriter(fis);
                BufferedWriter bw = new BufferedWriter(isr);
```