Program Structures and Algorithms
Assignment-5

Summer-2022

Dimpleben Kanjibhai Patel – 002965372

Task-1:

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.
For varying the number of threads available, you might want to consult the following resources:

Code:

```
    public static void main(String[] args) {
        processArgs(args);
//        System.out.println("Degree of parallelism: " +
ForkJoinPool.getCommonPoolParallelism());
        Random random = new Random();
        for(int size = 1000000; size < 16000000; size *= 2) {
            int[] array = new int[size];
            ArrayList<Long> timeList = new ArrayList<>();
            System.out.println("Array Size :" + size);
            ParSort.thread_count = 1;
            while (ParSort.thread_count <= 16) {
                ParSort.thread_Pool = new ForkJoinPool(ParSort.thread_count);
                System.out.println("Degree of parallelism: " +
ParSort.thread_Pool.getParallelism());
```

```
                for (int j = 0; j < 20; j++) {
//            ParSort.cutoff = 10000 * (j + 1);
                    ParSort.cutoff = size / 20 * (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(10000000);
                        ParSort.sort(array, 0, array.length);
                    }
                    long endTime = System.currentTimeMillis();
                    time = (endTime - startTime);
                    timeList.add(time);


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

                }
                try {
                    FileOutputStream fis = new FileOutputStream("./src/result" + size + "-" +
ParSort.thread_count + ".csv");
                    OutputStreamWriter isr = new OutputStreamWriter(fis);
                    BufferedWriter bw = new BufferedWriter(isr);
                    int j = 0;
                    for (long i : timeList) {
                        String content = (double) size / 20 * (j + 1) + "," + (double) i / 10
+ "\n";

                        j++;
                        bw.write(content);
                        bw.flush();
                    }
                    bw.close();

                } catch (IOException e) {
                    e.printStackTrace();
                }
                ParSort.thread_count *= 2;
            }
        }

    }
```

Parsort.java

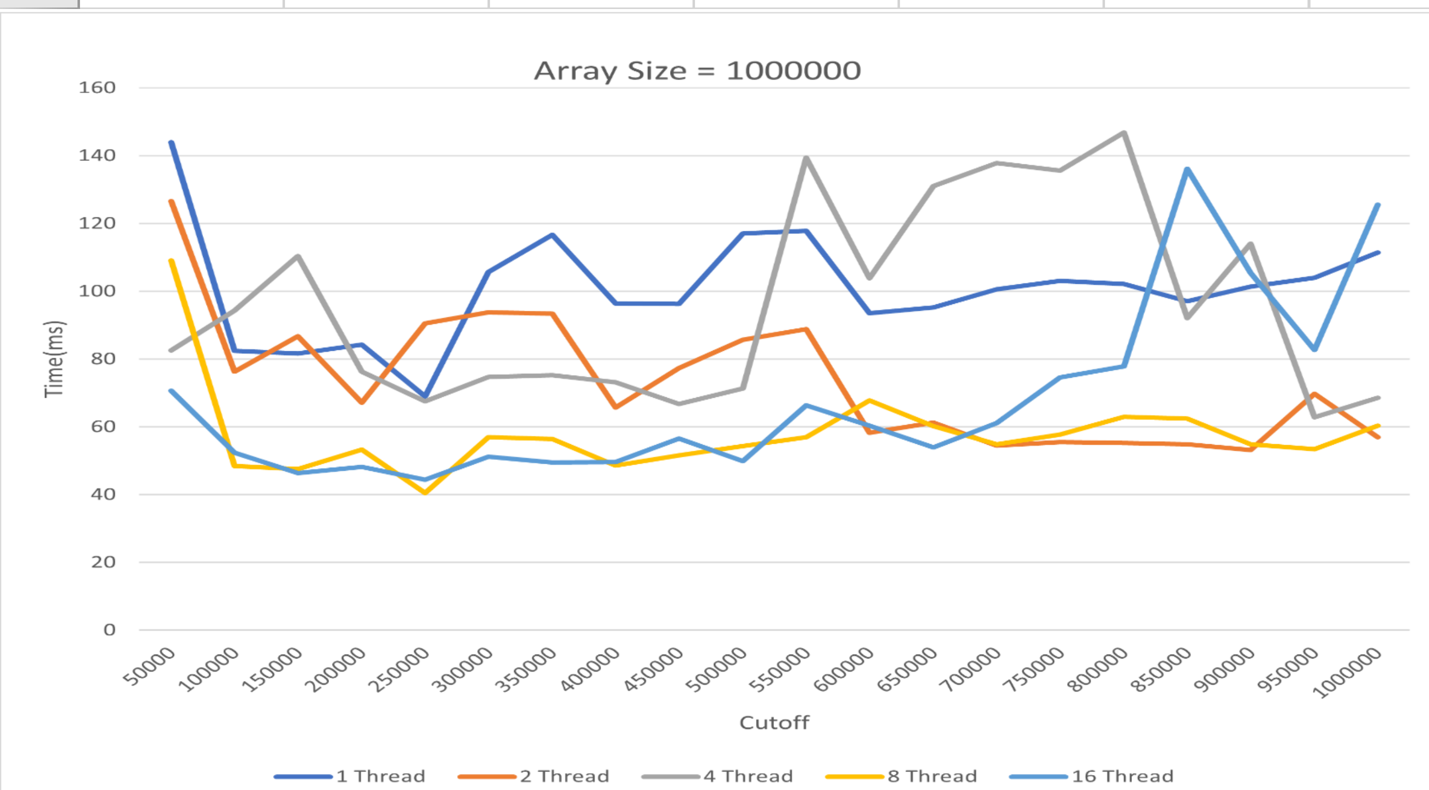
```
    public static int cutoff = 1000;
    public static int thread_count = 2;
    public static ForkJoinPool thread_Pool = new ForkJoinPool(thread_count);

    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[]> parsort1 = parsort(array, from, from + (to - from) / 2);
// TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, from + (to - from) / 2, to); //
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;
                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, 0, array,
from, result.length));
//              System.out.println("# threads: "+
ForkJoinPool.commonPool().getRunningThreadCount());
            parsort.join();
        }
    }

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

Comparison Graph:

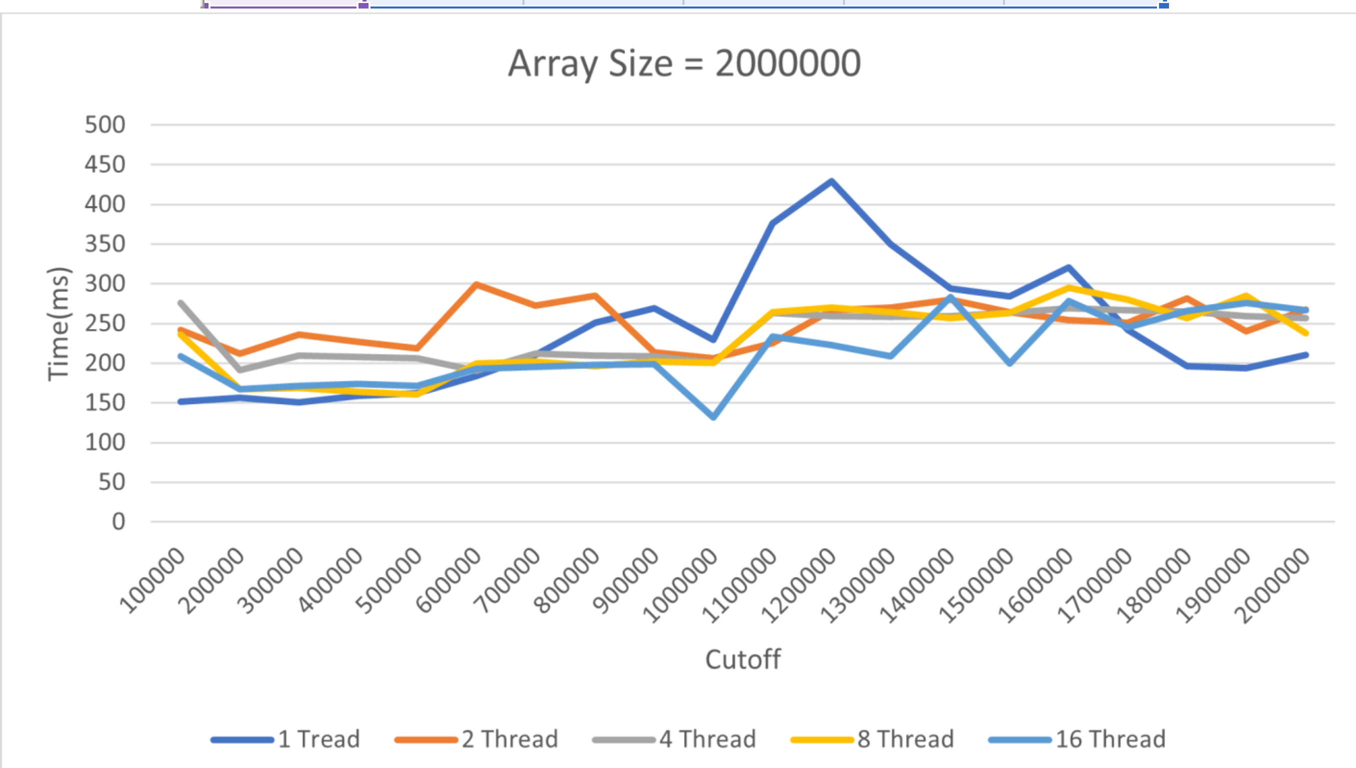
Graph for array size = 1000000, and Threads value= 1,2,4,8,16

Cutoff	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
50000	144	126.6	82.5	109.1	70.7
100000	82.4	76.3	94.3	48.4	52.4
150000	81.7	86.7	110.4	47.5	46.4
200000	84.2	67.1	76.3	53.3	48.2
250000	69	90.5	67.5	40.5	44.4
300000	105.7	93.8	74.7	56.9	51.2
350000	116.7	93.4	75.2	56.4	49.5
400000	96.4	65.7	73.1	48.6	49.7
450000	96.3	77.4	66.8	51.6	56.5
500000	117	85.7	71.3	54.4	49.9
550000	117.8	88.8	139.4	56.9	66.4
600000	93.6	58.2	103.9	67.8	60.4
650000	95.2	61.3	131.1	60.2	53.9
700000	100.6	54.5	137.8	54.8	61.1
750000	103.1	55.5	135.6	57.8	74.6
800000	102.2	55.2	146.9	63	77.9
850000	97.1	54.9	92.1	62.4	136.2
900000	101.4	53.2	114.1	54.9	105.4
950000	104	69.7	62.8	53.4	82.7
1000000	111.4	57	68.6	60.4	125.6



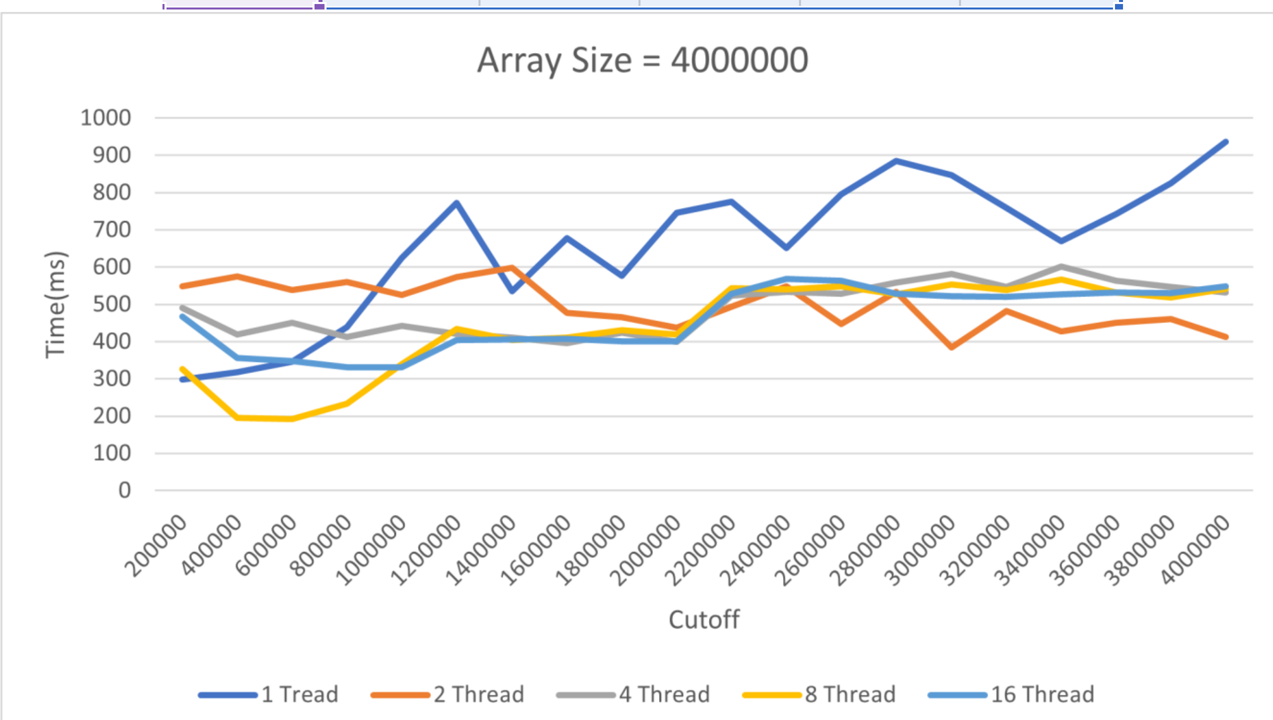
Graph for array size = 2000000, and Threads value= 1,2,4,8,16

Cutoff	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
100000	151.2	241.8	276	236.1	209
200000	156.6	211.7	190.9	167.2	167.2
300000	150.4	235.9	209.2	168.8	171.1
400000	159.2	226.6	207.7	163.6	174.1
500000	162.4	218.5	205.8	160.6	171.1
600000	183.7	299	190.3	199.7	192.8
700000	210.4	272.8	211.6	202.1	195
800000	251	284.8	209.6	196.4	197.5
900000	269.4	213.4	208.9	201.9	198.7
1000000	229.1	206.3	200.6	200.3	131.2
1100000	376.2	224.9	263	263.8	233.2
1200000	429.2	266.9	259	269.7	222.8
1300000	349.7	270	258.4	263.9	209
1400000	293.8	280.1	258.9	257.1	283.7
1500000	284.1	263.8	263.1	263.1	199.9
1600000	320.4	254.1	269	295.1	278.6
1700000	241.9	251.2	266.6	279.6	245.5
1800000	195.9	281.8	264.7	256.4	265.6
1900000	193.8	240.3	258.9	284.8	275.9
2000000	210.7	267.6	256.9	237.4	266.9



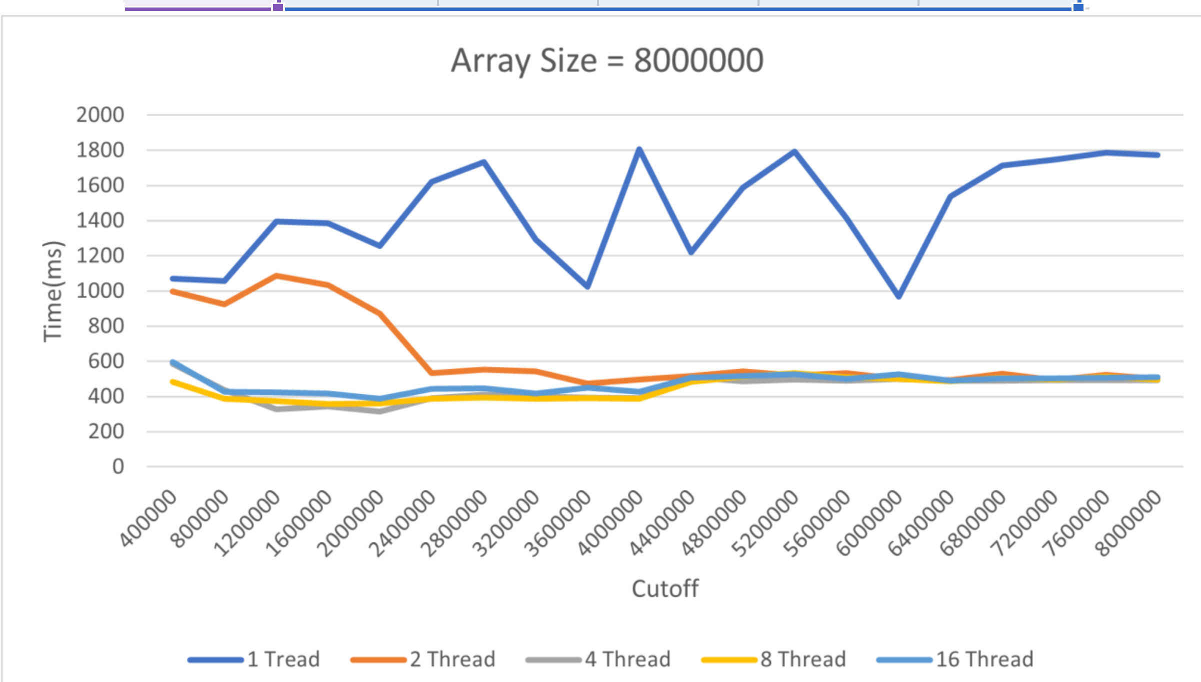
Graph for array size = 4000000, and Threads value= 1,2,4,8,16

Cutoff	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
200000	298	549.1	490.9	326.7	466.6
400000	317.7	574.9	419.3	195.5	355.8
600000	346.8	538.9	451.2	192.2	347.2
800000	438.2	559.9	411.5	233.1	330.4
1000000	623.8	525.1	442.2	339.9	331.7
1200000	772.2	572.6	420	433.7	403.8
1400000	535.4	598.5	410.2	403.9	405.9
1600000	677	477.6	396.5	410.5	407.4
1800000	576.7	464.6	424	430.1	399.9
2000000	745	437.2	398.3	418.8	401.1
2200000	775.7	494.1	524.2	543.4	528.3
2400000	650.5	548.8	533.2	540	568.8
2600000	795.9	446.7	528.1	547.9	563.9
2800000	884.7	534.1	559	527.3	528.1
3000000	847.5	384.5	581.7	553.8	522.4
3200000	759.1	482.7	546	539.1	520
3400000	668.7	427.2	600.7	566.3	526.2
3600000	741.6	450.5	563.9	532.1	532.2
3800000	824.5	459.7	547.3	518.2	529.8
4000000	936.6	412.5	531.9	542.1	549



Graph for array size = 8000000, and Threads value= 1,2,4,8,16

Cutoff	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread
400000	1070	995.6	586.5	482.4	595
800000	1056.2	925.1	437.4	387.7	425.5
1200000	1393.8	1085.7	327.5	372.1	424
1600000	1384.9	1034.1	342.1	358.1	415.6
2000000	1255.8	871	314.1	360.5	386.1
2400000	1620.2	533.8	388.9	387	444.8
2800000	1734.2	551.5	409.4	394.3	445
3200000	1292.2	541.9	399	387.3	416.6
3600000	1022.4	473.1	394.2	389.2	449.9
4000000	1805.5	497.3	391.6	386.4	427.5
4400000	1220.1	516.5	508.8	483.4	504.8
4800000	1586.8	543.4	487.2	512	515.3
5200000	1791.9	519.8	495.1	531.9	527.4
5600000	1416.1	532.9	489.1	508.1	499.5
6000000	967.4	498.8	498.4	498.6	525
6400000	1538.2	492.6	488.7	487.9	489.4
6800000	1714.8	529	489.6	507.4	502.3
7200000	1747.2	492.1	493.8	495.6	503.8
7600000	1787.3	524	492.4	512.9	504.8
8000000	1772.5	498	491.9	495.2	508.1



Conclusion :

- From the above simulation, it can be inferred that multi-threading is better than single thread.
- For 2 Threads, good cutoff value is at approximately 50% of the array size and for 4 Threads, good cutoff value is at approximately 25% of the array size.
- Thus, At my observation, Good cutoff can be obtained at

  cutoff = Array Size / Number of Threads
- In my laptop(Windows i5 16GB RAM) best result is obtained for the number of threads = 4