Program Structures and Algorithms
Spring 2023(SEC – 01)

NAME: Kasavaraju Venkata Pranay Kumar
NUID: 002701155

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 cut-off (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 cut-off. If there are fewer elements to sort than the cut-off, 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 cut-off schemes

Relationship Conclusion:
The observation suggests that there are several factors that can influence the time required to sort an array using parallel merge sort.
First, increasing the cut-off 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 cut-off value and number of threads used to ensure efficient sorting for arrays of various sizes.

Evidence to support that conclusion:
Observations cut-off vs time:

1)Array size: 100000

Cut-off	Thread 2(ms)	Thread 4(ms)	Thread 8(ms)	Thread 16(ms)	Thread 32(ms)	Thread 64(ms)
510000	144	167	148	129	159	136
560000	53	59	58	55	59	58
610000	51	55	53	53	51	56
660000	50	52	53	51	50	53
710000	49	51	49	53	51	50
760000	48	49	49	51	49	49
810000	48	50	49	51	49	51
860000	48	50	47	49	49	51
910000	49	48	48	49	47	50
960000	47	48	48	50	46	49

2)Array size: 300000

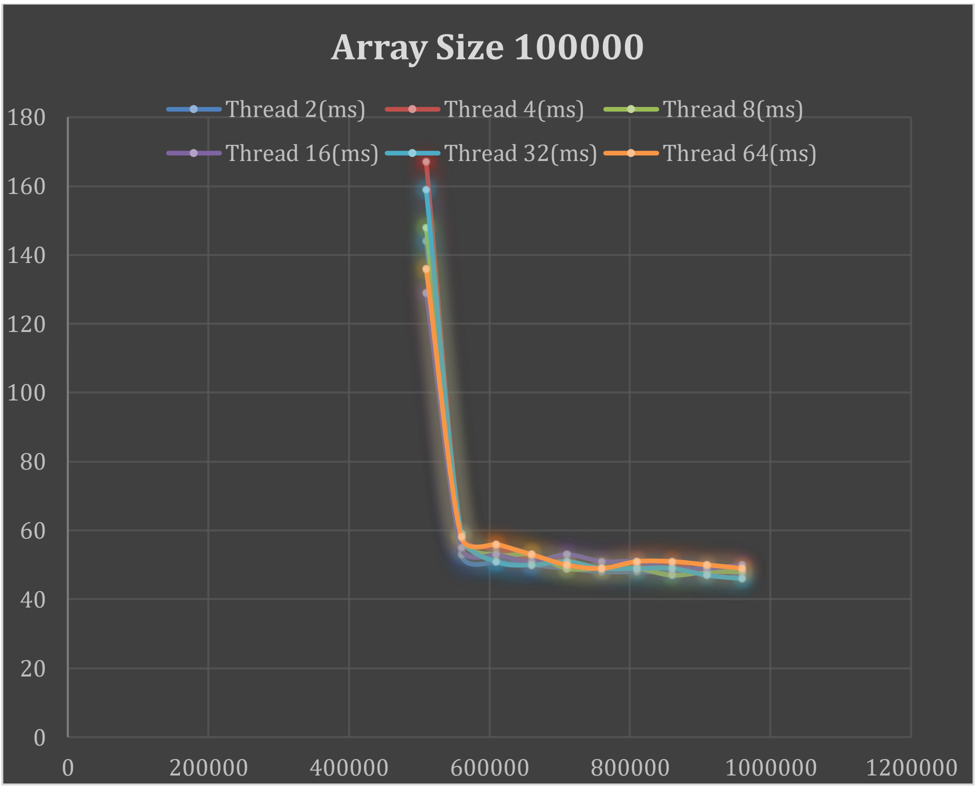
Cut-off	Thread 2(ms)	Thread 4(ms)	Thread 8(ms)	Thread 16(ms)	Thread 32(ms)	Thread 64(ms)
510000	299	277	282	300	340	335
560000	223	211	197	215	201	196
610000	211	202	204	204	196	199
660000	209	199	199	193	205	196
710000	193	201	196	207	205	198
760000	216	195	199	215	198	198
810000	205	201	187	213	203	201
860000	206	201	202	209	198	199
910000	199	196	196	201	198	193
960000	198	201	201	216	202	196

3)Array size: 500000

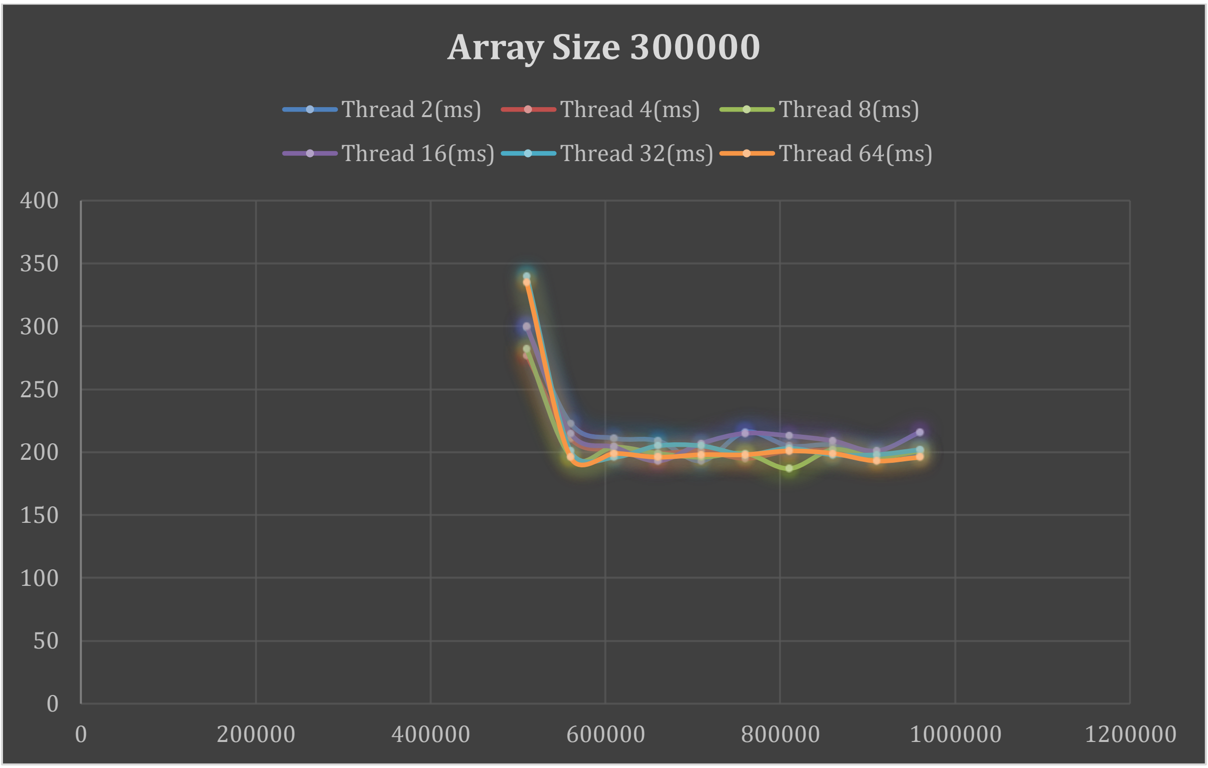
Cut-off	Thread 2(ms)	Thread 4(ms)	Thread 8(ms)	Thread 16(ms)	Thread 32(ms)	Thread 64(ms)
510000	440	477	481	447	488	484
560000	354	362	352	356	345	344
610000	342	357	348	338	343	352
660000	342	364	363	355	348	338
710000	343	373	365	347	352	372
760000	357	382	350	346	344	367
810000	355	362	342	349	351	350
860000	346	354	361	355	357	344
910000	348	365	350	353	357	348
960000	345	364	351	363	352	345

Graphical Representation:

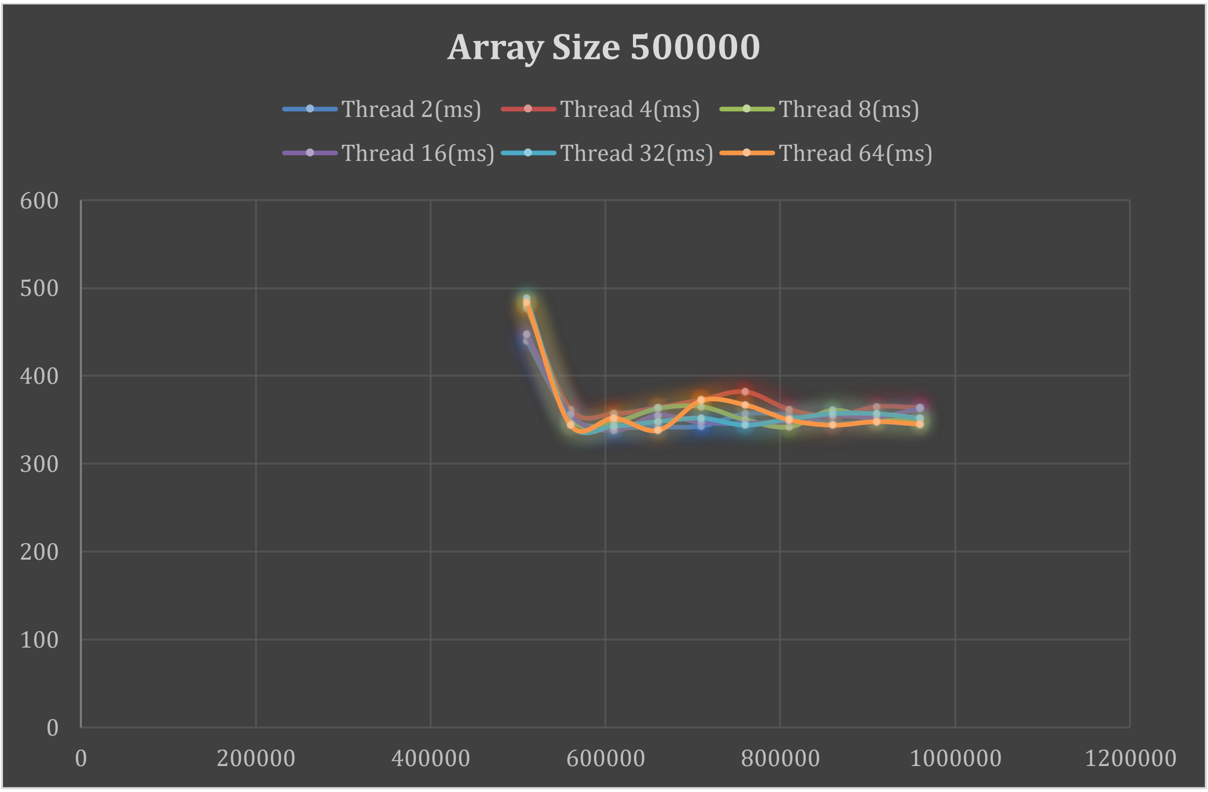
1. Cut-off Vs Time for Array Size 100000:



2. Cut-off Vs Time for Array Size 300000:



3. Cut-off Vs Time for Array Size 500000:



Unit Test Screenshots:

No unit tests were present in this assignment

Code Snippets:

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

```
private static CompletableFuture<int[]> parsort(int[] array, int from, int to) {
    Executor executor = Executors.newFixedThreadPool( nThreads: 64);
    return CompletableFuture.supplyAsync(
            () -> {
                int[] result = new int[to - from];
                // TO IMPLEMENT
                System.arraycopy(array,                                          h);
                sort(result,  from: 0,  t
                return result;
            }
    , executor);
}
}
```

```
int to
INFO6205
```

2. Sort Method:

```
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,  to: from + (to - from) / 2); // TO IMPLEMENT
        CompletableFuture<int[]> parsort2 = parsort(array,  from: 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,  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 on array size here)

```java
public static void main(String[] args) {
    processArgs(args);
    System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
    Random random = new Random();
    int[] array = new int[500000];
    ArrayList<Long> timeList = new ArrayList<>();
    for (int j = 50; j < 100; j+=5) {
        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");

    }
    try {
        FileOutputStream fis = new FileOutputStream( name: "./src/result.csv");
        OutputStreamWriter isr = new OutputStreamWriter(fis);
        BufferedWriter bw = new BufferedWriter(isr);
        int j = 0;
        for (long i : timeList) {
            String content = (double) 10000 * (j + 1) / 2000000 + "," + (double) i / 10 + "\n";
            j++;
            bw.write(content);
            bw.flush();
        }
        bw.close();
```