Program Structures and Algorithms

Spring 2023(SEC – 1)

NAME: Venkat Pavan Munaganti

NUID: 002722397

Task: Assignment 5: **Parallel Sorting**

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.

Experimental Analysis:

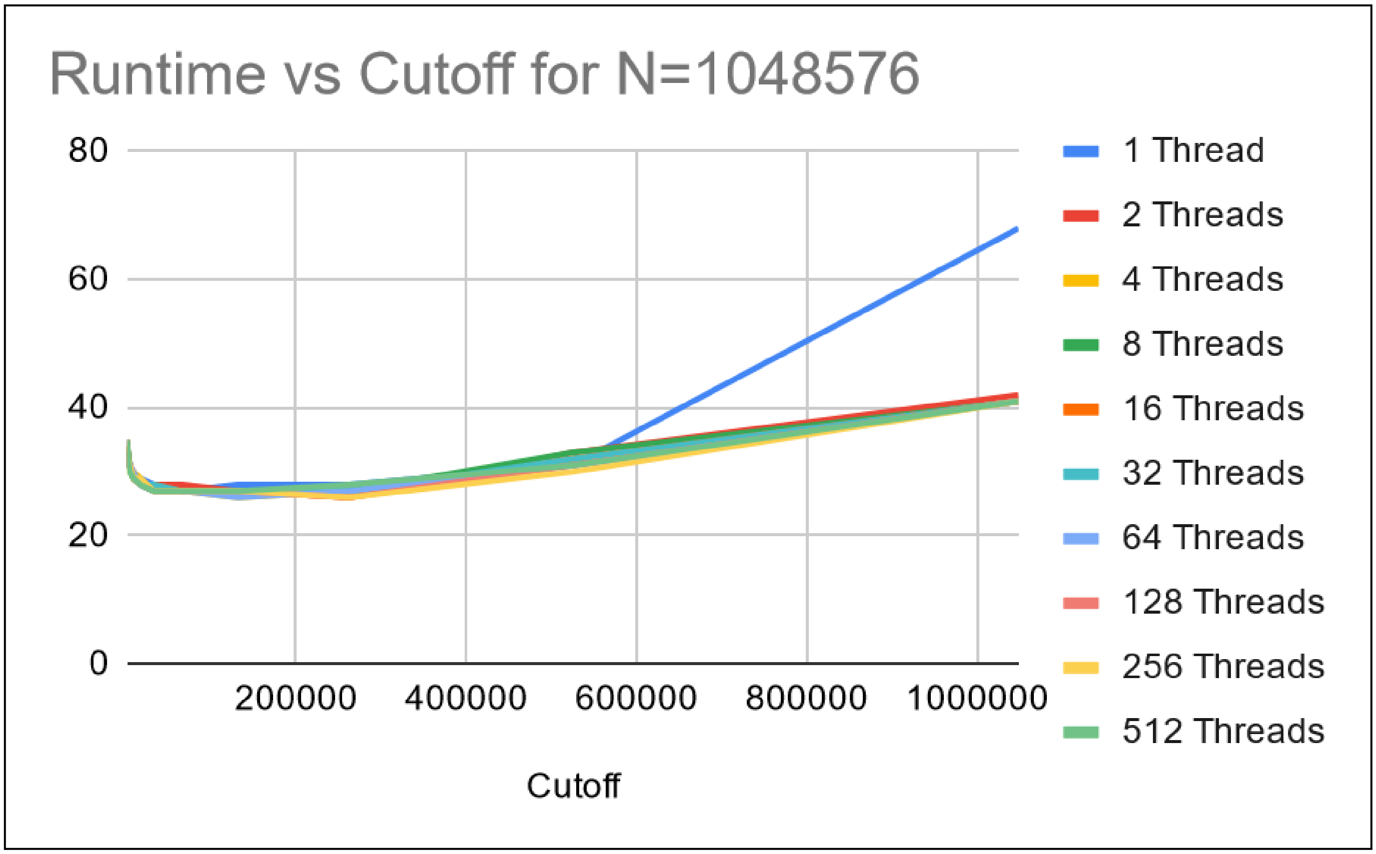
Ran the experiment with different cut off values and threads to find out which combination gives the best runtime. Below are the inputs and different values used for the experiment

1. **Input size**: Ran the experiment with different yet sufficiently large input sizes ranging from 2^{20} to 2^{24}
2. **Cut off values:** Used different cut off values for each input size. Incrementing cut off values in decrementing fractions of powers of 2 to input size
3. **Threads:** Incrementing the threads in powers of 2 for each input size to find the optimal thread allocation for parallel sorting

Below are the results of the experiment,

For input size ***N= 1048576*** ,

Parallel sort for input N=1048576										
Cutoff	**1 T**	**2 T**	**4 T**	**8 T**	**16 T**	**32 T**	**64 T**	**128 T**	**256 T**	**512 T**
1048576	68	42	41	41	41	41	41	41	41	41
524288	31	33	32	33	32	32	31	31	30	31
262144	28	26	27	27	26	27	27	26	26	28
131072	28	27	26	27	27	27	**26**	27	27	27
65536	27	28	27	27	27	27	27	27	27	27
32768	28	28	27	27	28	28	27	27	27	27
16384	29	28	29	28	28	28	28	28	29	28
8192	29	30	30	29	29	29	30	29	29	29
4096	31	31	30	31	31	31	31	31	31	30
2048	33	32	33	33	31	32	31	32	32	32
1024	33	35	34	33	34	35	33	34	34	35
Average	**33.18181**	**30.90909**	**30.54545**	**30.54545**	**30.36363**	**30.63636**	**30.18181**	**30.27272**	**30.27272**	**30.45454**

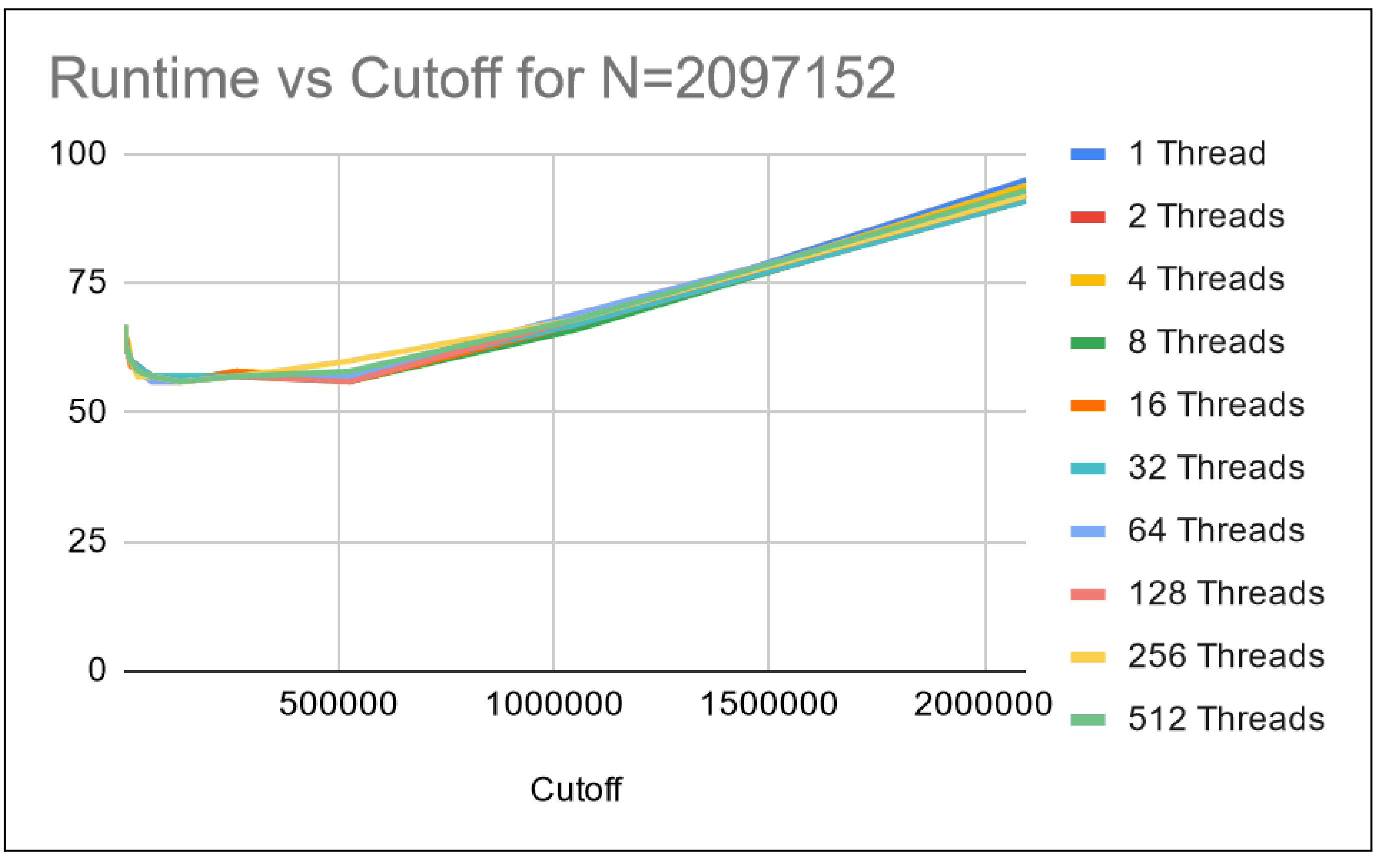


Analysis:

- Input size in this run is 2^{20}
- In above case, single thread is performing worse compared to multiple threads
- **Best cutoff:** Runtime is minimal when cutoff is almost equal to ⅛ of input size, i.e, N/8
- **Best no of threads:** 64 Threads is giving the best runtime in parallel sort
- Since input size is comparatively less all threads above 4 are performing equally
- **Best combination:** When both methods are combined, threads in range 8 and 256, and cut off values in range 2^2 to 2^8 are giving overall best runtime

For input size ***N= 2097152,***

Parallel sort for input N=2097152										
Cutoff	**1 T**	**2 T**	**4 T**	**8 T**	**16 T**	**32 T**	**64 T**	**128 T**	**256 T**	**512 T**
2097152	95	93	94	92	91	91	92	92	92	93
1048576	67	68	67	66	67	67	69	68	68	68
524288	57	56	57	56	56	57	57	56	60	58
262144	57	58	58	57	58	57	**56**	57	57	57
131072	57	56	56	57	56	57	56	56	56	56
65536	57	56	57	57	56	57	56	57	57	57
32768	59	59	58	59	58	59	56	58	57	58
16384	60	60	60	60	59	60	60	60	60	60
8192	62	62	64	62	62	62	62	63	62	62
4096	63	63	64	64	66	63	64	64	63	63
2048	66	66	67	67	66	67	66	64	67	67
Average	**63.63636**	**63.36363**	**63.81818**	**63.36363**	**63.18181**	**63.36363**	**63.09090**	**63.18181**	**63.54545**	**63.54545**

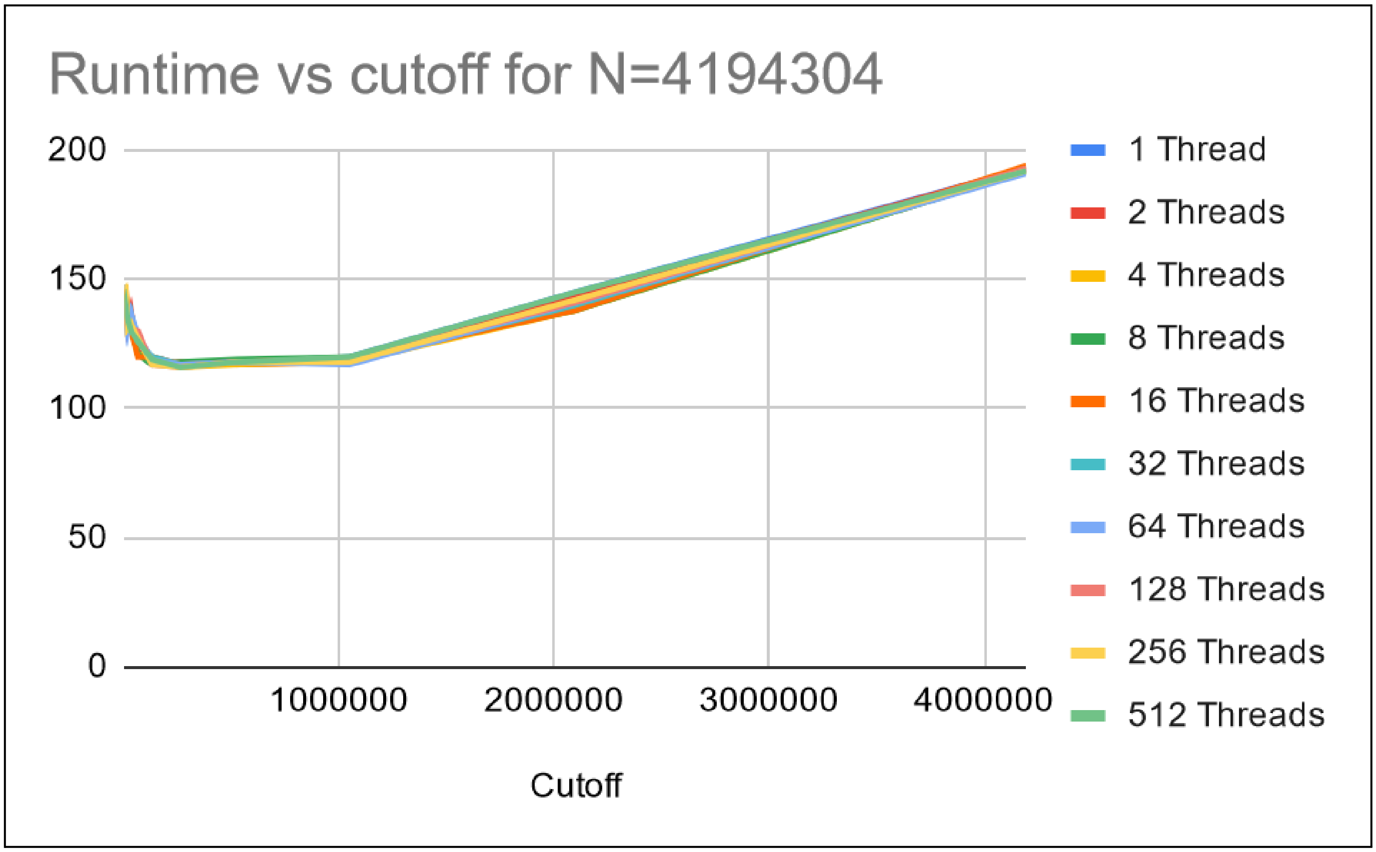


Analysis:

- Input size in this run is 2^{21}
- In above case, single thread is performing worse compared to multiple threads
- **Best cutoff:** Runtime is minimal when cutoff is almost equal to ⅛ of input size, i.e, N/8
- **Best no of threads:** 64 Threads is giving the best runtime in parallel sort
- **Best combination:** When both methods are combined, threads in range 32 and 256, and cut off values in range $N/2^3$ to $N/2^6$ are giving overall best runtime

<u>For input size ***N= 4194304,***</u>

Parallel sort for input N=4194304										
Cutoff	**1 T**	**2 T**	**4 T**	**8 T**	**16 T**	**32 T**	**64 T**	**128 T**	**256 T**	**512 T**
4194304	193	193	192	192	194	192	191	193	192	192
2097152	145	144	138	138	138	140	141	141	142	145
1048576	120	118	118	120	119	118	117	119	118	120
524288	118	117	117	119	118	118	**117**	118	118	118
262144	117	118	116	118	117	117	118	116	116	116
131072	120	117	120	117	120	120	119	117	117	119
65536	122	122	121	121	120	126	128	130	127	126
32768	135	138	131	131	130	137	137	131	133	130
16384	139	132	137	136	135	133	130	132	131	135
8192	143	140	144	140	146	142	135	139	148	140
4096	148	145	143	144	140	139	141	141	139	138
Average	**136.3636**	**134.9090**	**134.2727**	**134.1818**	**134.2727**	**134.7272**	**134**	**134.2727**	**134.6363**	**134.4545**

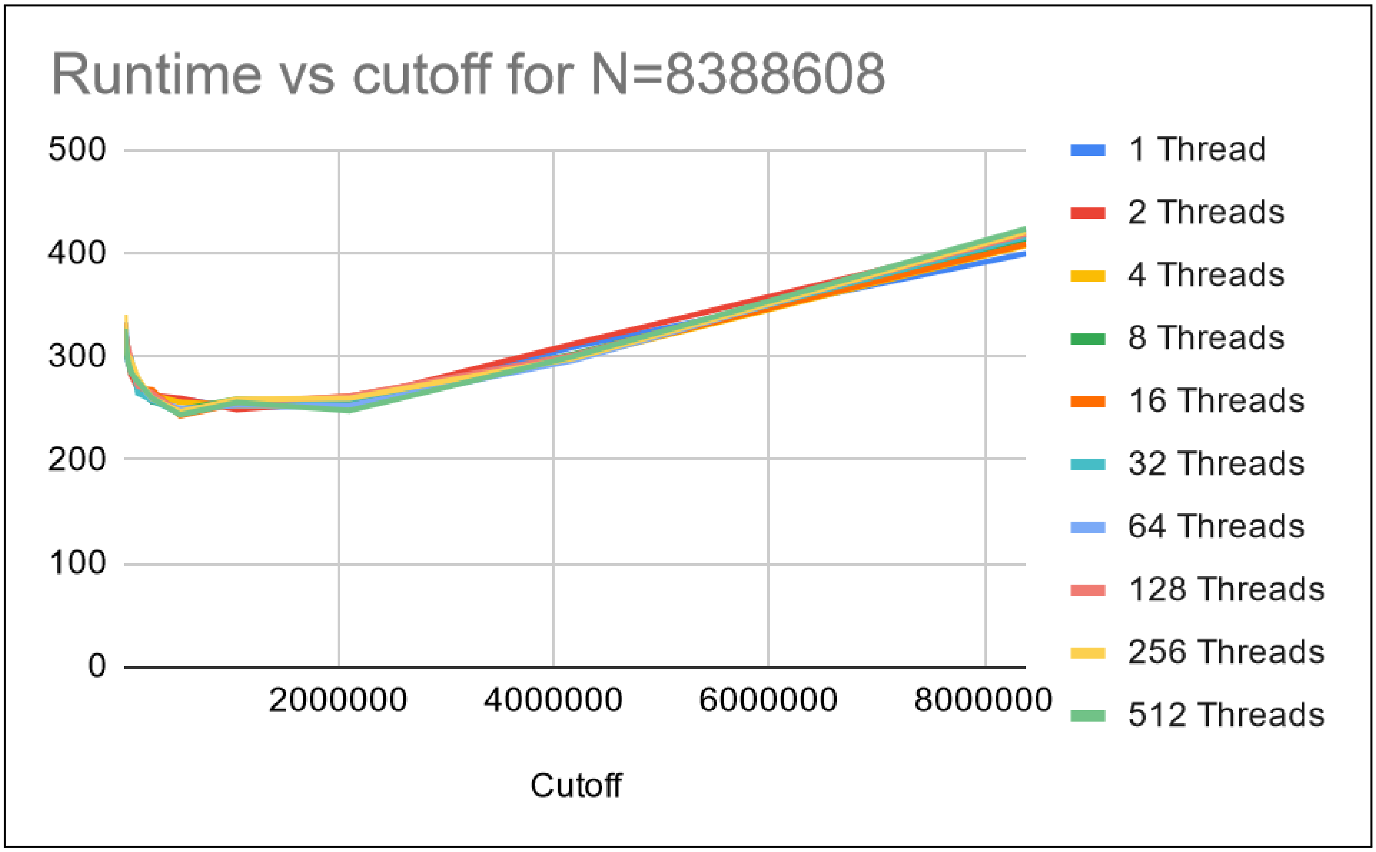


Analysis:

- Input size in this run is 2^{22}
- In above case, single thread is performing worse compared to multiple threads
- **Best cutoff:** Runtime is minimal when cutoff is almost equal to ⅛ of input size, i.e, N/8
- **Best no of threads:** 64 Threads is giving the best runtime in parallel sort
- **Best combination:** When both methods are combined, threads in range 16 and 256, and cut off values in range $N/2^2$ to $N/2^3$ are giving overall best runtime

For input size ***N= 8388608,***

Parallel sort for input N=8388608										
Cutoff	1 T	2 T	4 T	8 T	16 T	32 T	64 T	128 T	256 T	512 T
8388608	400	418	408	414	409	416	422	419	421	424
4194304	310	313	299	303	300	301	297	302	299	301
2097152	253	258	252	256	262	259	253	262	260	248
1048576	251	249	253	259	255	255	250	256	259	256
524288	259	260	256	249	243	245	253	246	247	244
262144	257	263	263	257	268	258	259	265	260	260
131072	278	269	269	272	270	265	275	272	279	276
65536	287	283	293	293	291	299	287	296	297	285
32768	297	305	306	299	310	308	310	311	302	298
16384	329	310	311	318	321	318	319	318	313	320
8192	334	324	329	327	333	329	326	326	341	327
Average	**295.9090**	**295.6363**	**294.4545**	**295.1818**	**296.5454**	**295.7272**	**295.5454**	**297.5454**	**298**	**294.4545**



Analysis:

- Input size in this run is 2^{23}
- In above case, single thread is performing worse compared to multiple threads
- **Best cutoff:** Runtime is minimal when cutoff is almost equal to ⅛ of input size, i.e, N/8
- **Best no of threads:** 64 Threads is giving the best runtime in parallel sort
- **Best combination:** When both methods are combined, 64 threads, and cut off values $N/2^3$ is giving the best runtime

<u>Conclusion</u>:

- The experiment is performed in i7 11th generation, with 16 cores and 16th threads, when under turbo boost and full performance it could run efficiently till 64 threads considering the background tasks
- From all the above cases it is evident that runtime is minimal when cut off value is ***⅛ of the input size***
- Runtime is minimal with ***64 Threads***
- As the size of the array is increasing the runtime is minimal at cut off ***⅛ of the input size***
- As the size of the array is increasing the runtime is minimal at cut off ***64 Threads***

Main.java

```java
import java.util.*;
import java.util.concurrent.Executor;
import java.util.concurrent.ForkJoinPool;

/**
 * This code has been fleshed out by Ziyao Qiao. Thanks very much.
 * CONSIDER tidy it up a bit.
 */
xiaohuanlin *
public class Main {
    xiaohuanlin *
    public static void main(String[] args) {
        processArgs(args);
        System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
        Random random = new Random();
        HashMap<Integer, List<List<Double>>> results = new HashMap<>();
        //Different array lengths in powers of 2
        for (int a = 20; a < 24; a++) {

            //Calculating the array size
            int size = (int) Math.pow(2, a);
            int[] array = new int[size];
            ArrayList<Long> timeList = new ArrayList<>();
            List<List<Double>> runtimes = new ArrayList<>();
            System.out.println("Parallel sort for input size N="+size);

            //Different cutt off values using doubling method, starting from 10000
            for (int j = 0; j <=10; j++) {
                List<Double> runtime = new ArrayList<>();
                ParSort.cutoff = size/ (int)Math.pow(2, j);
                runtime.add((double) ParSort.cutoff);
                System.out.print(ParSort.cutoff+",");
                //Running on different threads in powers of 2 from 2^0 to 2^5 that is 1 to 32 threads
                for (int tp = 0; tp < 10; tp++) {
                    ForkJoinPool executor = new ForkJoinPool((int) Math.pow(2, tp));
                    long time;
                    long startTime = System.currentTimeMillis();

                    //10 runs to take average run time
                    for (int t = 0; t < 10; t++) {
                        //Filling the array with random values using random class
```

```java
            for (int t = 0; t < 10; t++) {
                //Filling the array with random values using random class
                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);
            runtime.add((double) (time / 10));
            System.out.print((double) (time / 10)+",");
        }
        System.out.println();
        runtimes.add(runtime);
    }
    System.out.println();
    System.out.println();
    results.put(size, runtimes);
}

try {
    FileOutputStream fis = new FileOutputStream( name: "./src/parallelSort.txt");
    OutputStreamWriter isr = new OutputStreamWriter(fis);
    BufferedWriter bw = new BufferedWriter(isr);

    StringBuilder sb = new StringBuilder();
    for (Map.Entry<Integer, List<List<Double>>> e : results.entrySet()) {
        int size = e.getKey();
        List<List<Double>> runtimes = e.getValue();
        sb.append("parallel sort for input size N=" + size + "\n");
        for (List<Double> runtime : runtimes) {
            for(var i: runtime){
                sb.append(i+",");
            }
            sb.append("\n");
        }
        sb.append("\n\n");
    }
    bw.write(sb.toString());
    bw.flush();
    bw.close();
} catch (IOException e) {
```

```java
            bw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    1 usage   xiaohuanlin
    private static void processArgs(String[] args) {
        String[] xs = args;
        while (xs.length > 0)
            if (xs[0].startsWith("-")) xs = processArg(xs);
    }

    1 usage   xiaohuanlin
    private static String[] processArg(String[] xs) {
        String[] result = new String[0];
        System.arraycopy(xs, 2, result, 0, xs.length - 2);
        processCommand(xs[0], xs[1]);
        return result;
    }

    1 usage   xiaohuanlin
    private static void processCommand(String x, String y) {
        if (x.equalsIgnoreCase("N")) setConfig(x, Integer.parseInt(y));
        else
            // TODO sort this out
            if (x.equalsIgnoreCase("P")) //noinspection ResultOfMethodCallIgnored
                ForkJoinPool.getCommonPoolParallelism();
    }

    1 usage   xiaohuanlin
    private static void setConfig(String x, int i) {
        configuration.put(x, i);
    }

    /MismatchedQueryAndUpdateOfCollection/
    private static final Map<String, Integer> configuration = new HashMap<>();


}
```

```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" ...
Degree of parallelism: 15
Parallel sort for input size N=1048576
1048576,63.0,46.0,45.0,45.0,45.0,44.0,44.0,45.0,44.0,44.0,
524288,34.0,33.0,33.0,33.0,32.0,33.0,33.0,34.0,32.0,32.0,
262144,27.0,27.0,27.0,27.0,28.0,30.0,27.0,27.0,27.0,28.0,
131072,28.0,28.0,27.0,27.0,27.0,27.0,30.0,27.0,28.0,27.0,
65536,28.0,28.0,27.0,27.0,28.0,28.0,29.0,27.0,27.0,27.0,
32768,28.0,28.0,28.0,28.0,28.0,28.0,28.0,28.0,28.0,28.0,
16384,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,29.0,
8192,29.0,30.0,30.0,30.0,30.0,29.0,30.0,30.0,29.0,31.0,
4096,31.0,31.0,31.0,30.0,31.0,31.0,31.0,31.0,31.0,31.0,
2048,33.0,33.0,33.0,32.0,33.0,32.0,32.0,32.0,32.0,33.0,
1024,34.0,33.0,33.0,33.0,33.0,33.0,32.0,33.0,33.0,33.0,


Parallel sort for input size N=2097152
2097152,95.0,93.0,94.0,92.0,91.0,91.0,92.0,92.0,92.0,93.0,
1048576,67.0,68.0,67.0,66.0,67.0,67.0,69.0,68.0,68.0,68.0,
524288,57.0,56.0,57.0,56.0,56.0,57.0,57.0,56.0,60.0,58.0,
262144,57.0,58.0,58.0,57.0,58.0,57.0,57.0,57.0,57.0,57.0,
131072,57.0,56.0,56.0,57.0,56.0,57.0,56.0,56.0,56.0,56.0,
65536,57.0,56.0,57.0,57.0,56.0,57.0,56.0,57.0,57.0,57.0,
32768,59.0,59.0,58.0,59.0,58.0,59.0,58.0,58.0,57.0,58.0,
16384,60.0,60.0,60.0,60.0,59.0,60.0,60.0,60.0,60.0,60.0,
8192,62.0,62.0,64.0,62.0,62.0,62.0,62.0,62.0,62.0,62.0,
4096,63.0,63.0,64.0,64.0,65.0,63.0,64.0,64.0,63.0,63.0,
2048,66.0,66.0,67.0,67.0,66.0,67.0,66.0,64.0,67.0,67.0,


Parallel sort for input size N=4194304
4194304,193.0,193.0,192.0,192.0,194.0,192.0,191.0,193.0,192.0,192.0,
2097152,145.0,144.0,138.0,138.0,138.0,140.0,141.0,141.0,142.0,145.0,
1048576,120.0,118.0,118.0,120.0,119.0,118.0,117.0,119.0,118.0,120.0,
524288,118.0,117.0,117.0,119.0,118.0,118.0,118.0,118.0,118.0,118.0,
262144,117.0,118.0,116.0,118.0,117.0,117.0,117.0,116.0,116.0,116.0,
131072,120.0,117.0,120.0,117.0,120.0,120.0,119.0,117.0,117.0,119.0,
65536,122.0,122.0,121.0,121.0,120.0,126.0,128.0,130.0,127.0,126.0,
```

ParSort.java

```java
xiaohuanlin
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();
    }
}
```

```java
2 usages  xiaohuanlin
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, destPos: 0, result.length);
                sort(result, from: 0, to: to - from);
                return result;
            }
    );
}
```