Ebenezer Ajay Williams Vincent Sankey Raj (002166250)

Program Structures & Algorithms

Fall 2021

Assignment No. 4(Parallel Sort)

- **Task (List of tasks performed in the Assignment)**
    1) **Code added to the Main.java and ParSort.java classes .**
    2) **Experiment is performed with 1,2,4,8 and 16 threads with array sizes 5* 10^6 , 1 * 10^7 and 2 * 10^7.**
    3) **Graph is plotted to deduce the conclusion.**

- **Relationship Conclusion:**

  The relationship between the time taken to sort (t) and Threads (R) decreases with increasing threads, however it depends on the number of threads that can be supported by the CPU.

  If the CPU can support N number of threads for an array of size 2 * 10^7 , N number of threads will work fastest while sorting in parallel.

  The best scenario would be to configure the algorithm with thread count equal to the number of system threads and a cut off ranging between 600K - 900K

- **Evidence to support the conclusion:**

1. **Output (Snapshot of Code output in the terminal)**

```
cutoff: 830000      10times Time:592ms
cutoff: 840000      10times Time:699ms
cutoff: 850000      10times Time:658ms
cutoff: 860000      10times Time:582ms
cutoff: 870000      10times Time:546ms
cutoff: 880000      10times Time:584ms
cutoff: 890000      10times Time:598ms
cutoff: 900000      10times Time:688ms
cutoff: 910000      10times Time:599ms
cutoff: 920000      10times Time:548ms
cutoff: 930000      10times Time:578ms
cutoff: 940000      10times Time:564ms
cutoff: 950000      10times Time:588ms
cutoff: 960000      10times Time:571ms
cutoff: 970000      10times Time:592ms
cutoff: 980000      10times Time:565ms
cutoff: 990000      10times Time:619ms
cutoff: 1000000     10times Time:761ms
```

Code modified to get the required output on the terminal

- **main() method**

```
processArgs(args);
ForkJoinPool forkJoinPool = new ForkJoinPool( parallelism: 8);
System.out.println(Runtime.getRuntime().availableProcessors());
System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
```

```
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, forkJoinPool);
}
```

parsort() method

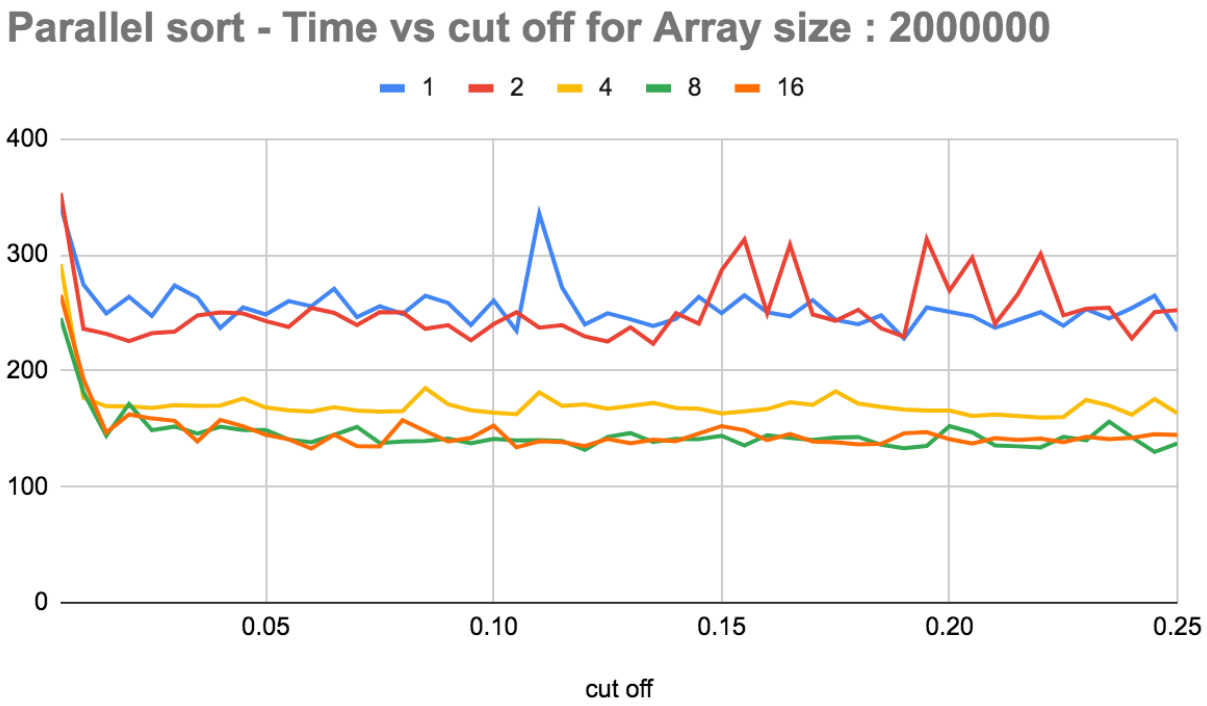
```
private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool forkjoinpool) {
    return CompletableFuture.supplyAsync(
            () -> {
                int[] result = new int[to - from];
                // TO IMPLEMENT
                System.arraycopy(array, from,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

sort() method in ParSort.java

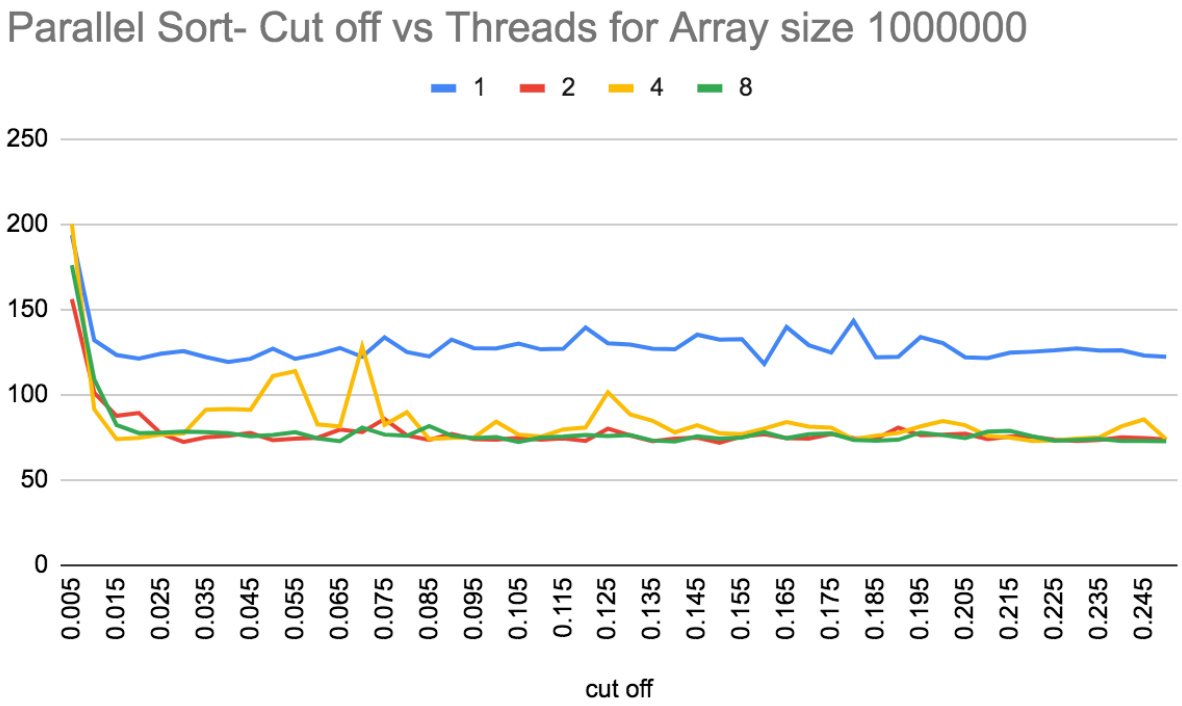
```
CompletableFuture<int[]> parsort1 = parsort(array, from, to: from + (to - from) / 2, forkjoinpool);
CompletableFuture<int[]> parsort2 = parsort(array, from: from + (to - from) / 2, to, forkjoinpool);
CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {
```

2. Graphical Representation

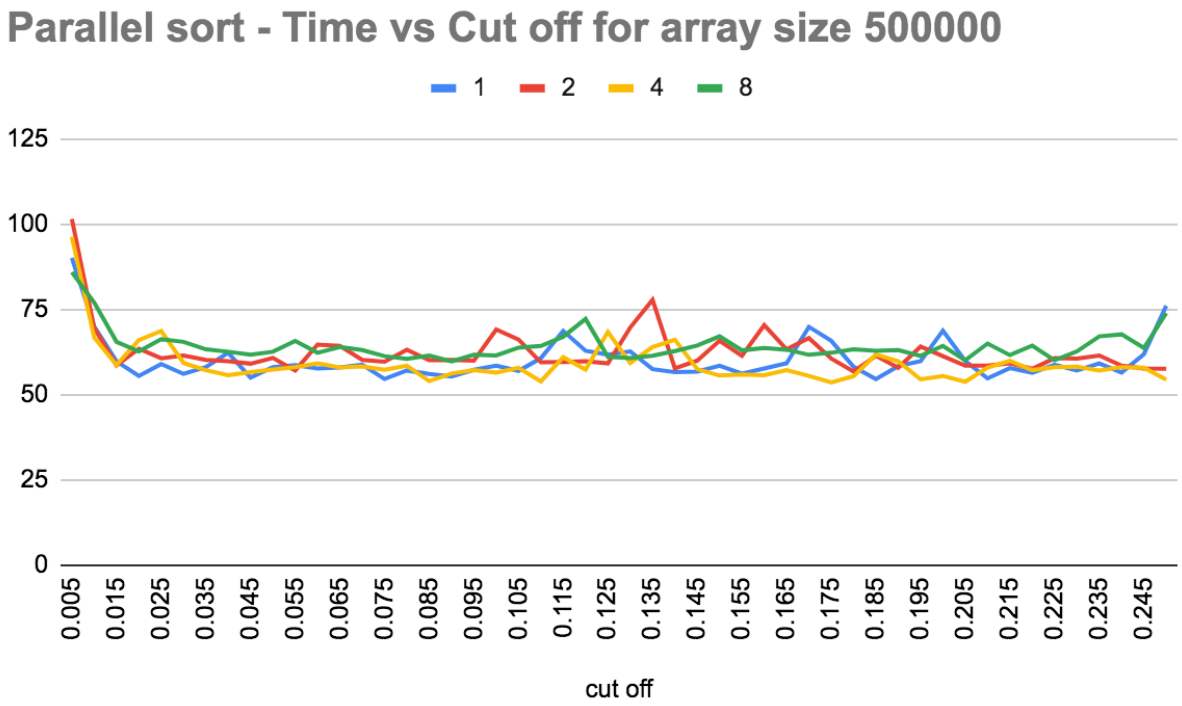
Line graph showing the relationship between the cut off value (i) vs the time taken (t) for different numbers of threads T.



We see that since the CPU can provide a maximum of 8 threads, even if the count is 16 there is no significant improvement.



With the size of the array halved the time for sorting is almost the same for any thread count other than 1.



With the size of the array halved further, we see that the the time taken to sort the array is the same for all number of threads.

Tabulated values:

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

CONCLUSION:

1) **The CPU used for his experiment has 4 cores and can provide upto 8 threads.**
2) **For all array sizes, the single thread configuration performs the worst with the slowest time.**
3) **As we see with the array size at 500K all the thread counts work similarly.There is a slight deviation after that. Upon performing multiple experiments the threshold is identified at a cut off greater than 600K approx.**
4) **When we increase the thread count from 8 to 16 , the performance reduces. This is because only 8 system threads are available.**
5) **When the size of the array is large we see that a cut off greater than 600K is applicable after which increases the time across all thread configurations.**
6) **Hence the best scenario would be to configure the algorithm with thread count equal to the number of system threads and a cut off ranging between 600K - 900K**

Final code of Assignment 4.java

```
class ParSort {
```

```
    public static int cutoff = 1000;

    public static void sort(int[] array, int from, int to, ForkJoinPool
forkjoinpool) {
        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, forkjoinpool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, from + (to -
from) / 2, to, forkjoinpool); // 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();
        }
    }
```