

Sneha Ravichandran (001096251)
Program Structures & Algorithms INFO 6205
Fall 2021
Assignment 5 (Parallel Sorting)

Task:

1. to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel
2. Choose an appropriate cut off value(take from user) for the array size such that it shifts to system sort once it goes below the cut off
3. Vary the number of threads(using forkpool) to run in parallel to accomplish the parallel sorting and also vary the array size

Part 1:

Main Program implementation with minor changes to accommodate different threads and updated the csv file output.

```
Q- processCom 2 results
1 package edu.neu.coe.info6205.sort.par;
2
3
4 import java.io.*;
5 import java.util.*;
6 import java.util.concurrent.ForkJoinPool;
7
8 /**
9  * This code has been fleshed out by Ziyao Qiao. Thanks very much.
10  * TODO tidy it up a bit.
11  */
12 public class Main {
13
14     public static void main(String[] args) {
15         processArgs(args);
16         //System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
17         Random random = new Random();
18         int[] array = new int[2000000];
19         int array_size = array.length;
20         ArrayList<Long> timeList = new ArrayList<>();
21         System.out.println("Enter the cutoff value");
22         Scanner s = new Scanner(System.in);
23         int cutoff=s.nextInt();
24         //int cutoff=Integer.parseInt(BufferedReader br= new BufferedReader(new InputStreamReader(System.in)));
25         for(int th=1;th<=16;th=th*2)
26         {
27             ForkJoinPool myPool = new ForkJoinPool(th);
28             double min = Integer.MAX_VALUE;
29             int bestCutoff = 0;
30             double avg = 0;
```

```
Q- processCom 2 results
31         for (int j = 50; j < 100; j++) {
32             ParSort.cutoff = cutoff * (j + 1);
33             // for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);
34             long time;
35             long startTime = System.currentTimeMillis();
36
37             for (int t = 0; t < 10; t++) {
38                 for (int i = 0; i < array.length; i++) array[i] = random.nextInt( bound: 10000000);
39                 ParSort.sort(array, from: 0, array.length, myPool);
40             }
41             long endTime = System.currentTimeMillis();
42             time = (endTime - startTime);
43             avg += (time/10);
44             if((time/10) < min){
45                 bestCutoff = cutoff * (j + 1);
46                 min = time/10;
47             }
48             timeList.add(time);
49             //System.out.println("cutoff: " + (ParSort.cutoff) + "\t\t10times Time:" + time + "ms");
50         }
51         System.out.println("For threads " + th + " min time = " + min + " at cutoff " + bestCutoff + " and average time is = " + avg / 50 + "ms");
52     }
53
54     try {
55         FileOutputStream fis = new FileOutputStream("name: ./src/result.csv");
56         OutputStreamWriter isr = new OutputStreamWriter(fis);
57         BufferedWriter bw = new BufferedWriter(isr);
58         int j = 50;
59         int threadcounter=1;
60         for (Long i : timeList) {
61             String content = (double) cutoff*(j+1) / array_size + "," + (double) i / 10 + "," + threadcounter + "\n";
```

```
processCom
2 results
61         j++;
62         if(j==100)
63         {
64             j=50;
65             threadcounter=threadcounter*2;
66         }
67         bw.write(content);
68         bw.flush();
69     }
70     bw.close();
71
72     } catch (IOException e) {
73         e.printStackTrace();
74     }
75 }
76
77 private static void processArgs(String[] args) {
78     String[] xs = args;
79     while (xs.length > 0)
80         if (xs[0].startsWith("-")) xs = processArg(xs);
81 }
82
83 @ private static String[] processArg(String[] xs) {
84     String[] result = new String[0];
85     System.arraycopy(xs, srcPos: 2, result, destPos: 0, length: xs.length - 2);
86     processCommand(xs[0], xs[1]);
87     return result;
88 }
89
90 @ private static void processCommand(String x, String y) {
```

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

private static void setConfig(String x, int i) { configuration.put(x, i); }
@SuppressWarnings("MismatchedQueryAndUpdateOfCollection")
private static final Map<String, Integer> configuration = new HashMap<>();
}
```

Changes made to ParSort Program to accommodate the different threads to run program in parallel.

```
package edu.neu.coe.info6205.sort.par;
import java.util.concurrent.ForkJoinPool;

import java.util.Arrays;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ForkJoinPool;

/**
 * This code has been fleshed out by Ziyao Qiao. Thanks very much.
 * TODO tidy it up a bit.
 */
class ParSort {

    public static int cutoff = 20000;

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

```

```

        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();
}

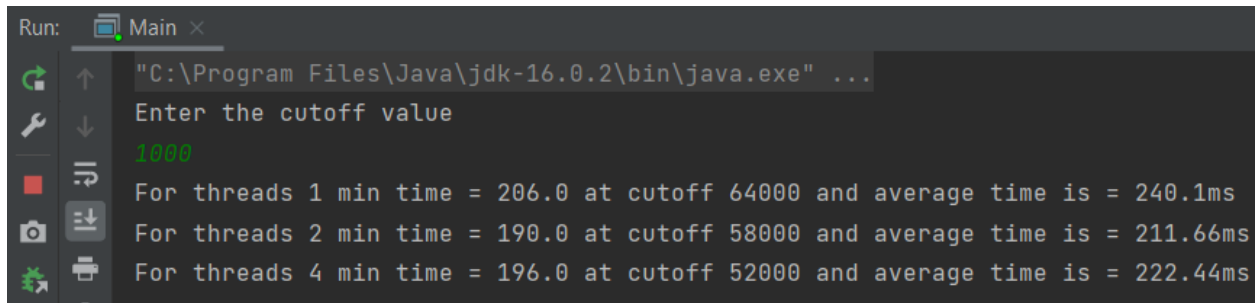
}

private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool mypool) {
    // System.out.println(from + " " + to + " thread count " + mypool.getParallelism());
    // System.out.println(from);
    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, mypool);
            return result;
        }, mypool
    );
}

```

Part2:

Taking cut off from the user.



The screenshot shows a Java IDE console window titled 'Run: Main x'. The command prompt shows the execution of 'C:\Program Files\Java\jdk-16.0.2\bin\java.exe'. The program prompts the user to 'Enter the cutoff value', and the input '1000' is shown in green. The output displays performance metrics for different thread counts: 1 thread (min time 206.0, cutoff 64000, average time 240.1ms), 2 threads (min time 190.0, cutoff 58000, average time 211.66ms), and 4 threads (min time 196.0, cutoff 52000, average time 222.44ms).

```
Run: Main x
"C:\Program Files\Java\jdk-16.0.2\bin\java.exe" ...
Enter the cutoff value
1000
For threads 1 min time = 206.0 at cutoff 64000 and average time is = 240.1ms
For threads 2 min time = 190.0 at cutoff 58000 and average time is = 211.66ms
For threads 4 min time = 196.0 at cutoff 52000 and average time is = 222.44ms
```

For data size of 0.5 million elements the different best cutoffs are at:

```
For threads 1 min time = 18.0 at cutoff 56000 and average time is = 25.98ms
For threads 2 min time = 19.0 at cutoff 52000 and average time is = 21.32ms
For threads 4 min time = 15.0 at cutoff 70000 and average time is = 18.08ms
For threads 8 min time = 14.0 at cutoff 68000 and average time is = 18.04ms
For threads 16 min time = 14.0 at cutoff 68000 and average time is = 17.16ms
```

For data size of 1 million elements the different best cutoffs are at:

```
For threads 1 min time = 40.0 at cutoff 60000 and average time is = 48.6ms
For threads 2 min time = 35.0 at cutoff 52000 and average time is = 41.5ms
For threads 4 min time = 31.0 at cutoff 90000 and average time is = 37.94ms
For threads 8 min time = 30.0 at cutoff 63000 and average time is = 37.04ms
For threads 16 min time = 30.0 at cutoff 73000 and average time is = 36.14ms
```

For data size of 2 million elements the different best cutoffs are at:

```
For threads 1 min time = 84.0 at cutoff 54000 and average time is = 110.34ms
For threads 2 min time = 74.0 at cutoff 59000 and average time is = 91.94ms
For threads 4 min time = 71.0 at cutoff 85000 and average time is = 87.0ms
For threads 8 min time = 65.0 at cutoff 92000 and average time is = 82.88ms
For threads 16 min time = 62.0 at cutoff 98000 and average time is = 75.92ms
```

For data size of 4 million elements the different best cutoffs are at:

```
For threads 1 min time = 227.0 at cutoff 62000 and average time is = 346.46ms
For threads 2 min time = 164.0 at cutoff 62000 and average time is = 281.5ms
For threads 4 min time = 184.0 at cutoff 91000 and average time is = 302.16ms
For threads 8 min time = 169.0 at cutoff 80000 and average time is = 300.42ms
For threads 16 min time = 172.0 at cutoff 52000 and average time is = 220.48ms
```

From this data we can infer that higher the parallelism better is the efficiency ie the time taken to sort the array in parallel. Thread 16 gives me a better time in comparison to threads 1 or 4 or 8.

Part 3:

The csv file output from running the program.

For 0.5 million:

cutoff/arraysize	thread1	thread2	thread4	thread8	thread16
0.102	65.1	23.2	21.1	25.4	20.2
0.104	24.7	19.8	18.6	20.2	33.6
0.106	22.6	19.7	19.5	20.1	20.2
0.108	20.4	22.3	20.1	18.2	16.6
0.11	19.3	19.8	22.2	17.4	17.4
0.112	18.8	22.8	20.7	18.3	18.6
0.114	19.8	19.6	21.3	27.4	17
0.116	20.2	20	20.6	21.2	19.2
0.118	20.5	21.3	24	17.2	24.5
0.12	20.6	21.2	21.7	26.3	22.2
0.122	19.8	19.1	18.7	22.6	18.3
0.124	19.7	19.2	20.1	19.5	20
0.126	23.5	25.6	18.3	15.8	16.5
0.128	32.8	26.7	17.1	18.5	20.1
0.13	28.1	24.1	19.7	17.3	16.4
0.132	27.9	19.5	17.6	16.4	21
0.134	28.1	20.6	16.9	15.6	20.6
0.136	30.1	21	17.5	14.9	14.9
0.138	29	21.7	17.3	14.6	15.1
0.14	28.7	21.9	15.9	14.6	17.5
0.142	28.5	21.5	16.9	15	16.9
0.144	28.2	23.8	17.1	14.3	17.7
0.146	27.5	23.8	17.2	16.4	16.4
0.148	27.2	25.9	16.3	14.6	14.9
0.15	27.2	22	16.3	18.6	15
0.152	26.9	25.4	16.7	18.3	19.4
0.154	26.7	23.3	16.2	16	19.5
0.156	26.7	20.9	16.3	15.3	16

0.158	27.2	21.5	19.2	16	15.6
0.16	27.2	23	18	27.7	14.8
0.162	27.2	19.6	16.9	20.2	19.8
0.164	26.7	19.7	17.9	19.1	18.3
0.166	26.8	19.8	18.7	16.3	18.7
0.168	26.2	19.5	22.4	16.6	16.3
0.17	26.7	21.2	19.9	18.4	16.6
0.172	26.4	22.7	18.3	19.6	16.5
0.174	26.2	27.2	18.5	19.7	16.3
0.176	26.6	27.1	18.2	14.5	15
0.178	26.3	21.8	16.4	17	16.7
0.18	27	19.9	17.9	14.2	14.6
0.182	27.3	19.9	18	15.5	14.9
0.184	28.8	19.6	18.6	15.4	14.3
0.186	26.5	19	21	19.5	15.3
0.188	24.7	26	19.1	19.8	15.2
0.19	26.8	25	16.8	29	14.7
0.192	24.9	22.9	17.2	21.6	17.5
0.194	28.3	19.2	16.4	17.3	16.4
0.196	25.5	19	17.5	16.7	14.5
0.198	24.6	20.5	18	18.6	15.8
0.2	26.5	19.8	18.8	19.1	16.4

For 1 million:

cutoff/array size	thread1	thread2	thread4	thread8	thread16
0.051	80.4	49.8	45.8	40.4	41.4
0.052	45.7	35.9	40.6	33.9	32.4
0.053	46.2	40.3	36.3	37.5	40.2
0.054	42.3	43.6	37.9	38.5	34.9
0.055	43.2	37.5	38.8	46.3	32.3
0.056	44.3	40.8	35.4	36.1	33.1
0.057	51	40.6	34.7	38.1	36.4

0.058	49.5	42.6	34.3	33.9	35.2
0.059	49.3	39.8	37.9	35.7	38.8
0.06	40.4	41.4	35.5	40.1	32.4
0.061	43.2	36.6	39.6	37.5	36.5
0.062	40	35.8	34.6	36.8	40.1
0.063	51.2	39.1	40.3	30.8	31.5
0.064	49.2	39.3	37.1	46.7	38
0.065	46.7	38.3	36	35.2	44.6
0.066	53.2	42.5	41.1	42.9	37.7
0.067	45	44	36.5	34.5	35.2
0.068	49.2	44.4	34.1	33.9	31
0.069	48.5	43.6	36.3	34.3	37.3
0.07	50.6	41.2	38.8	42	38.9
0.071	52.7	44.3	33.3	34.5	34
0.072	52.7	42.4	39.9	31.5	34.6
0.073	46.4	38.5	40.8	46	30.8
0.074	45.8	39.5	41.3	49.4	34.2
0.075	46.2	48.2	32.8	37	30.7
0.076	48	49	45.5	34.1	35.1
0.077	49	37.8	43.6	36	33.1
0.078	49.1	41.6	40.2	39.8	34.2
0.079	52.3	45.7	36.8	35.5	33.8
0.08	48	44.1	37	33.2	38.1
0.081	47.5	40.8	41.6	33.9	32.3
0.082	51.2	39.9	37	47.6	51
0.083	48.8	49.5	34.4	39.2	39
0.084	46.4	52.6	48	35.8	40
0.085	44.3	42.3	36.5	36.6	41
0.086	43.9	40.7	45.4	36.5	34.9
0.087	50.6	39.6	37.7	36.3	38.3
0.088	55.3	39.1	41.6	40	34
0.089	48.2	37.7	40.7	37	31
0.09	54.3	37.6	31.9	35.2	38.1

0.091	43.8	42.2	38.9	37.5	38.7
0.092	50.9	40.5	34.1	36.8	33.2
0.093	58.3	37.4	36.2	35.1	35.6
0.094	49.1	40	44.3	39.6	38
0.095	48.6	42.4	39.7	33	50.9
0.096	52.5	42.5	34.7	32.6	36.6
0.097	54.5	39.4	38.6	32.4	41.5
0.098	45.6	46.8	41	43.4	32.4
0.099	45.4	58.6	35.9	37.2	40.1
0.1	50.1	41.1	40.9	35.6	31.3

For 2 million:

cutoff/array size	thread1	thread2	thread4	thread8	thread16
0.0255	153.4	104.7	93.3	92.9	84.8
0.026	97.3	83.4	81.9	79.8	67.3
0.0265	92.7	75.3	96.8	82.6	76.4
0.027	84.2	84.3	74.4	75.1	73.9
0.0275	94.8	86.1	73.8	77.5	78.7
0.028	102.8	89.9	85.7	82.2	73.8
0.0285	104.3	78.7	96.5	80.4	75.3
0.029	125.2	82.9	84.4	81.6	79.3
0.0295	165.5	74.1	85.3	67.7	78.6
0.03	138.8	85	76.1	75	76.2
0.0305	95.5	86.9	90.3	80.2	73.4
0.031	103.5	78.4	97	79.1	72.7
0.0315	102.6	91.2	95.1	84.1	90.2
0.032	113	102.6	87.7	87.4	111.6
0.0325	146.2	83.3	72.5	67.9	104.2
0.033	110.2	84.8	86.3	74.7	70.6
0.0335	134.7	98	94.6	84	83.5
0.034	164.9	77.8	89.8	76.8	80.9
0.0345	140.3	84	101.3	72.4	76.7

0.035	133.6	117.6	86	78.8	74.4
0.0355	112.1	103	89.4	69.9	74.2
0.036	100.5	102.5	86.1	67.3	72.6
0.0365	105.7	103.3	75.7	82.9	74.8
0.037	95.7	93.8	84.9	71.9	75.7
0.0375	106.2	77.9	95.4	70	77.2
0.038	93.7	87.5	78.4	95.8	70.2
0.0385	107.5	130	91.3	80.4	71.8
0.039	101	89.7	88.9	75.1	71.2
0.0395	96.9	81	78	93.8	70.6
0.04	119.8	81.1	84.2	72.4	81.3
0.0405	105.3	83.9	84.7	72.9	71.3
0.041	127.4	81	103.4	72.9	78.7
0.0415	105.3	111.4	74.3	80.5	75.5
0.042	103.9	96.4	87.3	97.9	78.7
0.0425	109.6	92.3	71.8	100.1	77
0.043	89.6	92.6	83.6	90.3	80.9
0.0435	86.7	91.2	88.4	70.9	68.2
0.044	96.9	92.8	77.4	69.3	74.7
0.0445	109.8	100.2	80.2	116.2	69
0.045	109.7	88.5	82.5	108.5	71.4
0.0455	105.1	80.1	96.5	90.5	71.1
0.046	90.6	97.8	125.3	65.7	78.7
0.0465	114.2	129.7	88.8	67.2	71
0.047	101.4	96.2	87.3	100.2	70.2
0.0475	92.5	88.3	79.9	88.6	74.9
0.048	133.4	83	75.8	89.3	73.1
0.0485	106.4	86.3	95.5	110.1	77.3
0.049	104.8	91.6	88.2	120.4	62.3
0.0495	101.8	97.6	82	85.7	79.4
0.05	105.9	138	118.7	111.9	73.4

For 4 million:

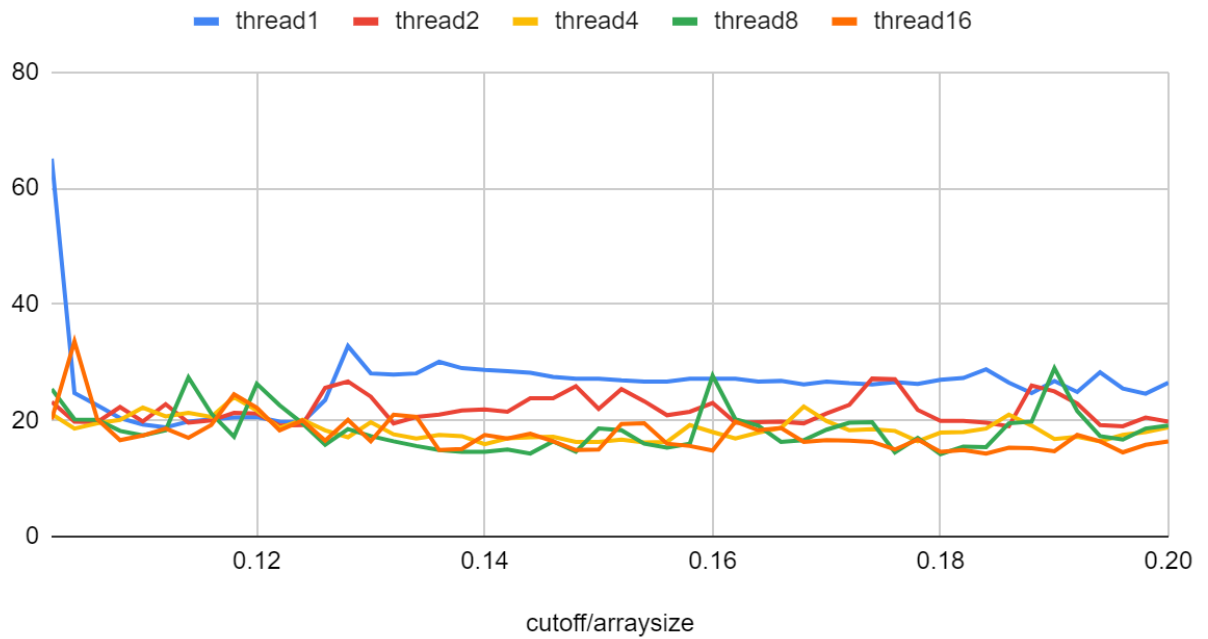
cutoff/arraysize	thread1	thread2	thread4	thread8	thread16
0.01275	218.4	162.2	226.2	173.7	181.5
0.013	150.9	188.1	177.5	148	169.1
0.01325	142.4	158.1	128.4	144.6	152.3
0.0135	177.2	269.4	165.4	218.9	172
0.01375	203.9	207.5	211.5	145.7	156.6
0.014	168.3	213.1	158.9	148.1	170.2
0.01425	171	174.5	290.6	162.6	161.7
0.0145	158.5	164.4	299.2	140.1	169.4
0.01475	150.8	157.4	305.1	155.3	161.3
0.015	194.4	163.5	257.5	143.6	180.4
0.01525	190.8	200	243.9	223.1	250.3
0.0155	176.7	162.3	265.7	187.3	180.3
0.01575	172.8	284.9	301.2	173.7	196.8
0.016	362.9	185.5	239.5	227.4	167.3
0.01625	350.7	188.5	307	211.6	194.6
0.0165	371	155.3	323.2	152	229.4
0.01675	333.4	318	333.9	217.9	159.3
0.017	306.2	279.9	480.8	232.5	147.7
0.01725	242.4	336.3	365.4	229.7	171.9
0.0175	261.2	200.4	208.7	191.3	165.1
0.01775	353	166.6	301.3	266.4	156.3
0.018	279.9	216.3	245.4	244.3	153.2
0.01825	289.1	263.7	266.1	203.8	152.4
0.0185	329	223.5	290.4	186.5	161.5
0.01875	270.5	183.3	186.9	181	141.3
0.019	283	207.2	201.8	167.2	164.1
0.01925	294.7	229.9	168.3	208.3	164.1
0.0195	285.5	170.2	219.2	204.3	140.4
0.01975	342.9	183.3	258.5	217.2	142.5
0.02	216.5	176.4	333	166	158.2

0.02025	346.1	165.2	332.2	153.7	138.8
0.0205	236.9	353	280.7	153.3	145
0.02075	222	306.3	263.6	192.6	159.7
0.021	281	179.3	252.6	179.1	160.1
0.02125	266	160.1	276.2	196.7	130.2
0.0215	200.5	172.8	239.3	157.6	143.4
0.02175	169.2	151.4	257.4	163.9	135.3
0.022	166.8	151.3	209.3	263.6	143.7
0.02225	383.4	239.7	212.2	174.9	164.8
0.0225	370.4	187.7	198.6	217.5	151.2
0.02275	328.3	186.4	225	195	221.2
0.023	356.1	302.7	292	156.4	242.2
0.02325	269.1	317.7	236.5	145.7	222.5
0.0235	366.1	228.4	271.4	200.6	269.4
0.02375	333.5	330.5	169.4	235	175.4
0.024	185.3	202.5	212.5	181.8	189.9
0.02425	259.5	220.3	224.3	155.6	175.9
0.0245	276.1	152.4	322.9	147.1	167.4
0.02475	292.1	181.7	319.7	256.4	170.3
0.025	252	256.9	246.5	286.9	147.2

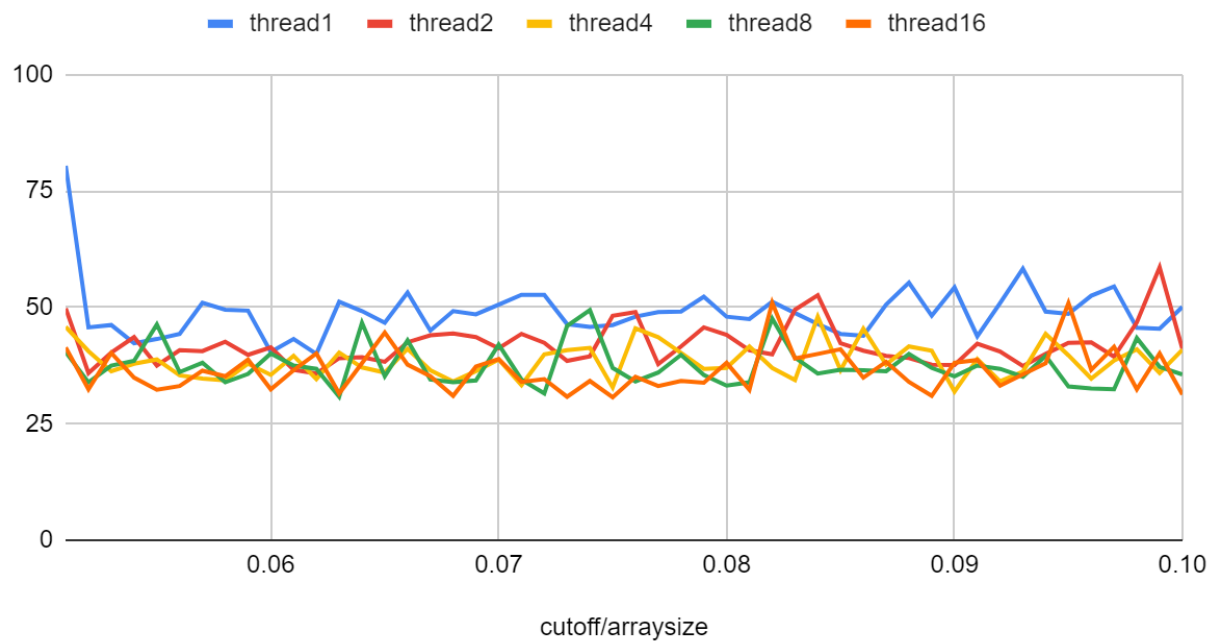
Evidence:

Comparing the graphs to observe the trends:

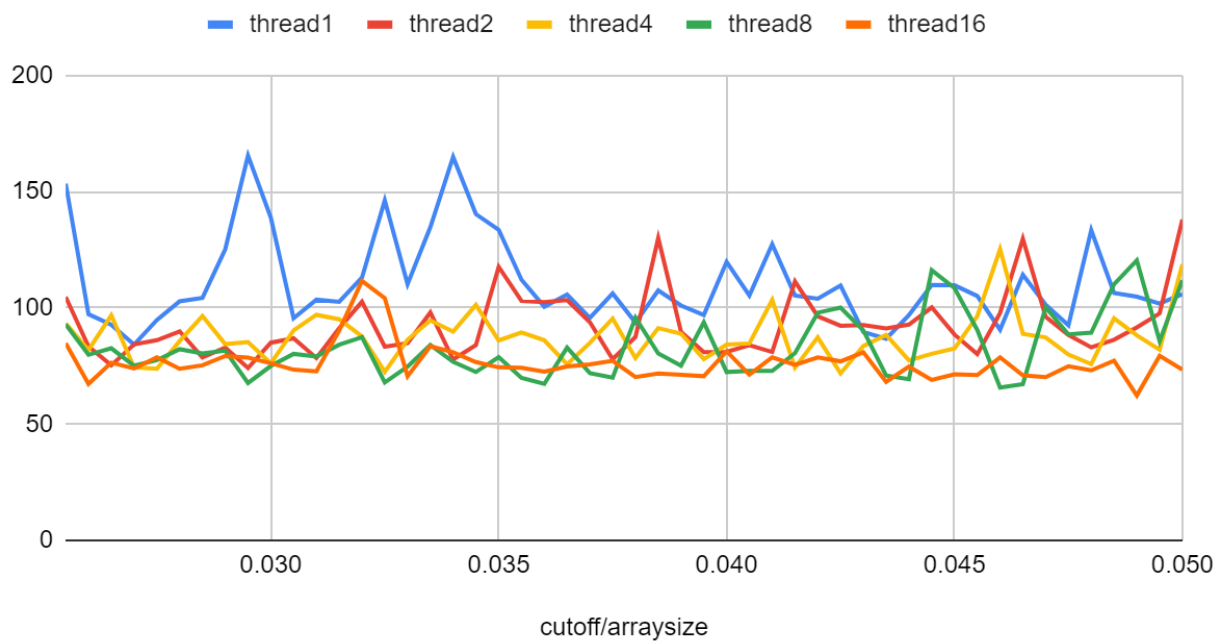
thread1, thread2, thread4, thread8 and thread16 0.5 million



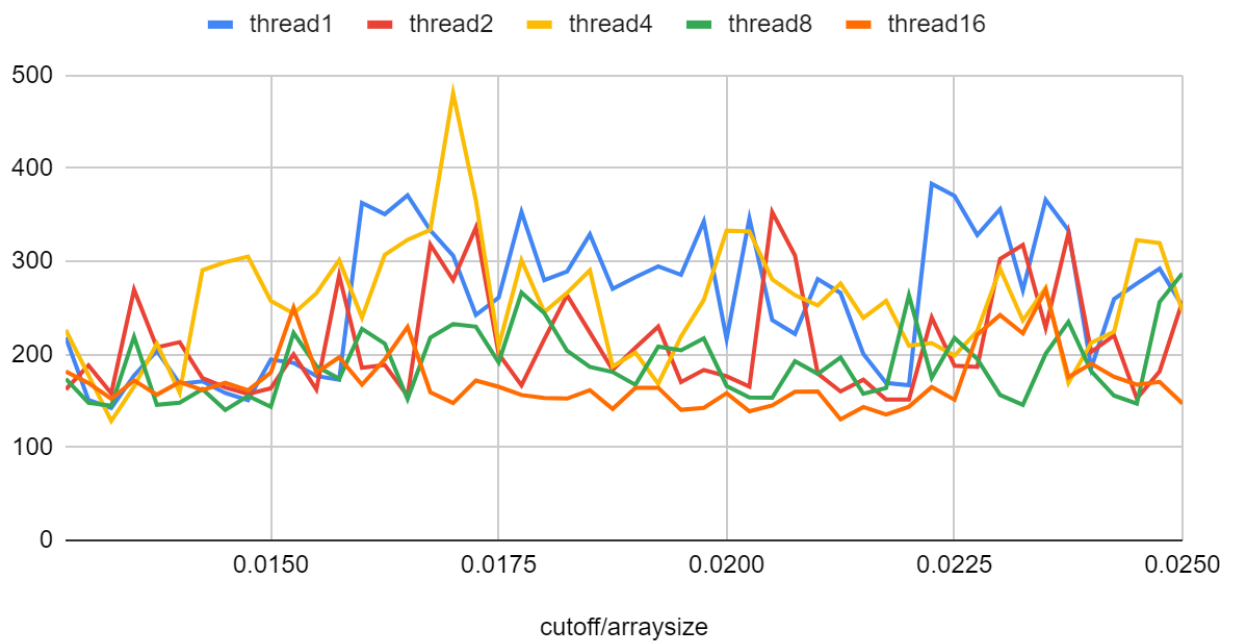
thread1, thread2, thread4, thread8 and thread16 1million



thread1, thread2, thread4, thread8 and thread16 2 Million



thread1, thread2, thread4, thread8 and thread16 4 million



Conclusion:

After comparing all the graphs and the thread performance in each graph, we see that the 16 threaded program is giving a better performance in my mac than a 1 threaded program. 1 threaded program gives me the worst performance.

The 8 threaded program is giving the second best output.

As we keep increasing the number of threads (parallelism) in the program, the time taken to run the program is faster and the efficacy is better.

Hence running parts of the program in parallel increases the efficiency.