

# Program Structures & Algorithms

Spring 2022

## Assignment No. 4

### Parallel Sorting

**Manoj Reddy Amireddy**

**002196218**

**Section - 8**

#### Task

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.

#### Data Output

Exported Console Output into a pdf and added to the code base repo. Also added observations into the codebase.

#### Code Changes

```
IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools Git Window Help
INFO6205 - ParSort.java

Main.java x ParSort.java x
src main java edu neu coe info6205 sort par ParSort parsort Lambda

22 int[] result = new int[xs1.length + xs2.length];
23 // TO IMPLEMENT
24 int i = 0;
25 int j = 0;
26 for (int k = 0; k < result.length; k++) {
27     if (i >= xs1.length) {
28         result[k] = xs2[j++];
29     } else if (j >= xs2.length) {
30         result[k] = xs1[i++];
31     } else if (xs2[j] < xs1[i]) {
32         result[k] = xs2[j++];
33     } else {
34         result[k] = xs1[i++];
35     }
36 }
37 return result;
38 });
39
40 parsort.whenComplete((result, throwable) -> System.arraycopy(result, srcPos: 0, array, from, result.length));
41 // System.out.println("# threads: " + ForkJoinPool.commonPool().getRunningThreadCount());
42 parsort.join();
43 }
44 }
45
46 @ private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool parallelPool) {
47     return CompletableFuture.supplyAsync(
48         () -> {
49             int[] result = new int[to - from];
50             // TO IMPLEMENT
51             System.arraycopy(array, from, result, destPos: 0, result.length);
52             sort(result, from: 0, to: to - from, parallelPool);
53             return result;
54         }, parallelPool);
55 }
56 }
57 }
```

```
IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools Git Window Help
INFO6205 - ParSort.java

Main.java x ParSort.java x
src main java edu neu coe info6205 sort par ParSort parsort Lambda

11 class ParSort {
12
13     public static int cutoff = 1000;
14
15     public static void sort(int[] array, int from, int to, ForkJoinPool parallelPool) {
16         if (to - from < cutoff) Arrays.sort(array, from, to);
17         else {
18             // FIXME next few lines should be removed from public repo.
19             CompletableFuture<int[]> parsort1 = parsort(array, from, to: from + (to - from) / 2, parallelPool); // TO IMPLEMENT
20             CompletableFuture<int[]> parsort2 = parsort(array, from: from + (to - from) / 2, to, parallelPool); // TO IMPLEMENT
21             CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {
22                 int[] result = new int[xs1.length + xs2.length];
23                 // TO IMPLEMENT
24                 int i = 0;
25                 int j = 0;
26                 for (int k = 0; k < result.length; k++) {
27                     if (i >= xs1.length) {
28                         result[k] = xs2[j++];
29                     } else if (j >= xs2.length) {
30                         result[k] = xs1[i++];
31                     } else if (xs2[j] < xs1[i]) {
32                         result[k] = xs2[j++];
33                     } else {
34                         result[k] = xs1[i++];
35                     }
36                 }
37                 return result;
38             });
39
40             parsort.whenComplete((result, throwable) -> System.arraycopy(result, srcPos: 0, array, from, result.length));
41             // System.out.println("# threads: " + ForkJoinPool.commonPool().getRunningThreadCount());
42             parsort.join();
43         }
44     }
45
46     @ private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool parallelPool) {
47         return CompletableFuture.supplyAsync(
48             () -> {
49                 int[] result = new int[to - from];
50                 // TO IMPLEMENT
51                 System.arraycopy(array, from, result, destPos: 0, result.length);
52                 sort(result, from: 0, to: to - from, parallelPool);
53                 return result;
54             }, parallelPool);
55     }
56 }
```

```
INFO6205 - Main.java
src/main/java/edu/neu/coe/info6205/sort/par/Main
Main.java ParSort.java
45 array[k] = random.nextInt( bound: 1000000);
46 }
47 ParSort.sort(array, from: 0, array.length, pool);
48 }
49 long endTime = System.currentTimeMillis();
50 timeTaken = endTime - startTime;
51 timeList.add(timeTaken);
52 System.out.println("Cutoff used ::: " + ParSort.cutoff + " , and time for 10 samples ::: " + timeTaken);
53 }
54 }
55 try{
56     FileOutputStream fileOutputStream = new FileOutputStream( name: "./src/" + "arraySize-" + arraySize + "-thread" + threadsCount + ".csv");
57     OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream);
58     BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);
59     for(int index=0; index<timeList.size(); index++) {
60         StringBuilder stringBuilder = new StringBuilder();
61         stringBuilder.append(cutoff*(index+1));
62         stringBuilder.append(",");
63         stringBuilder.append((double) timeList.get(index)/10);
64         stringBuilder.append("\n");
65         bufferedWriter.write(stringBuilder.toString());
66         bufferedWriter.flush();
67     }
68     bufferedWriter.close();
69 } catch (IOException e) {
70     e.printStackTrace();
71 }
72 }
73 threadsCount *= 2;
74 }
75 threadsCount = 2;
76 arraySize*=2;
77 }
78 }
79 }
80 }
81 }
82 private static void processArgs(String[] args) {
83     String[] xs = args;
84     while (xs.length > 0)
```

```
INFO6205 - Main.java
src/main/java/edu/neu/coe/info6205/sort/par/Main
Main.java ParSort.java
16 //
17 public class Main {
18 }
19 public static void main(String[] args) {
20 }
21 int threadsCount = 2;
22 int arraySize = 50000;
23 int cutoff = 5000;
24 // Changing the threads and sizes for computing avg times for sorting
25 processArgs(args);
26 for(int i=1; i<6; i++) {
27 }
28 for (int tc=1; tc<6; tc++) {
29 }
30 System.out.println("Size of the Array ::: " + arraySize);
31 ForkJoinPool pool = new ForkJoinPool(threadsCount);
32 System.out.println("Current pool of threads ::: " + threadsCount);
33 Random random = new Random();
34 int[] array = new int[arraySize];
35 ArrayList<Long> timeList = new ArrayList<>();
36 }
37 for (int j = 1; j < arraySize/cutoff+1; j++) {
38 }
39 ParSort.cutoff = cutoff * j;
40 long timeTaken;
41 long startTime = System.currentTimeMillis();
42 for (int t = 0; t < 10; t++) {
43     for (int k = 0; k < array.length; k++) {
44         array[k] = random.nextInt( bound: 1000000);
45     }
46     ParSort.sort(array, from: 0, array.length, pool);
47 }
48 long endTime = System.currentTimeMillis();
49 timeTaken = endTime - startTime;
50 timeList.add(timeTaken);
51 System.out.println("Cutoff used ::: " + ParSort.cutoff + " , and time for 10 samples ::: " + timeTaken);
52 }
53 }
54 }
55 try{
56     FileOutputStream fileOutputStream = new FileOutputStream( name: "./src/" + "arraySize-" + arraySize + "-thread" + threadsCount + ".csv");
57     OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream);
58     BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);
59     for(int index=0; index<timeList.size(); index++) {
60         StringBuilder stringBuilder = new StringBuilder();
61         stringBuilder.append(cutoff*(index+1));
62         stringBuilder.append(",");
63         stringBuilder.append((double) timeList.get(index)/10);
64         stringBuilder.append("\n");
65         bufferedWriter.write(stringBuilder.toString());
66         bufferedWriter.flush();
67     }
68     bufferedWriter.close();
69 } catch (IOException e) {
70     e.printStackTrace();
71 }
72 }
73 threadsCount *= 2;
74 }
75 threadsCount = 2;
76 arraySize*=2;
77 }
78 }
79 }
80 }
81 }
82 private static void processArgs(String[] args) {
83     String[] xs = args;
84     while (xs.length > 0)
```

# Observations

After loading the outputs onto sheets and plotting the values of the different cutoffs and threads. I have come to a conclusion that 4 will be the optimal number of threads for processing as there is no change in the performance much as we increase the threads. Also the lowest performance is achieved when the cutoff is exactly  $\frac{1}{4}$  th the size of the array.