Shibo Lu (001548075)

Program Structures & Algorithms

Fall 2021

Assignment No. 5

Task:

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.

Part 1:

Requirement:

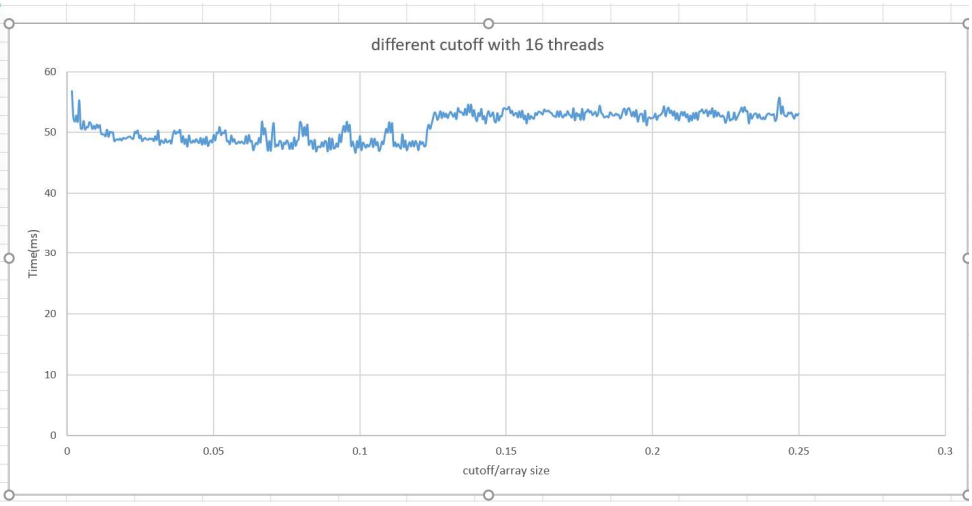
1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.

Using different cutoff to test parSort in main() method:

```
17 public class Main {
18
19     public static int threadCount = 16;
20     public static int arraySize = 2000000;
21     public static int cutoffTime = 1000;
22     public static void main(String[] args) {
23         processArgs(args);
24         System.out.println("Degree of parallelism: " + threadCount);
25         Random random = new Random();
26         int[] array = new int[arraySize];
27         ArrayList<Long> timeList = new ArrayList<>();
28         for (int j = 0; j < 500; j++) {
29             ParSort.cutoff = cutoffTime * (j + 1);
30             // for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);
31             long time;
32             long startTime = System.currentTimeMillis();
33             for (int t = 0; t < 10; t++) {
34                 for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);
35                 ParSort.sort(array, 0, array.length);
36             }
37             long endTime = System.currentTimeMillis();
38             time = (endTime - startTime);
39             timeList.add(time);
40
41
42             System.out.println("cutoff: " + (ParSort.cutoff) + "\t\t10times Time:" + time + "ms");
43
44         }
```

Part 1 Output:

0.0015	56.8
0.002	52.4
0.0025	51.8
0.003	52.8
0.0035	51.8
0.004	55.3
0.0045	50.7
0.005	50.6
0.0055	51.9
0.006	50.5
0.0065	50.9
0.007	50.9
0.0075	51.7
0.008	51.4
0.0085	50.6
0.009	51.1
0.0095	50.6
0.01	51.3
0.0105	50.8
0.011	51.2
0.0115	49.8
0.012	49.7
0.0125	49.7
0.013	49.4
0.0135	50.5
0.014	49.3
0.0145	50.1

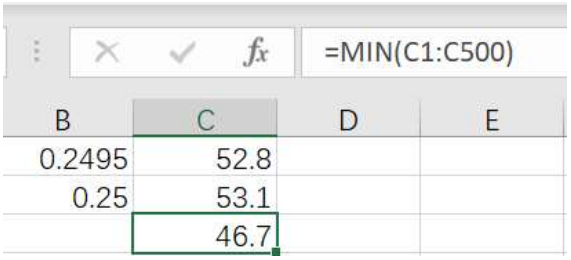



```
Degree of parallelism: 16
cutoff: 1000            10times Time:1357ms
cutoff: 2000            10times Time:1110ms
cutoff: 3000            10times Time:776ms
cutoff: 4000            10times Time:742ms
cutoff: 5000            10times Time:529ms
cutoff: 6000            10times Time:528ms
cutoff: 7000            10times Time:522ms
cutoff: 8000            10times Time:552ms
cutoff: 9000            10times Time:507ms
cutoff: 10000           10times Time:507ms
cutoff: 11000           10times Time:507ms
cutoff: 12000           10times Time:505ms
cutoff: 13000           10times Time:522ms
cutoff: 14000           10times Time:510ms
cutoff: 15000           10times Time:522ms
cutoff: 16000           10times Time:499ms
cutoff: 17000           10times Time:514ms
cutoff: 18000           10times Time:504ms
cutoff: 19000           10times Time:500ms
cutoff: 20000           10times Time:507ms
cutoff: 21000           10times Time:500ms
cutoff: 22000           10times Time:508ms
cutoff: 23000           10times Time:507ms
cutoff: 24000           10times Time:510ms
cutoff: 25000           10times Time:562ms
cutoff: 26000           10times Time:553ms
cutoff: 27000           10times Time:539ms
cutoff: 28000           10times Time:540ms
cutoff: 29000           10times Time:546ms
cutoff: 30000           10times Time:553ms
cutoff: 31000           10times Time:546ms
```

Result:

After using parSort to sort array with different ‘cutoff’, I found that the parSort much faster when the ‘cutoff’ between 9000-248000. When the ‘cutoff’ is 197000 parSort can sort 2000000 numbers in 46.7ms with 16 threads.

```
cutoff: 192000          10times Time:502ms
cutoff: 193000          10times Time:512ms
cutoff: 194000          10times Time:478ms
cutoff: 195000          10times Time:485ms
cutoff: 196000          10times Time:476ms
cutoff: 197000          10times Time:467ms
cutoff: 198000          10times Time:486ms
cutoff: 199000          10times Time:475ms
cutoff: 200000          10times Time:494ms
cutoff: 201000          10times Time:471ms
cutoff: 202000          10times Time:483ms
```



Part 2:

Requirement:

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).

Using different thread number in Parsort() method:

```
class ParSort {

    public static int cutoff = 1000;
    public static ForkJoinPool myPool = new ForkJoinPool(Main.threadCount);
    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, from + (to - from) / 2); // TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, 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, 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) {
        return CompletableFuture.supplyAsync(
                () -> {
                    int[] result = new int[to - from];
                    // TO IMPLEMENT
                    System.arraycopy(array, from, result, 0, result.length);
                    sort(result, 0, to - from);
                    return result;
                }, myPool
        );
    }
```

Output:

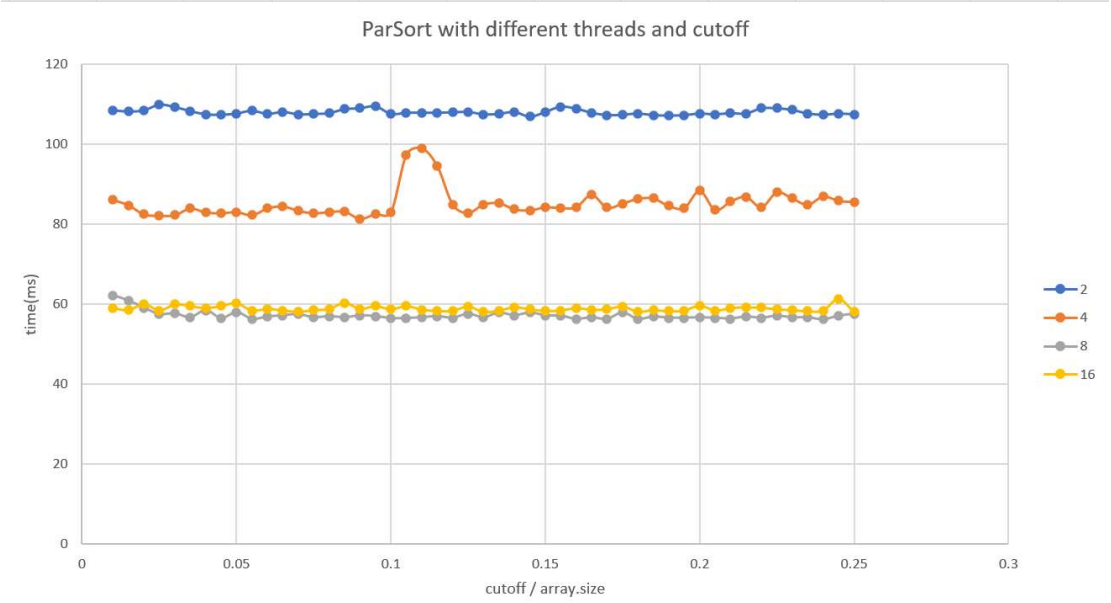
UF_HWQUPC.java | Main.java × | ParSort.java

```
1  package edu.neu.coe.info6205.sort.par;
2
3  import java.io.BufferedWriter;
12
13 /**
14  * This code has been fleshed out by Ziyao Qiao. Thanks very
15  * TODO tidy it up a bit.
16  */
17 public class Main {
18
19     public static int threadCount = 16;
20     public static int arraySize = 2000000;
```

Problems | Javadoc | Declaration | Console ×

<terminated> Main [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe

```
cutoff: 209000          10times Time:585ms
cutoff: 210000          10times Time:572ms
cutoff: 211000          10times Time:590ms
cutoff: 212000          10times Time:570ms
cutoff: 213000          10times Time:576ms
cutoff: 214000          10times Time:569ms
cutoff: 215000          10times Time:580ms
cutoff: 216000          10times Time:567ms
```



Result:

As we can see, the more threads we used the faster Parsort is. Because the CPU in my computer is Intel i7-11800H(8 Cores), 8 and 16 threads will be the most threads I can used.

Git:

https://github.com/ShiboLu/INFO6205-Shibo-Lu/tree/main/INFO6205-Fall2021/src/main/java/edu/neu/coe/info6205/sort/par

Part 3:

3. An appropriate combination of these.

Relationship Conclusion:

When the ratio of cutoff and array sizes are between 0.004-0.124, the Parsort will have highest efficiency. With no more than 16 threads, the more thread we used the higher speed ParSort have. The prefer ratio is 0.08 (most stable value).

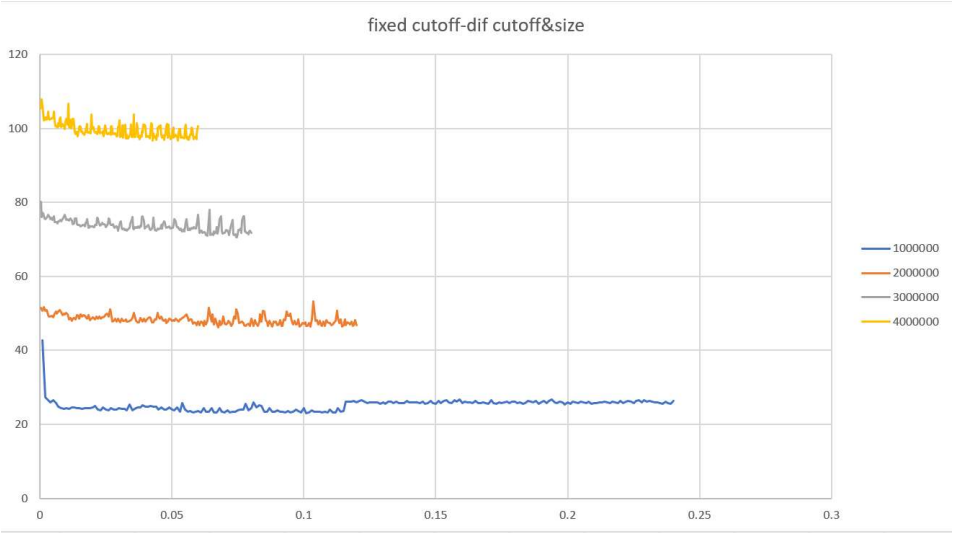
Evidence to support the conclusion:

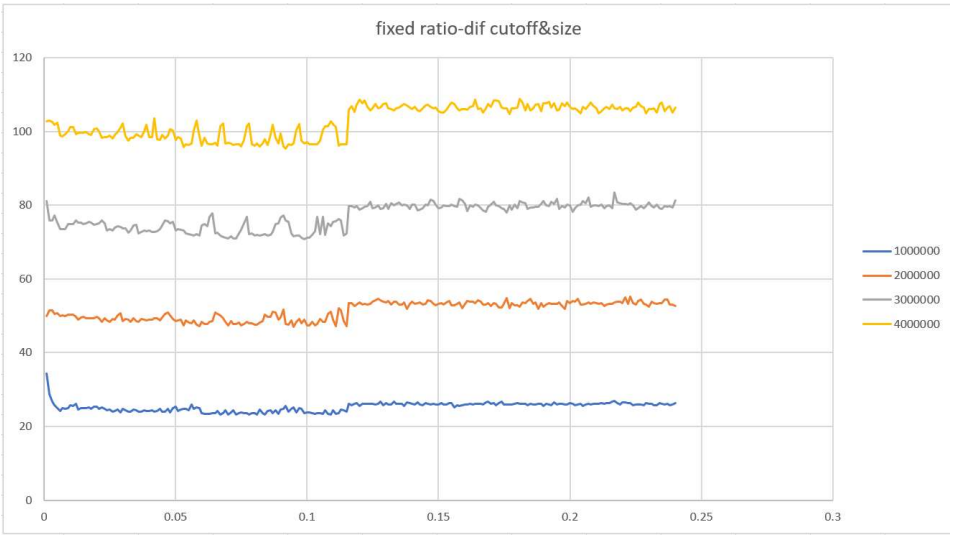
In part 2, I am sure that 16 threads will make Parsort fastest. Then I test Parsort in 16 threads with different cutoff and array sizes. First test is for specific cutoff of with different array sizes. It shows that the running time is relatively stable. In 1 million array size sample, the running time are lower when ratio of cutoff and array sizes are between 0.005-0.12. Second test is for specific proportion of cutoff and array size with different array sizes. In this test, I found that ParSort will more efficient when the ratio of cutoff and array sizes are between 0.004-0.124. It also showed in result in part 1. The ratio of 'cutoff' between 9000-248000 and array size 2000000 almost equals ratio 0.004-0.124.

Analysis：

In parallel sort, we can always get a result: more threads used higher efficiency sort have. But this efficiency will not more than the value when the saved time of data sort in parallel by each thread equals to the wasted time of data sent to each thread. The other side value appears when all the threads in the pool can be used. The ratio between these two values will always make the sort algorithm had high efficiency. This can be a great reason for the area I found above.

Graphical Representation:





Output:

Main.java × ParSort.java

```
18 public class Main {
19
20     public static int threadCount = 16;
21     public static int arraySize = 1000000;
22     public static int cutoffTime = 1000;
23     @SuppressWarnings("static-access")
24     public static void main(String[] args) {
25         processArgs(args);
26         System.out.println("Degree of parallelism: " + ParSo
```

Problems Javadoc Declaration Console ×

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe
Degree of parallelism: 15
Threads:15 size: 1000000  cutoff: 11000  10times Time:394ms
Threads:15 size: 1000000  cutoff: 12000  10times Time:325ms
Threads:15 size: 1000000  cutoff: 13000  10times Time:258ms
Threads:15 size: 1000000  cutoff: 14000  10times Time:253ms
Threads:15 size: 1000000  cutoff: 15000  10times Time:259ms
Threads:15 size: 1000000  cutoff: 16000  10times Time:246ms
Threads:15 size: 1000000  cutoff: 17000  10times Time:244ms
Threads:15 size: 1000000  cutoff: 18000  10times Time:242ms
Threads:15 size: 1000000  cutoff: 19000  10times Time:246ms
Threads:15 size: 1000000  cutoff: 20000  10times Time:244ms
Threads:15 size: 1000000  cutoff: 21000  10times Time:242ms
Threads:15 size: 1000000  cutoff: 22000  10times Time:247ms
```

Git:

https://github.com/ShiboLu/INFO6205-Shibo-Lu/tree/main/INFO6205-Fall2021/src/main/java/edu/neu/coe/info6205/sort/par