Programming Structures & Algorithms
Spring 2022
Assignment no: 4

Name: Shivani Madan Chavan
NUID – 001582611

Tasks:

- Performed changes in the code to accept an object of ForkJoinPool which we have created instead of the static commonPool
- Performed changes in the main method to sort arrays of varying sizes and different cut-off values and number of threads. The output which is avg. time for sorting is printed to console.

Code Snapshots:

main() method:

Notable points:

- A new array is created called arrLengths which represents the size of the array being sorted.
- A new array threadNum is created which represents the number of threads in ForkJoinPool
- A for loop is initialized which loops through arrLengths.
- Array cutoffLen represents the various cutoff values
- Two for loops are implemented in which the outer loop loops through cutoffLen array. The inner loop loops through noOfThreads array.
- A ForkJoinPool corresponding with the number of threads is initiated in the inner loop.
- The array size, its cutoff values, the number of threads and the average time taken is printed to console.

```
public static void main(String[] args) {
    processArgs(args);
    System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());

    //creating an array of length of the array with the lengths being powers of 2
        int[] arrLengths = {524288,1048576,2097152,4194304};

    //creating an array of the NoOfThreads
    int[] threadNum = {2, 4, 8, 16,32};

    ForkJoinPool forkPool;

    Random random = new Random();
    int[] array;
    ArrayList<Long> timeList = new ArrayList<>();
    for (int i = 0; i < arrLengths.length; i++) {
        int length = arrLengths[i];
        array = new int[length];
        System.out.println("Array size " + length);
```

```
        //creating an array of cutoff length
        int[] cutoffLen = {length / 1024 + 1, length / 512 + 1, length / 256 + 1, length / 128 + 1,
                length / 64 + 1, length / 32 + 1, length / 16 + 1, length / 8 + 1, length / 4 + 1,
                length / 2 + 1, length + 1};

        for (int c = 0; c < cutoffLen.length; c++) {
            ParSort.cutoff = cutoffLen[c];
            for (int n = 0; n < threadNum.length; n++) {
                forkPool = new ForkJoinPool(threadNum[n]);
                long duration;
                long startTimems = System.currentTimeMillis();
                for (int t = 0; t < 10; t++) {
                    for (int j = 0; j < length; j++) array[j] = random.nextInt( bound: 10000000);
                    ParSort.sort(array, from: 0, array.length, forkPool);
                }
                long endTimems = System.currentTimeMillis();
                duration = (endTimems - startTimems);

                System.out.println("array size: " + length + " cutoff: " + (ParSort.cutoff) +
                        " noofThreads: " + (threadNum[n]) + "\t\taverage Time:" + (duration / 10) + "ms");
                timeList.add(duration);
            }
        }
    }
```

sort() method:

This method is modified to accept the ForkJoinPool parameter.

```
public static void sort(int[] array, int from, int to, ForkJoinPool forkPool) {
    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,forkPool); // TO IMPLEMENT
        CompletableFuture<int[]> parsort2 = parsort(array, from: from + (to - from) / 2, to,forkPool); // TO IMPLEMENT
        CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {
            int[] result = new int[xs1.length + xs2.length];
```

parsort() method:

This method is modified to accept the ForkJoinPool parameter.

```
private static CompletableFuture<int[]> parsort(int[] array, int from, int to, ForkJoinPool forkPool) {
    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, forkPool);
                return result;
            }, forkPool
    );
}
```

Program Output:

```
Array size 524288
array size: 524288 cutoff: 513 noofThreads: 2        average Time:107ms
array size: 524288 cutoff: 513 noofThreads: 4        average Time:50ms
array size: 524288 cutoff: 513 noofThreads: 8        average Time:56ms
array size: 524288 cutoff: 513 noofThreads: 16       average Time:69ms
array size: 524288 cutoff: 513 noofThreads: 32       average Time:77ms
array size: 524288 cutoff: 1025 noofThreads: 2       average Time:56ms
array size: 524288 cutoff: 1025 noofThreads: 4       average Time:39ms
array size: 524288 cutoff: 1025 noofThreads: 8       average Time:31ms
array size: 524288 cutoff: 1025 noofThreads: 16      average Time:30ms
array size: 524288 cutoff: 1025 noofThreads: 32      average Time:32ms
array size: 524288 cutoff: 2049 noofThreads: 2       average Time:30ms
array size: 524288 cutoff: 2049 noofThreads: 4       average Time:42ms
array size: 524288 cutoff: 2049 noofThreads: 8       average Time:40ms
array size: 524288 cutoff: 2049 noofThreads: 16      average Time:31ms
array size: 524288 cutoff: 2049 noofThreads: 32      average Time:27ms
array size: 524288 cutoff: 4097 noofThreads: 2       average Time:27ms
array size: 524288 cutoff: 4097 noofThreads: 4       average Time:28ms
array size: 524288 cutoff: 4097 noofThreads: 8       average Time:26ms
array size: 524288 cutoff: 4097 noofThreads: 16      average Time:76ms
array size: 524288 cutoff: 4097 noofThreads: 32      average Time:36ms
```

```
Array size 1048576
array size: 1048576 cutoff: 1025 noofThreads: 2      average Time:94ms
array size: 1048576 cutoff: 1025 noofThreads: 4      average Time:68ms
array size: 1048576 cutoff: 1025 noofThreads: 8      average Time:68ms
array size: 1048576 cutoff: 1025 noofThreads: 16     average Time:111ms
array size: 1048576 cutoff: 1025 noofThreads: 32     average Time:83ms
array size: 1048576 cutoff: 2049 noofThreads: 2      average Time:64ms
array size: 1048576 cutoff: 2049 noofThreads: 4      average Time:62ms
array size: 1048576 cutoff: 2049 noofThreads: 8      average Time:59ms
array size: 1048576 cutoff: 2049 noofThreads: 16     average Time:61ms
array size: 1048576 cutoff: 2049 noofThreads: 32     average Time:60ms
array size: 1048576 cutoff: 4097 noofThreads: 2      average Time:55ms
array size: 1048576 cutoff: 4097 noofThreads: 4      average Time:54ms
array size: 1048576 cutoff: 4097 noofThreads: 8      average Time:57ms
array size: 1048576 cutoff: 4097 noofThreads: 16     average Time:55ms
array size: 1048576 cutoff: 4097 noofThreads: 32     average Time:55ms
array size: 1048576 cutoff: 8193 noofThreads: 2      average Time:56ms
array size: 1048576 cutoff: 8193 noofThreads: 4      average Time:53ms
array size: 1048576 cutoff: 8193 noofThreads: 8      average Time:52ms
array size: 1048576 cutoff: 8193 noofThreads: 16     average Time:57ms
```

```
Array size 2097152
array size: 2097152 cutoff: 2049 noofThreads: 2      average Time:171ms
array size: 2097152 cutoff: 2049 noofThreads: 4      average Time:156ms
array size: 2097152 cutoff: 2049 noofThreads: 8      average Time:291ms
array size: 2097152 cutoff: 2049 noofThreads: 16     average Time:181ms
array size: 2097152 cutoff: 2049 noofThreads: 32     average Time:143ms
array size: 2097152 cutoff: 4097 noofThreads: 2      average Time:126ms
array size: 2097152 cutoff: 4097 noofThreads: 4      average Time:120ms
array size: 2097152 cutoff: 4097 noofThreads: 8      average Time:165ms
array size: 2097152 cutoff: 4097 noofThreads: 16     average Time:131ms
array size: 2097152 cutoff: 4097 noofThreads: 32     average Time:395ms
array size: 2097152 cutoff: 8193 noofThreads: 2      average Time:165ms
array size: 2097152 cutoff: 8193 noofThreads: 4      average Time:142ms
array size: 2097152 cutoff: 8193 noofThreads: 8      average Time:122ms
array size: 2097152 cutoff: 8193 noofThreads: 16     average Time:110ms
array size: 2097152 cutoff: 8193 noofThreads: 32     average Time:114ms
array size: 2097152 cutoff: 16385 noofThreads: 2     average Time:114ms
array size: 2097152 cutoff: 16385 noofThreads: 4     average Time:108ms
array size: 2097152 cutoff: 16385 noofThreads: 8     average Time:103ms
array size: 2097152 cutoff: 16385 noofThreads: 16        average Time:110ms
array size: 2097152 cutoff: 16385 noofThreads: 32        average Time:98ms
```

```
Array size 4194304
array size: 4194304 cutoff: 4097 noofThreads: 2      average Time:289ms
array size: 4194304 cutoff: 4097 noofThreads: 4      average Time:289ms
array size: 4194304 cutoff: 4097 noofThreads: 8      average Time:282ms
array size: 4194304 cutoff: 4097 noofThreads: 16     average Time:248ms
array size: 4194304 cutoff: 4097 noofThreads: 32     average Time:290ms
array size: 4194304 cutoff: 8193 noofThreads: 2      average Time:266ms
array size: 4194304 cutoff: 8193 noofThreads: 4      average Time:292ms
array size: 4194304 cutoff: 8193 noofThreads: 8      average Time:262ms
array size: 4194304 cutoff: 8193 noofThreads: 16     average Time:241ms
array size: 4194304 cutoff: 8193 noofThreads: 32     average Time:245ms
array size: 4194304 cutoff: 16385 noofThreads: 2     average Time:230ms
array size: 4194304 cutoff: 16385 noofThreads: 4     average Time:235ms
array size: 4194304 cutoff: 16385 noofThreads: 8     average Time:223ms
array size: 4194304 cutoff: 16385 noofThreads: 16        average Time:241ms
array size: 4194304 cutoff: 16385 noofThreads: 32        average Time:227ms
array size: 4194304 cutoff: 32769 noofThreads: 2     average Time:232ms
array size: 4194304 cutoff: 32769 noofThreads: 4     average Time:209ms
array size: 4194304 cutoff: 32769 noofThreads: 8     average Time:219ms
```

Analysis:

Array Size	Cut-Off Value	Number of Threads	2	4	8	16	32
524288	513		73	67	48	81	72
	1025		38	36	29	29	33
	2049		30	29	29	31	35
	4097		31	36	27	25	25
	8193		32	25	26	26	24
	16385		31	29	29	27	22
	32769		31	28	30	42	30
	65537		52	46	33	26	28
	131073		99	37	28	41	27
	262145		34	64	36	34	31
	524289		40	38	40	40	37

Array Size	Cut-Off Value	Number of Threads	2	4	8	16	32
1048576	1025		100	76	78	77	96
	2049		61	68	64	83	65
	4097		63	57	61	57	57
	8193		58	57	52	55	58
	16385		53	57	55	51	48
	32769		65	58	56	55	44
	65537		60	60	57	49	41
	131073		70	67	52	47	42
	262145		75	60	46	46	47
	524289		62	58	58	59	56
	1048577		77	75	71	74	74

Array Size	Cut-Off Value	Number of Threads	2	4	8	16	32
2097152	2049		376	135	134	136	136
	4097		114	124	122	118	118
	8193		112	108	105	111	107
	16385		108	105	104	106	104
	32769		104	105	105	95	99
	65537		111	113	112	110	84
	131073		121	122	109	92	85
	262145		155	143	109	76	76
	524289		146	114	91	86	94
	1048577		105	101	104	101	102
	2097152		150	150	147	149	148

Array Size	Cut-Off Value	Number of Threads	2	4	8	16	32
4194304	4097		281	260	454	671	386
	8193		276	253	275	281	389
	16385		465	252	242	261	287
	32769		211	230	224	219	232
	65537		222	225	231	238	218
	131073		225	229	215	233	232
	262145		252	275	238	213	195
	524289		302	322	236	159	166
	1048577		320	299	190	185	187
	2097152		288	254	225	252	259
	4194304		367	415	313	338	322

The last row of each table represents time taken by system sort to sort the arrays since the cut-off value is equal to the array size. The number of threads is not making a difference here since we are not performing parallel merge sort. The time taken by system sort is higher than parallel sort.

When the cut-off values are large and near to the array length, the recursion tree's depth is low. In other words, by the time we switch to System sort, there will be relatively few asynchronous jobs to perform.

Conclusion:

- When the cut-off value is high and approximately same as the array size, we can set no. of threads to a high value. In these cases, the arrays are system sorted rather than parallel sort.
- In cases where the cut-off value is equal to or smaller than ~ $(\boldsymbol{length\ of\ array})/\mathbf{2^6} + \mathbf{1}$ **and greater than ~** $(\boldsymbol{length\ of\ array})/\mathbf{2^{10}} + \mathbf{1}$, if we set the value of no. of threads to 4, it is optimal since this is the same as the no. of available processors which are being utilized completely. In such cases, the sub-arrays get parallel sorted.
- Parallel sorting is ideal for larger arrays with full use of available processors when the below condition is true:

$$\frac{\boldsymbol{length\ of\ array}}{\mathbf{2^{10}}} + \mathbf{1} < \boldsymbol{cutoff\ value} \leq (\boldsymbol{length\ of\ array})/\mathbf{2^6} + \mathbf{1}$$

Number of threads: 4