Program Structures & Algorithms
Spring 2022
Assignment No. 4

Name: Xinlin Ying

(NUID): 001535622

- **Task**

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.

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.

Shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cutoff schemes.

- **Output screenshot**

Main.java and ParSort.java

Run the Main.java to get the time for a fixed length array. The cutoff is from 510000 to 1000000 and threads number is 2 to 12.

```java
package edu.neu.coe.info6205.sort.par;

import java.util.Arrays;

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

    static int cutoff = 1000;
    /*
     * Change part I
     */
    static int threadCount = 3;
    public static ForkJoinPool myPool = new ForkJoinPool(threadCount);

    public static void sort(int[] array, int from, int to) {
        if (to - from < cutoff) Arrays.sort(array, from, to);
        else {
            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;
```

Problems Javadoc Declaration Console

<terminated> Main [Java Application] D:\JDK-8\bin\javaw.exe (2022-3-31 20:51:05 – 20:52:51)

```
cutoff: 890000          10times Time:2227ms
cutoff: 900000          10times Time:2179ms
cutoff: 910000          10times Time:2233ms
cutoff: 920000          10times Time:2340ms
cutoff: 930000          10times Time:2135ms
cutoff: 940000          10times Time:1936ms
cutoff: 950000          10times Time:2089ms
cutoff: 960000          10times Time:2019ms
cutoff: 970000          10times Time:2104ms
cutoff: 980000          10times Time:2064ms
cutoff: 990000          10times Time:2172ms
cutoff: 1000000         10times Time:2030ms
```

- **Relationship Conclusion**
    - **Relationship**

Analyzing the graph we can find that:

- sorting time vs array size:

As the array length increases, the sorting time increases in proportion to the array length.

- sorting time vs threads

For each fixed-length array, the sorting time decreases as the number of threads increases, but when the number of threads increases to a certain threshold, the sorting time optimization is not significant. For example, in the case of a 1M array, when the number of threads reaches 8, the sorting time does not decrease significantly, even though the number of threads is increasing.

- sorting time vs cutoff

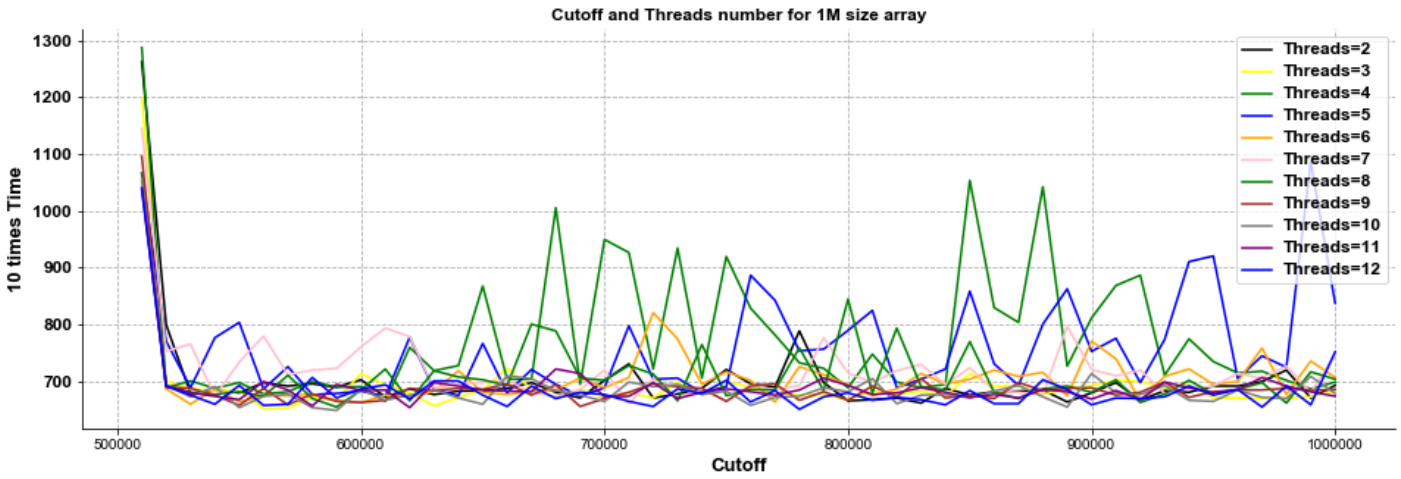
For cutoff, there is a big optimization between 510000 and 610000 but after 510000 there is no big change. It is more obvious when the array length is smaller, for example, for a 1M length array, the sorting time of 510000 cutoff is twice as long as 610000. But for 2M and 4M, it is only about 20% lower.

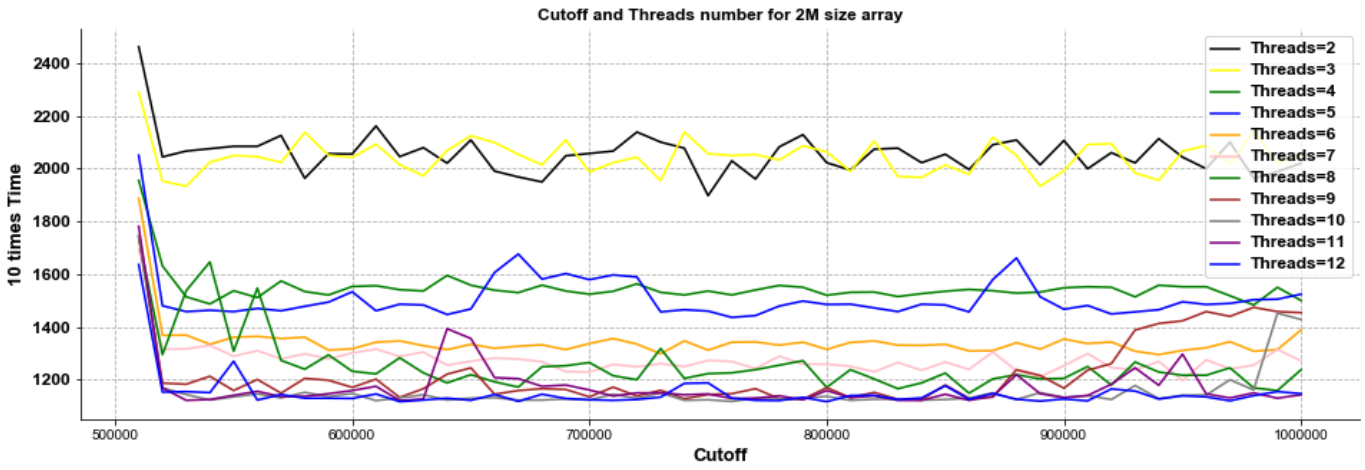
- **Conclusion**

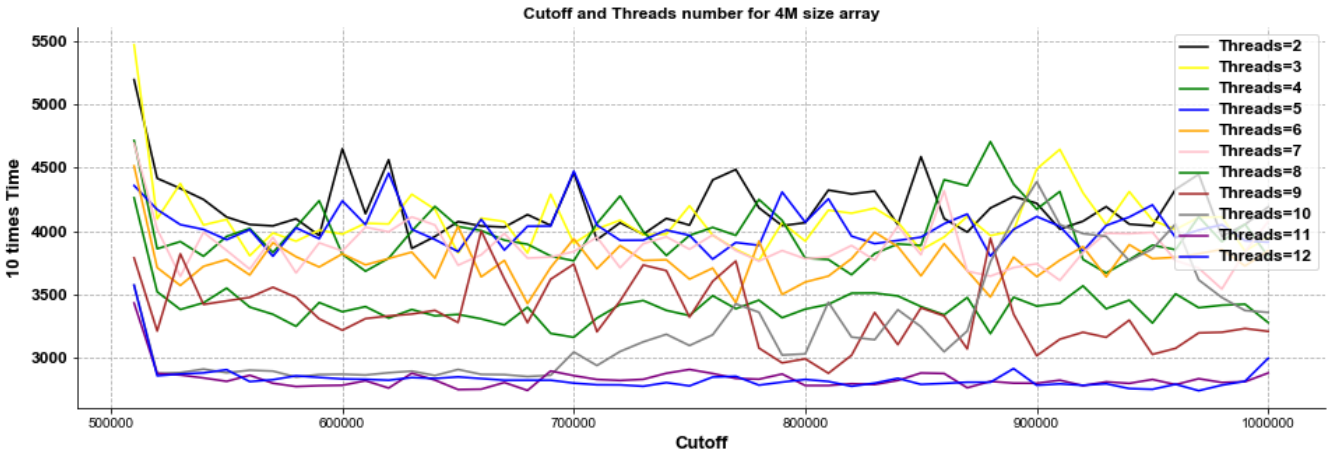
A good strategy of cutoff and number of threads seems to be 510000 length as cutoff and 8 threads. A really large number of cutoff won't make much of a difference because other costs will compensate for the parallel sorting and the number of threads doesn't make difference after a point(generally around 8-10) as we can see from the graph and data sheet.

- **Evidence / Graph**

Line graph between sort time and cutoff







Data Sheet

10 times Time under specific Cutoff and Threads number for 1M size array

Thread / Cuttoff	2	3	4	5	6	7	8	9	10	11	12
510000	**1262**	**1198**	**1287**	**1065**	**1052**	**1144**	**1066**	**1095**	**1057**	**1037**	**1040**
610000	672	693	664	673	675	793	721	666	666	684	693
710000	730	679	926	797	706	660	727	680	698	673	664
810000	667	682	747	824	674	697	679	691	704	676	666
910000	697	701	868	775	739	709	702	675	679	683	670
1000000	692	682	698	837	706	745	745	685	678	673	751

10 times Time under specific Cutoff and Threads number for 2M size array

Thread / Cuttoff	2	3	4	5	6	7	8	9	10	11	12
510000	**2461**	**2288**	**1955**	**2050**	**1887**	**1697**	**1739**	**1743**	**1745**	**1781**	**1636**
610000	2161	2092	1556	1461	1342	1316	1222	1202	1121	1175	1146
710000	2066	2022	1535	1597	1356	1258	1215	1172	1145	1138	1122
810000	1994	1989	1531	1486	1341	1251	1238	1132	1122	1132	1140
910000	1999	2091	1552	1481	1337	1229	1250	1236	1140	1140	1120
1000000	2020	2051	1500	1524	1389	1271	1238	1454	1428	1143	1147

10 times Time under specific Cutoff and Threads number for 4M size array

Thread / Cuttoff	2	3	4	5	6	7	8	9	10	11	12
510000	**5197**	**5473**	**4716**	**4360**	**4515**	**4693**	**4263**	**3788**	**3572**	**3431**	**3572**
610000	4137	4062	3683	4051	3733	4031	3403	3308	2862	2817	2828
710000	3931	4017	4060	4047	3702	3952	3312	3204	2937	2827	2785
810000	4324	4166	3775	4255	3644	3796	3421	2876	3437	2779	2811
910000	4016	4646	4313	4038	3770	3610	3430	3145	4051	2821	2792
1000000	4191	3951	3798	3912	3845	4076	3277	3207	3357	2878	2994