MAYANK YADAV NUID: 02193976

INFO 6205 - Fall 2020

Program Structures & Algorithms

Assignment 4

1. Task:

- Implement a parallel sorting algorithm such that each partition is sorted in parallel.
- Implement a cutoff which will update and find out a good value of this cutoff through experimentation.
- Using determination, decide upon an ideal number of separate threads.

2. Implementation

To produce the desired output, in ParSort.java I completed the implementation of the parallelized merge sort by spawning threads using ForkJoinPool API and passing the thread class in the CompletableFuture's supplyAsync method parallelly partition and merge.

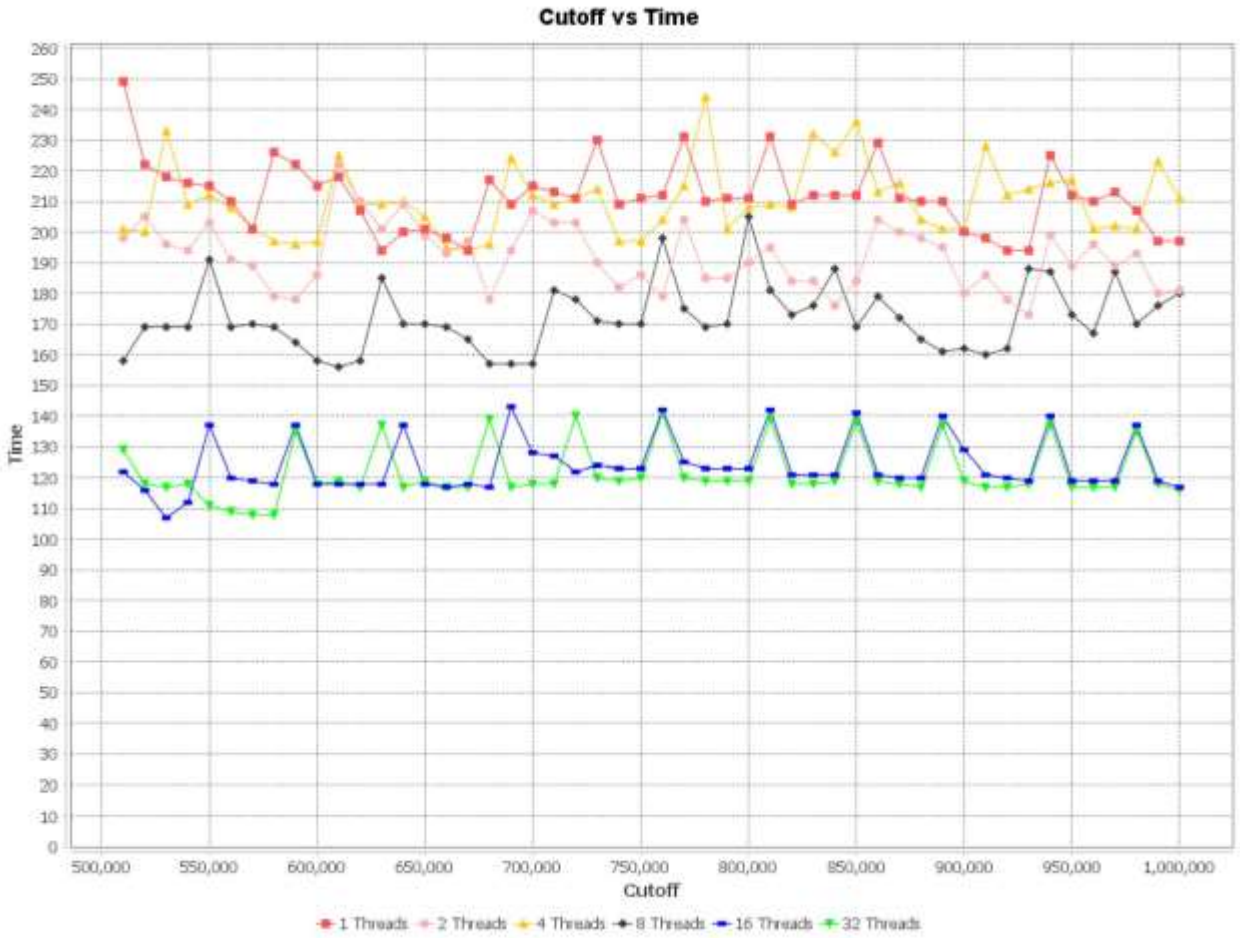
To form a conclusion, I added code in the main.java file which allowed to be plot various comparisons. I formed the following comparisons.

- How different number of threads behave according to different cutoffs.
- How parallelized sorting compares directly to system sort.
- Repeating both above for different number of array elements.

3. Output/Observation

I created a graph with differing cutoffs and different threads for 2 million and 4 million elements to map if a relationship exists or not. I also calculated some basic statistics in each case.

Threads	Average time	Minimum time	Min time cutoff
1	122ms	113ms	680,000
2	116ms	108ms	510,000
4	92ms	86ms	510,000
8	65ms	62ms	830,000
16	65ms	62ms	760,000
32	66ms	62ms	520,000



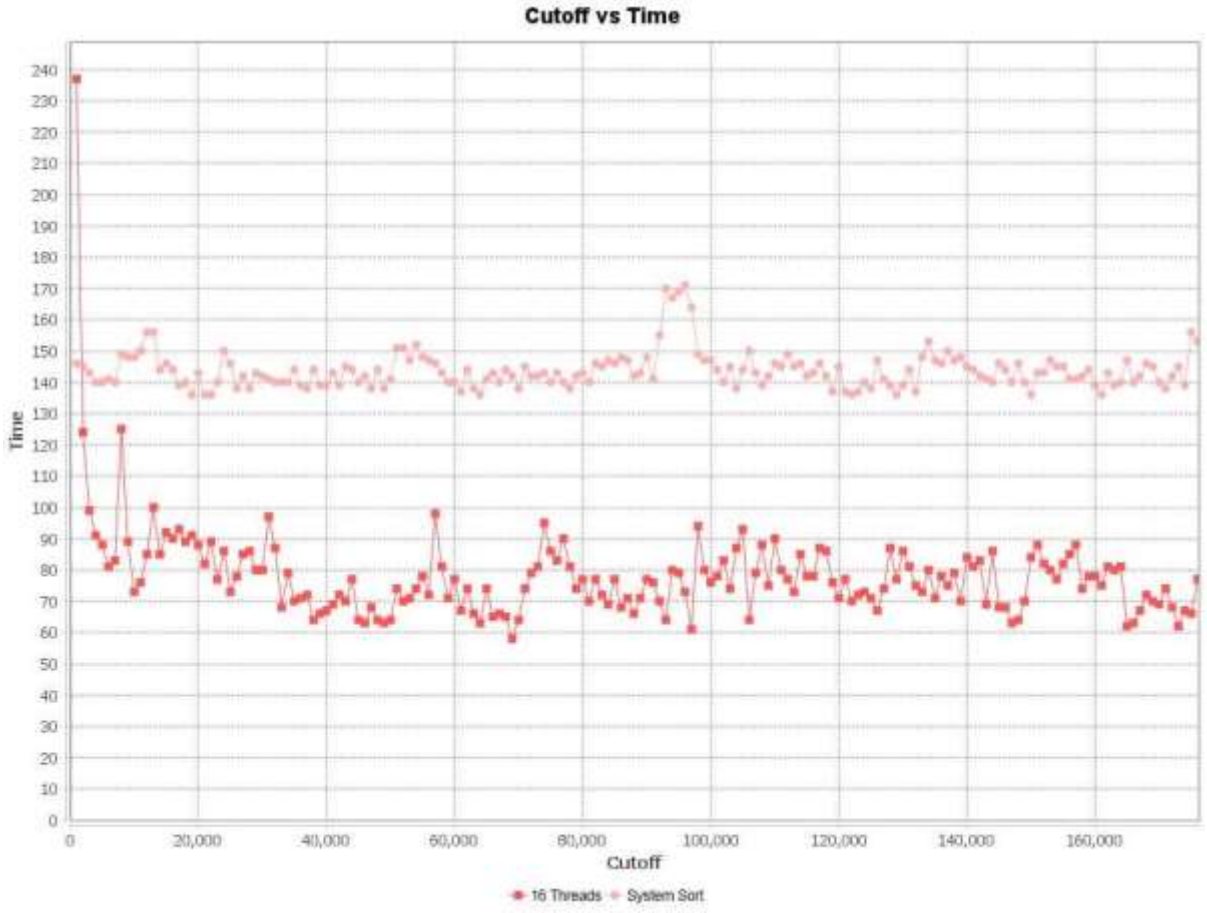
From the above graphs and data on 2 million and 4 million elements, we can notice diminishing returns as we increase the number of threads. As we reach 16 threads and beyond, we can notice how the time taken to sort the array settles in a range.

Example console output

```
Degree of parallelism: 15
cutoff:510000           20 times Time:4364ms
cutoff:520000           20 times Time:4021ms
cutoff:530000           20 times Time:4039ms
cutoff:540000           20 times Time:4012ms
cutoff:550000           20 times Time:4169ms
cutoff:560000           20 times Time:4245ms
cutoff:570000           20 times Time:3893ms
cutoff:580000           20 times Time:4001ms
cutoff:590000           20 times Time:8512ms
cutoff:600000           20 times Time:10605ms
```

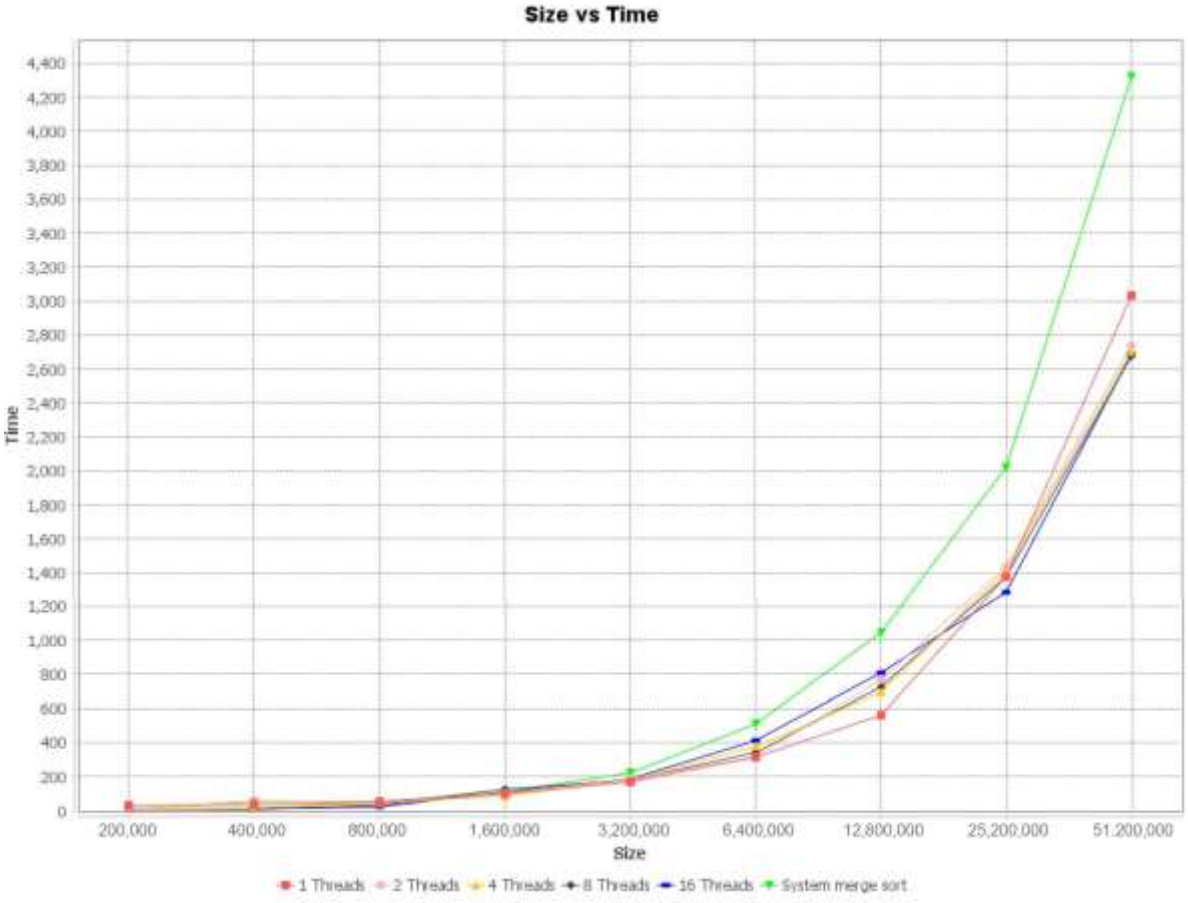
4. Analysis

From the above shown data, we can observe diminishing returns after 16 threads, which is why I settled for 16 threads as an ideal. The question now arises that what cutoff is the ideal cutoff, to find out how the relationship between cutoff and time works, I can the algorithm for 2 million elements for cutoffs between 1000 to 1 million and compared it to the systems built-in merge sort. I got the following graph.



From the above graph we can clearly see, initially with a very low value of cut off the time taken by the 16 threaded merge sort is way higher than the system provided merge sort. But as the cutoff progresses, we start to notice 16 threaded merge sort becomes faster than the system provided. This phenomena can be explained by the fact that when we have a very low cutoff, we end up with a lot of partitions, so most of the time is spent on different threads trying to co-ordinate the data instead of actually sorting the partitions. Hence as the cutoff increases, we get to a point where multithreaded merge sort starts to work faster than the single threaded merge sort. From the graph and data produced above, I can conclude, a cutoff greater than 20,000 is good enough for my 16 threaded merge sort to outperform the single threaded system provided merge sort.

To further prove that multithreaded merge sort can work better than the singlethreaded version at cutoff value that is not too small I plotted time taken by multiple multithreaded mergesorts at different element values against the singlethreaded merge sort.



From the above graph we can clearly see our multithreaded sorts doing better than the system provided merge sort, at the decided cutoff of 160,000 elements. The difference is amplified as we increase the number of elements we are trying to sort. The biggest difference is at 51,200,000 elements, where 16 threaded merge sort performs the best out of the bunch and the system provided merge sort the worst.

5. Conclusion

Note: The experiment and raw figures will be extremely dependent upon system spec and installed ram, for context I have a 16-core system with 32 threads and 32GB of installed RAM.

- After 16 threads, we have diminishing returns and benefits.
- After a certain cutoff value, the multithreaded sorts perform better than the single- threaded merge sort. Which in my case was around 20,000 elements, with the best performance in the range of 100,000 to 300,000 elements while sorting 2 million elements.

- When the cutoff size is too small, single-threaded merge sort is the best because in multithreaded sorting algorithms, the overhead of managing so many partitions becomes too large.

6. Code:

Main.java | ParSort.java | UF.java | UFClient.java | UFClientPlotter.java | plotter.java

```
            ParSort.threadPool = new ForkJoinPool(ParSort.threadCount);
            String seriesName = "" + k + " Threads";
            XYSeries timeSeries = new XYSeries(seriesName);
            double min = 99999;
            int minCutoff = 0;
            double avg = 0;
            for (int j = 50; j < 100; j++) {

                ParSort.cutoff = 10000 * (j + 1);
                // for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);
                long time;
                long startTime = System.currentTimeMillis();
                for (int t = 0; t < 20; t++) {
                    for (int i = 0; i < array.length; i++)
                        array[i] = random.nextInt(10000000);
                    ParSort.sort(array, 0, array.length);
                }
                long endTime = System.currentTimeMillis();
                time = (endTime - startTime);
                avg += time/20;
                if(time/20 < min){
                    minCutoff = 10000 * (j + 1);
                    min = time/20;
                }
                timeList.add(time);
                timeSeries.add(10000 * (j + 1), time/20);
                System.out
                    .println("cutoff:" + (ParSort.cutoff) + "\t\t20 times Time:" + time + "ms");

            }
            System.out.println("For threads " + k + " min is - " + min + " at cutoff " + minCutoff + " and average is - " +
            plot.addSeries(timeSeries);
        }

        plot.initUI();
        plot.setVisible(true);
```

Main.java | ParSort.java | UF.java | UFClient.java | UFClientPlotter.java | plotter.java

```
        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;
                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;
                }, threadPool
        );
    }
}
```