Program Structures & Algorithms

Spring 2022

Assignment No. 4

Parallel Sorting

Manoj Reddy Amireddy
002196218
Section - 8

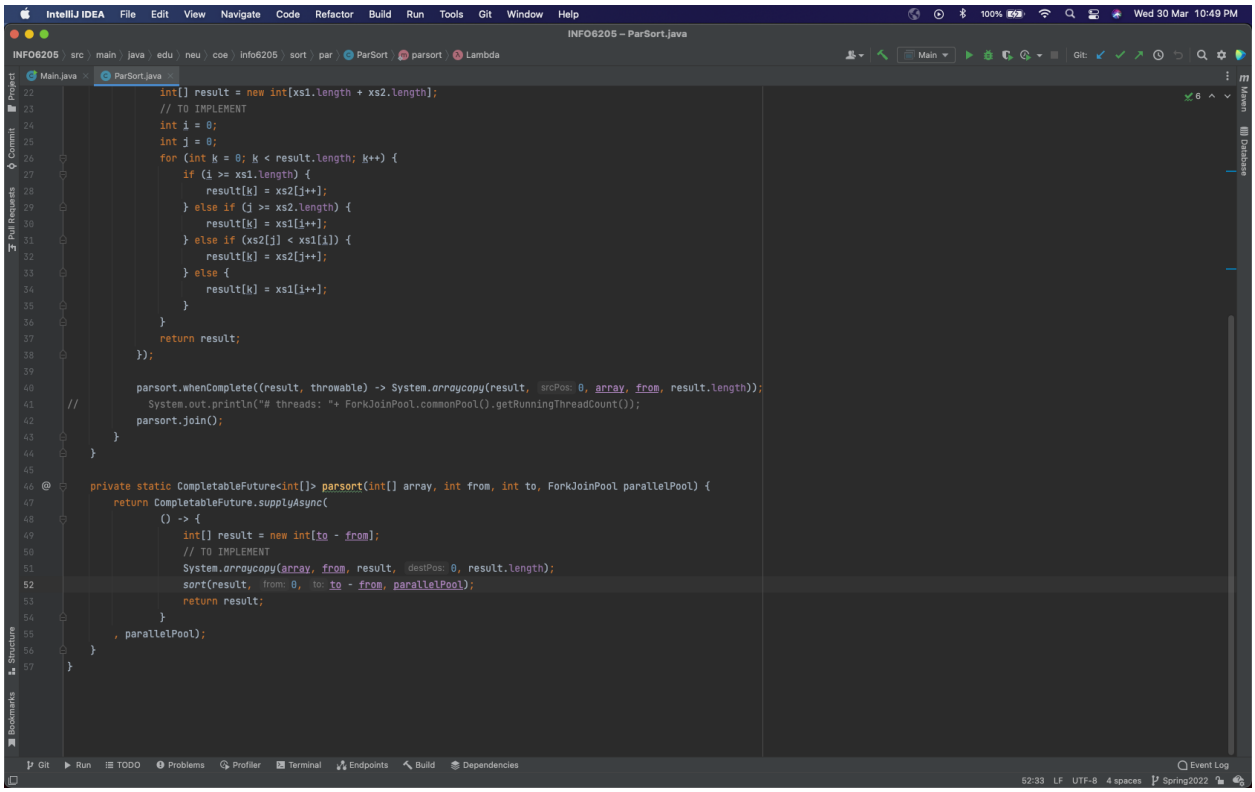
Task

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.

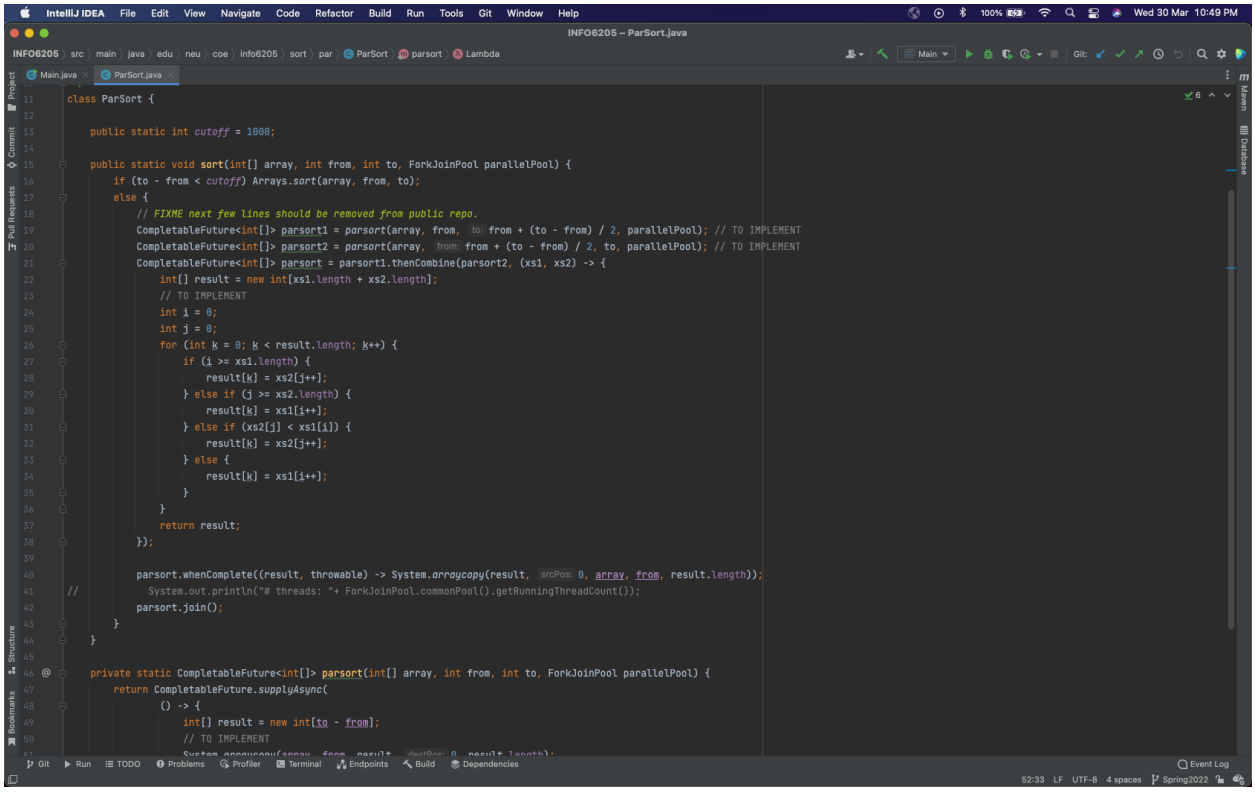
Data Output

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

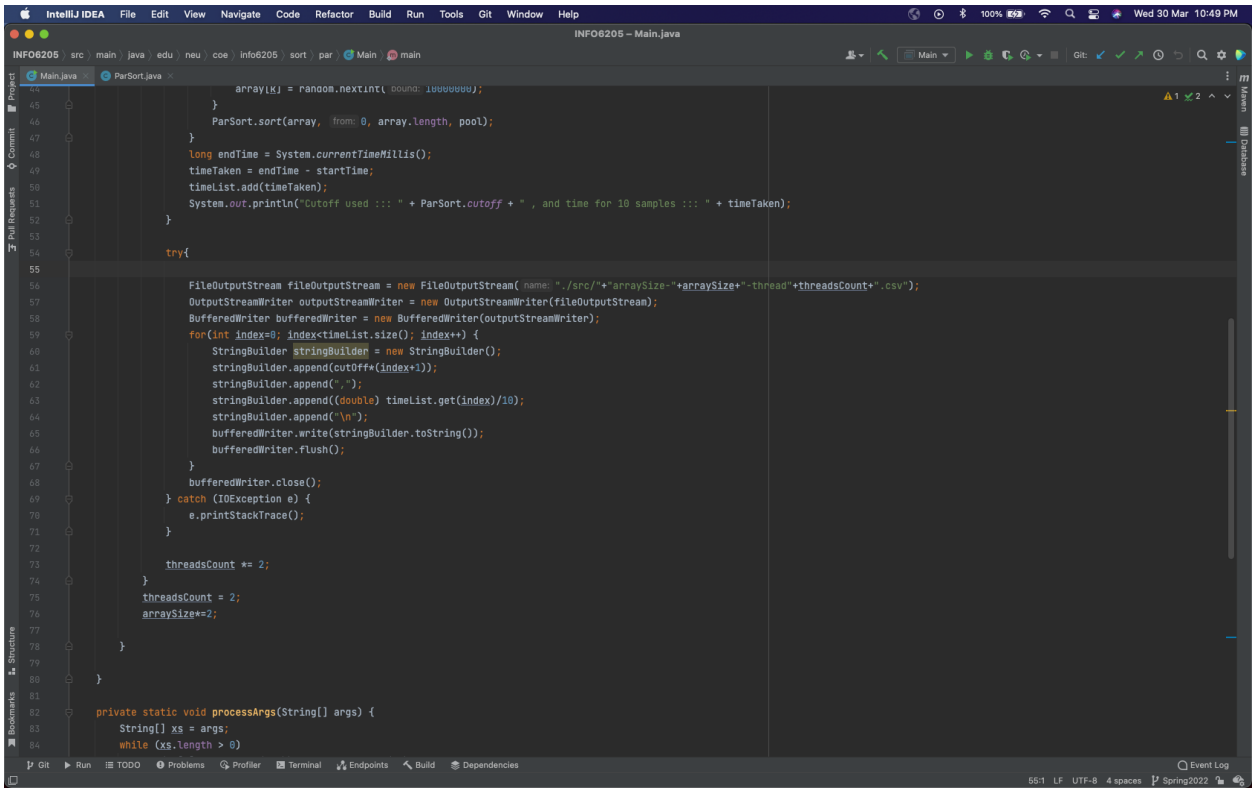
Code Changes

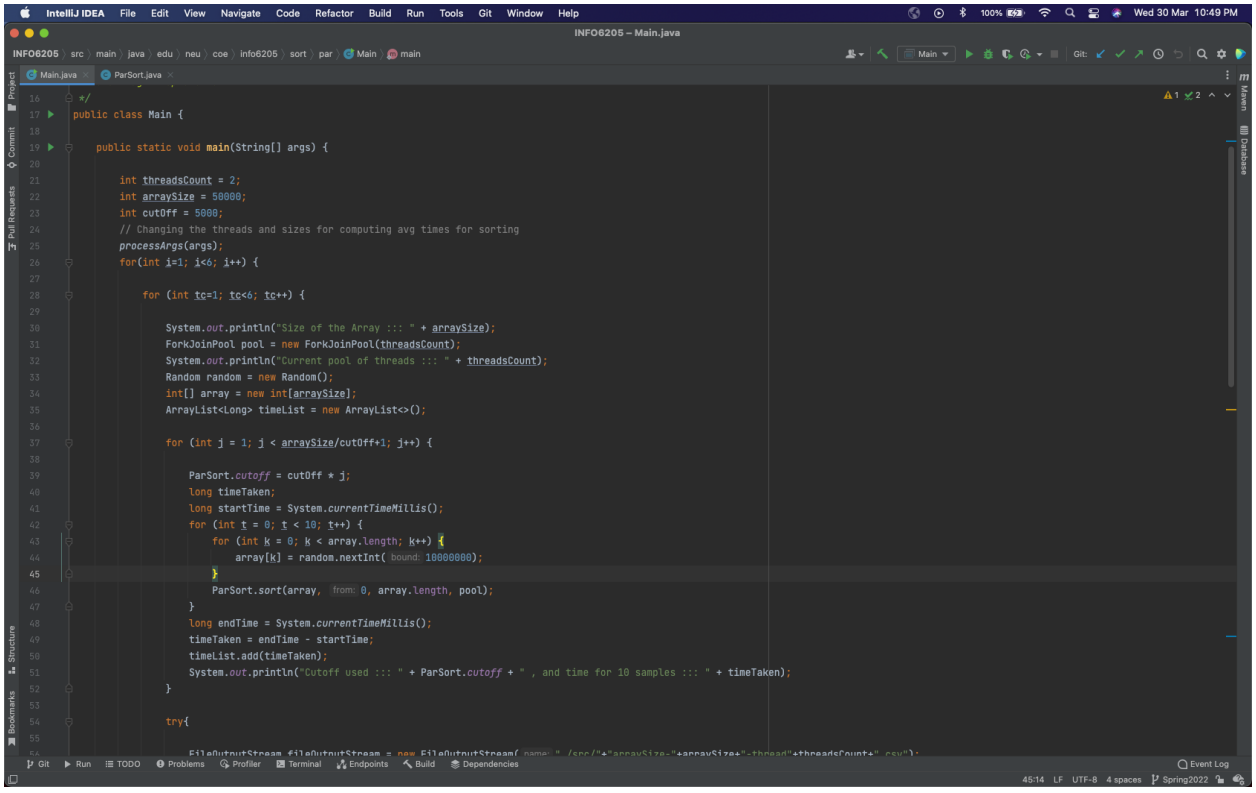


```
                int[] result = new int[xs1.length + xs2.length];
                // TO IMPLEMENT
                int i = 0;
                int j = 0;
                for (int k = 0; k < result.length; k++) {
                    if (i >= xs1.length) {
                        result[k] = xs2[j++];
                    } 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, srcPos: 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, ForkJoinPool parallelPool) {
        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, parallelPool);
                    return result;
                }
        , parallelPool);
    }
}
```



```
class ParSort {

    public static int cutoff = 1000;

    public static void sort(int[] array, int from, int to, ForkJoinPool parallelPool) {
        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, parallelPool); // TO IMPLEMENT
            CompletableFuture<int[]> parsort2 = parsort(array, from: from + (to - from) / 2, to, parallelPool); // 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, srcPos: 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, ForkJoinPool parallelPool) {
        return CompletableFuture.supplyAsync(
                () -> {
                    int[] result = new int[to - from];
                    // TO IMPLEMENT
```





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 ¼ th the size of the array.