Program Structures and Algorithms
Spring 2023(SEC - 8)

NAME: Dinesh Singh Shekhawat
NUID: 002776484

TASK:

1. Implementing a parallel sorting algorithm such that each partition of the array is sorted in parallel. Following schemas must be considered while deciding if the sorting must be done in parallel
    a. A cutoff which will update. If there are fewer elements to sort than the cutoff, then the system sort will be used.
    b. Decide on an ideal number of separate threads (t) in powers of 2 and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of lg t is reached).
    c. An appropriate combination of these
2. Study the efficacy of the implemented Parallelizing Sort method by experimenting with sufficient array sizes and cut off schemes.

RELATIONSHIP CONCLUSION:

Parallel sorting means sorting a big list of items by dividing it into smaller sub-lists that are then sorted in parallel (at the same time). The cutoff value is the point at which we stop using parallel sorting and start using the regular sorting method.

As we increase the cutoff value, the size of the sub-lists also increases. This means that it takes longer to sort each sub-list using the regular sorting method. However, because there are fewer sub-lists to sort in parallel, there are also fewer results to combine. So, the total number of operations required to sort the entire list decreases. This can offset the longer time needed to sort each sub-list, resulting in faster overall sorting time.

However, if the cutoff value is too small, the time it takes to create and manage parallel threads can be more than the time it takes to sort each sub-list in regular order. This can slow down the sorting process. But as the cutoff value increases, the time required to create and manage parallel threads becomes less important compared to the benefits of parallel sorting.

In summary, parallel sorting works best with larger cutoff values. This is because the benefits of parallel sorting outweigh the longer time needed to sort each sub-list, and the time required to manage parallel threads becomes less important.

EVIDENCE TO SUPPORT THAT CONCLUSION

Source Code

Main

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

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.concurrent.ForkJoinPool;

import edu.neu.coe.info6205.sort.par.ParallelSortGraphPlotter.Graph;
import edu.neu.coe.info6205.sort.par.ParallelSortGraphPlotter.Group;
import edu.neu.coe.info6205.sort.par.ParallelSortGraphPlotter.Pair;
import edu.neu.coe.info6205.util.LazyLogger;

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

    private static final LazyLogger logger = new LazyLogger(Main.class);

    private static final String FRAME_TITLE = "Parallel Sort Comparison";
    private static final String X_AXIS_TITLE = "Cut Off";
    private static final String Y_AXIS_TITLE = "Time taken (milliseconds)";

    private static final String TEMPLATE_GRAPH_TITLE = "Array Size = %s";
    private static final String TEMPLATE_GROUP_NAME = "Thread size = %s";

    private static final int CUTOFF_START = 50000;
    private static final int CUTOFF_END = 100000;
    private static final int CUTOFF_INCREMENT = 1000;

    private static final int ITERATIONS = 10;

    private static final int RANDOM_MAX_RANGE = 10000000;
      private static final int MILLION = 1000000;
```

```
	public static void main(String[] args) {
		Random random = new Random();
		List<Integer> arraySizes = Arrays.asList(
				2 * MILLION,
				3 * MILLION,
				4 * MILLION,
				5 * MILLION
		);

		/*
		List<Integer> arraySizes = Arrays.asList(
				200,
				300,
				400,
				500);
		*/

		List<Integer> threadSizes = Arrays.asList(
				(int) Math.pow(2, 2),
				(int) Math.pow(2, 3),
				(int) Math.pow(2, 4),
				(int) Math.pow(2, 5)
		);
		ParallelSortGraphPlotter plotter = new
ParallelSortGraphPlotter(
				FRAME_TITLE,
				X_AXIS_TITLE,
				Y_AXIS_TITLE,
				1200,
				800);

		for (int arraySize : arraySizes) {
			logger.info("########### Computation start for arraySize:
" + arraySize);
			List<Group> groups = new ArrayList<>(threadSizes.size());

			for (int threadSize : threadSizes) {
				logger.info("---------- Start sorting for
threadSize: " + threadSize);
				int[] array = new int[arraySize];

				List<Pair> pairs = new ArrayList<>();
```

```
                        for (int cutoff = CUTOFF_START; cutoff <=
CUTOFF_END; cutoff += CUTOFF_INCREMENT) {
//                              logger.info("++++++++ Start sorting for
cutoff: " + cutoff);

                                ParSort.cutoff = cutoff;
                                ParSort.forkJoinPool = new
ForkJoinPool(threadSize);

                                long startTimestamp =
System.currentTimeMillis();

                                for (int t = 0; t < ITERATIONS; ++t) {
                                      for (int i = 0; i < array.length; i++)
array[i] = random.nextInt(RANDOM_MAX_RANGE);
                              ParSort.sort(array, 0, array.length);
                                }

                                long endTimestamp =
System.currentTimeMillis();
                                long timeTaken = endTimestamp -
startTimestamp;
                                long averageTimeTaken = timeTaken /
ITERATIONS;

                                logger.info(
                                            "[METRIC] thread-count: " +
threadSize
                                            + ", cutoff: " + cutoff
                                            + ", averageTimeTaken: " +
averageTimeTaken);

                                Pair pair = new Pair(cutoff,
averageTimeTaken);
                                pairs.add(pair);
                        }

                        String groupName =
String.format(TEMPLATE_GROUP_NAME, threadSize);
                        Group group = new Group(groupName, pairs);
                        groups.add(group);
```

```
                }

                String graphName = String.format(TEMPLATE_GRAPH_TITLE,
arraySize);
                Graph graph = new Graph(graphName, groups);
                plotter.addGraph(graph);
            }

            plotter.plot();
      }
      /*
    public static void main(String[] args) {
        processArgs(args);
        System.out.println("Degree of parallelism: " +
ForkJoinPool.getCommonPoolParallelism());
        Random random = new Random();
        int[] array = new int[2000000];
        ArrayList<Long> timeList = new ArrayList<>();
        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 < 10; 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);
            timeList.add(time);

            System.out.println("cutoff:" + (ParSort.cutoff) + "\t\t10times
Time:" + time + "ms");

        }
        try {
            FileOutputStream fis = new
FileOutputStream("./src/result.csv");
            OutputStreamWriter isr = new OutputStreamWriter(fis);
            BufferedWriter bw = new BufferedWriter(isr);
```

```
            int j = 0;
            for (long i : timeList) {
                String content = (double) 10000 * (j + 1) / 2000000 + "," +
(double) i / 10 + "\n";
                j++;
                bw.write(content);
                bw.flush();
            }
            bw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private static void processArgs(String[] args) {
        String[] xs = args;
        while (xs.length > 0)
            if (xs[0].startsWith("-")) xs = processArg(xs);
    }
    private static String[] processArg(String[] xs) {
        String[] result = new String[0];
        System.arraycopy(xs, 2, result, 0, xs.length - 2);
        processCommand(xs[0], xs[1]);
        return result;
    }
    private static void processCommand(String x, String y) {
        if (x.equalsIgnoreCase("N")) setConfig(x, Integer.parseInt(y));
        else
            // TODO sort this out
            if (x.equalsIgnoreCase("P")) //noinspection
ResultOfMethodCallIgnored
                ForkJoinPool.getCommonPoolParallelism();
    }
    private static void setConfig(String x, int i) {
        configuration.put(x, i);
    }

    @SuppressWarnings("MismatchedQueryAndUpdateOfCollection")
    private static final Map<String, Integer> configuration = new
HashMap<>();
    */

}
```

ParSort

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

import java.util.Arrays;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ForkJoinPool;

/**
 * This code has been fleshed out by Ziyao Qiao. Thanks very much.
 * CONSIDER tidy it up a bit.
 */
class ParSort {
    public static int cutoff = 1000;
    public static ForkJoinPool forkJoinPool;

    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;
                },
                forkJoinPool
        );
    }
}
```

ParallelSortGraphPlotter

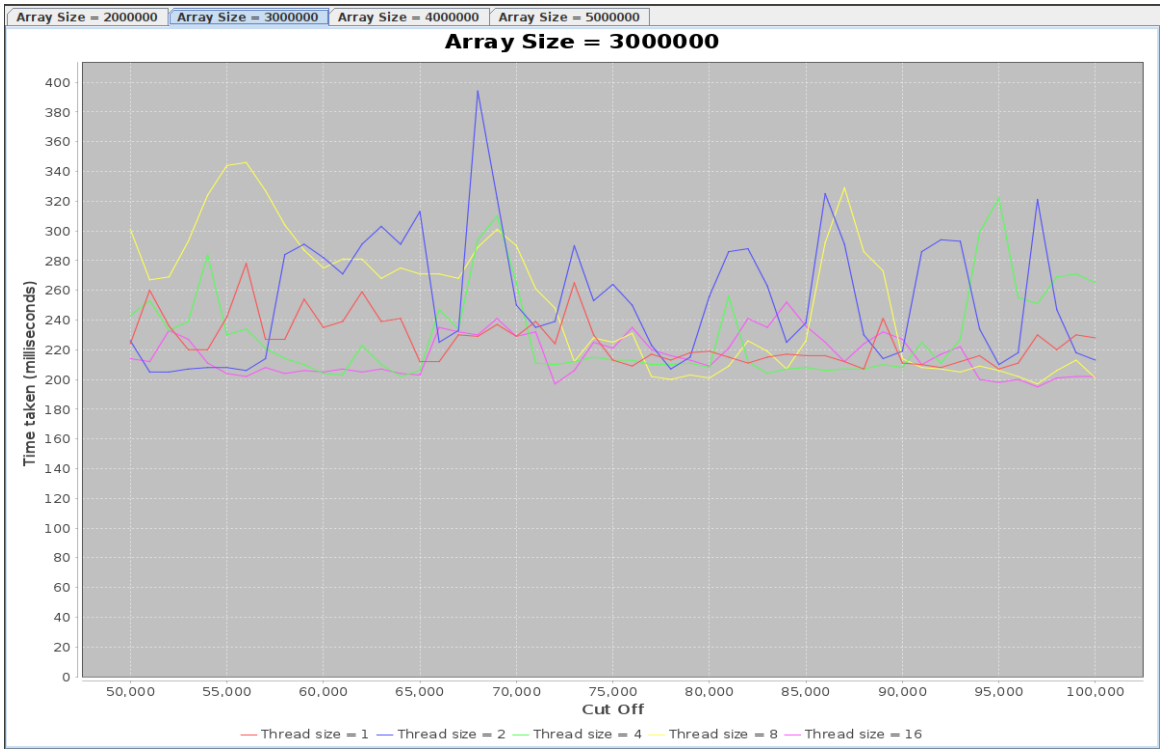
```
package edu.neu.coe.info6205.sort.par;

import java.util.ArrayList;
import java.util.List;

import javax.swing.JFrame;
import javax.swing.JTabbedPane;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

public class ParallelSortGraphPlotter {
	private String frameTitle;
	private String xAxis;
	private String yAxis;
	private int width;
	private int height;

	private List<Graph> graphs;

	public static class Graph {
		private String name;
		private List<Group> groups;

		public Graph(String name, List<Group> groups) {
			this.name = name;
			this.groups = groups;
		}

		public String getName() {
			return name;
		}
		public List<Group> getGroups() {
			return groups;
		}
	}
```

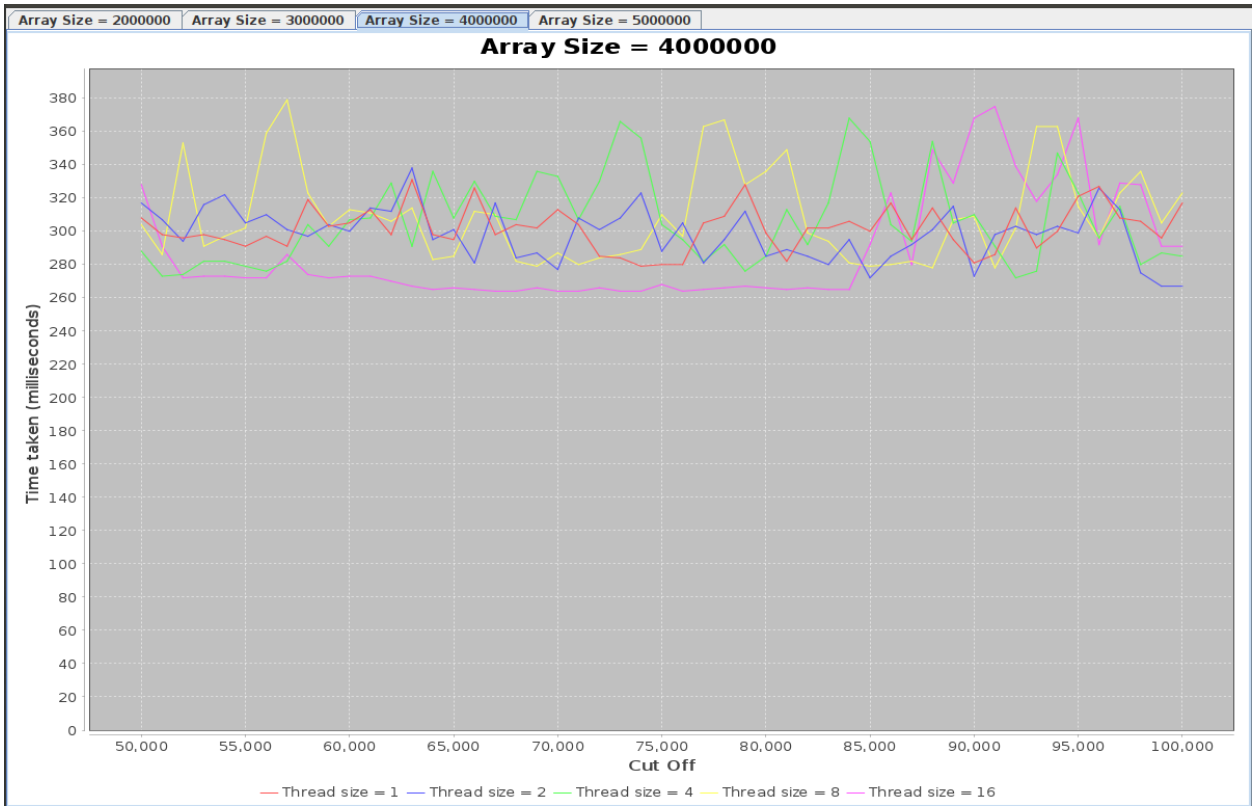
```
	public static class Group {
		private String name;
		private List<Pair> pairs;

		public Group(String name, List<Pair> pairs) {
			this.name = name;
			this.pairs = pairs;
		}

		public String getName() {
			return name;
		}
		public List<Pair> getPairs() {
			return pairs;
		}
	}

	public static class Pair {
		private double x;
		private double y;

		public Pair(double x, double y) {
			this.x = x;
			this.y = y;
		}

		public double getX() {
			return x;
		}
		public double getY() {
			return y;
		}
	}

	public ParallelSortGraphPlotter(
			String frameTitle,
			String xAxis,
			String yAxis,
			int widht,
			int height) {
		this.frameTitle = frameTitle;
		this.xAxis= xAxis;
```

```
            this.yAxis = yAxis;
            this.width = widht;
            this.height = height;

            this.graphs = new ArrayList<>();
      }

      public void addGraph(Graph graph) {
            graphs.add(graph);
      }

      public void plot() {
            JTabbedPane tabbedPane = new JTabbedPane();

            for (Graph graph : graphs) {
                  XYSeriesCollection collection = new XYSeriesCollection();

                  List<Group> groups = graph.getGroups();

                  for (Group group : groups) {
                        String name = group.getName();

                        XYSeries series = new XYSeries(name);

                        List<Pair> pairs = group.getPairs();
                        for (Pair pair : pairs) {
                              double x = pair.getX();
                              double y = pair.getY();

                              series.add(x, y);
                        }

                        collection.addSeries(series);
                  }

                  String graphName = graph.getName();

                  JFreeChart chart = ChartFactory.createXYLineChart(
                              graphName,
                              xAxis,
                              yAxis,
                              collection);
```

```
                ChartPanel chartPanel = new ChartPanel(chart);

                chartPanel.setVisible(true);
                chartPanel.setSize(width, height);

                tabbedPane.addTab(graphName, chartPanel);
            }

            JFrame frame = new JFrame(frameTitle);
            frame.add(tabbedPane);
            frame.setResizable(false);
            frame.setSize(width, height);

            frame.setVisible(true);

frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
      }
}
```

GRAPHICAL REPRESENTATION




Array Size = 2000000
Array Size = 3000000
Array Size = 4000000
Array Size = 5000000
Array Size = 4000000
Time taken (milliseconds)
Cut Off
Thread size = 1
Thread size = 2
Thread size = 4
Thread size = 8
Thread size = 16


Array Size = 2000000
Array Size = 3000000
Array Size = 4000000
Array Size = 5000000
Array Size = 5000000
Time taken (milliseconds)
Cut Off
Thread size = 1
Thread size = 2
Thread size = 4
Thread size = 8
Thread size = 16

UNIT TEST SCREENSHOTS

NOT APPLICABLE