1. BFS=filename-ParallelBFS.java

```
import java.time.Duration;
import java.time.Instant;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class ParallelBFS {
   private static BlockingQueue<TreeNode> queue = new LinkedBlockingQueue<>();

   public static void parallelBFS(TreeNode root) throws InterruptedException, ExecutionException {

      Instant startTime = Instant.now(); // Capture start time
      queue.add(root);

      ExecutorService executor = Executors.newFixedThreadPool(6);

      while (!queue.isEmpty()) {

         int currentLevelSize = queue.size();
         List<Future<?>> tasks = new ArrayList<>();
```

```
        for (int i = 0; i < currentLevelSize; i++) {

            TreeNode node = queue.take();

            tasks.add(executor.submit(() -> {
                // visitNode(node);
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }));
        }
        for (Future<?> task : tasks) {
            task.get();
        }
    }
    executor.shutdown();
    Instant endTime = Instant.now(); // Capture end time

    Duration timeElapsed = Duration.between(startTime, endTime);

    System.out.println("Parallel BFS execution time: " +
            timeElapsed.toMillis() + " ms");
}

public static void main(String[] args) throws ExecutionException {

    int numNodes = 200000000;
    TreeNode root = generateRandomTree(numNodes);
```

```
    try {

        parallelBFS(root);
        SerialBFS.serialBFS(root);

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static class SerialBFS {
    public static void serialBFS(TreeNode root) {
        if (root == null) {
            return;
        }
        Instant startTime = Instant.now(); // Capture start time
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
        Instant endTime = Instant.now(); // Capture end time
        Duration timeElapsed = Duration.between(startTime,
                endTime);
```

```
        System.out.println("Serial BFS execution time: " +
                timeElapsed.toMillis() + " ms");
    }
}

public static TreeNode generateRandomTree(int numNodes) {
    if (numNodes == 0) {
        return null;
    }
    Random random = new Random();
    TreeNode root = new TreeNode(random.nextInt()); // Generate random value for root
    // Recursively generate left and right subtrees with random probabilities
    if (random.nextBoolean()) {
        root.left = generateRandomTree(random.nextInt(numNodes));
        // Random number of nodes in left subtree
    }
    if (random.nextBoolean()) {
        root.right = generateRandomTree(numNodes - (root.left != null ? root.left.size() : 0) - 1); //
Avoid
                                                                    // exceeding
                                                                    // numNodes
    }
    // Calculate total size of the tree
    int size = 1;
    if (root.left != null) {
        size += root.left.size();
    }
    if (root.right != null) {
        size += root.right.size();
    }
    return root;
```

```
    }
}

class TreeNode {

    int value;
    TreeNode left;
    TreeNode right;
    int size;

    public TreeNode(int value) {
        this.value = value;
        this.size = 1;
    }

    public int size() {

        int size = 1; // Size of the current node

        if (left != null) {
            size += left.size();
        }
        if (right != null) {
            size += right.size();
        }
        return size;
    }
}
```

2. DFS-filename= ParallelDFS.java

```
import java.time.Duration;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicInteger;

public class ParallelDFS {
   private static final int THREAD_POOL_SIZE = 6;

   public static void parallelDFS(TreeNode root) throws InterruptedException,
ExecutionException {
      Instant startTime = Instant.now(); // Capture start time

      ExecutorService executor = Executors.newFixedThreadPool(THREAD_POOL_SIZE);
      List<Future<?>> tasks = new ArrayList<>();
      Stack<TreeNode> stack = new Stack<>();

      stack.push(root);

      while (!stack.isEmpty()) {
         TreeNode node = stack.pop();
         tasks.add(executor.submit(() -> {
            // visitNode(node);
            if (node.right != null) {
               stack.push(node.right);
            }
            if (node.left != null) {
               stack.push(node.left);
            }
         }));
      }

      for (Future<?> task : tasks) {
         task.get();
      }

      executor.shutdown();
      Instant endTime = Instant.now(); // Capture end time

      Duration timeElapsed = Duration.between(startTime, endTime);
```

```
        System.out.println("Parallel DFS execution time: " + timeElapsed.toMillis() + " ms");
    }

    public static void main(String[] args) throws ExecutionException {
        int numNodes = 200000000;
        TreeNode root = generateRandomTree(numNodes);

        try {
            parallelDFS(root);
            SerialDFS.serialDFS(root);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static TreeNode generateRandomTree(int numNodes) {
        if (numNodes == 0) {
            return null;
        }
        Random random = new Random();
        TreeNode root = new TreeNode(random.nextInt()); // Generate random value for root

        if (random.nextBoolean()) {
            root.left = generateRandomTree(random.nextInt(numNodes));
        }
        if (random.nextBoolean()) {
            root.right = generateRandomTree(numNodes - (root.left != null ? root.left.size() : 0) -
1);
        }

        // Calculate total size of the tree
        int size = 1;
        if (root.left != null) {
            size += root.left.size();
        }
        if (root.right != null) {
            size += root.right.size();
        }
        return root;
    }
}

class SerialDFS {
    public static void serialDFS(TreeNode root) {
        if (root == null) {
            return;
        }
        Instant startTime = Instant.now(); // Capture start time
```

```
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();
            if (node.right != null) {
                stack.push(node.right);
            }
            if (node.left != null) {
                stack.push(node.left);
            }
        }
        Instant endTime = Instant.now(); // Capture end time
        Duration timeElapsed = Duration.between(startTime, endTime);
        System.out.println("Serial DFS execution time: " + timeElapsed.toMillis() + " ms");
    }
}

class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;
    int size;

    public TreeNode(int value) {
        this.value = value;
        this.size = 1;
    }

    public int size() {
        int size = 1; // Size of the current node
        if (left != null) {
            size += left.size();
        }
        if (right != null) {
            size += right.size();
        }
        return size;
    }
}
```

3. Bubble sort file name- MultiThreadedBubbleSort.java

```
import java.util.*;

class BubbleSort {

    private static final int MAX_THREADS = 4;

    private static class SortThread extends Thread {
        private Integer[] array;
        private int start;
        private int end;

        SortThread(Integer[] array, int start, int end) {
            this.array = array;
            this.start = start;
            this.end = end;
        }

        @Override
        public void run() {
            bubbleSort(array, start, end);
        }
    }

    public static void threadedSort(Integer[] array) {
        long time = System.currentTimeMillis();
        final int length = array.length;
        final int segmentSize = (int) Math.ceil((double) length / MAX_THREADS);
        final List<Thread> threads = new ArrayList<>();
```

```
    for (int i = 0; i < MAX_THREADS; i++) {
        int start = i * segmentSize;
        int end = Math.min((i + 1) * segmentSize, length);
        Thread t = new SortThread(array, start, end);
        threads.add(t);
        t.start();
    }

    for (Thread t : threads) {
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    time = System.currentTimeMillis() - time;
    System.out.println("Time spent for custom multi-threaded bubble_sort(): " + time + "ms");
}

public static void bubbleSort(Integer[] array, int start, int end) {
    for (int i = start; i < end - 1; i++) {
        boolean swapped = false;
        for (int j = start; j < end - 1 - i + start; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                swapped = true;
            }
        }
```

```
            if (!swapped) {
                break;
            }
        }
    }
}
public class MultiThreadedBubbleSort {

    private static Random random = new Random();
    private static final Integer list[] = new Integer[1000];

    static {
        for (int i = 0; i < 1000; i++) {
            list[i] = random.nextInt(1000 + (1000 - 1)) - (1000 - 1);
        }
    }

    public static void main(String[] args) {
        System.out.println("\nInput.length = " + list.length);

        Integer[] arr1 = Arrays.copyOf(list, list.length);
        long startTime = System.currentTimeMillis();
        BubbleSort.bubbleSort(arr1, 0, arr1.length);
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent for custom single threaded bubble_sort(): " + (endTime -
startTime) + "ms");

        Integer[] arr2 = Arrays.copyOf(list, list.length);
        BubbleSort.threadedSort(arr2);
    }
}
```

4. Merge sort file name-MergeSort.java

```
import java.util.*;

class MergeSort {
   private static final int MAX_THREADS = 6;

   private static class SortThreads extends Thread {

      SortThreads(Integer[] array, int begin, int end) {

         super(() -> {
            MergeSort.mergeSort(array, begin, end);
         });
         this.start();
      }
   }

   public static void threadedSort(Integer[] array) {
      long time = System.currentTimeMillis();
      final int length = array.length;
      boolean exact = length % MAX_THREADS == 0;
      int maxlim = exact ? length / MAX_THREADS : length / (MAX_THREADS - 1);

      maxlim = maxlim < MAX_THREADS ? MAX_THREADS : maxlim;
      final ArrayList<SortThreads> threads = new ArrayList<>();
      for (int i = 0; i < length; i += maxlim) {
         int beg = i;
         int remain = (length) - i;
         int end = remain < maxlim ? i + (remain - 1) : i + (maxlim - 1);
```

```
            final SortThreads t = new SortThreads(array, beg, end);
            threads.add(t);
        }
        for (Thread t : threads) {
            try {
                t.join();
            } catch (InterruptedException ignored) {
            }
        }

        for (int i = 0; i < length; i += maxlim) {
            int mid = i == 0 ? 0 : i - 1;
            int remain = (length) - i;
            int end = remain < maxlim ? i + (remain - 1) : i + (maxlim - 1);
            merge(array, 0, mid, end);
        }
        time = System.currentTimeMillis() - time;
        System.out.println("Time spent for custom multi-threaded recursive merge_sort(): " + time +
"ms");
    }

    public static void mergeSort(Integer[] array, int begin, int end) {
        if (begin < end) {
            int mid = (begin + end) / 2;
            mergeSort(array, begin, mid);
            mergeSort(array, mid + 1, end);
            merge(array, begin, mid, end);
        }
    }

    public static void merge(Integer[] array, int begin, int mid, int end) {
```

```
    Integer[] temp = new Integer[(end - begin) + 1];
    int i = begin, j = mid + 1;
    int k = 0;
    while (i <= mid && j <= end) {
        if (array[i] <= array[j]) {
            temp[k] = array[i];
            i += 1;
        } else {
            temp[k] = array[j];
            j += 1;
        }
        k += 1;
    }
    while (i <= mid) {
        temp[k] = array[i];
        i += 1;
        k += 1;
    }
    while (j <= end) {
        temp[k] = array[j];
        j += 1;
        k += 1;
    }
    for (i = begin, k = 0; i <= end; i++, k++) {
        array[i] = temp[k];
    }
}

public static void main(String[] args) {
    // Generate random input array
    Random random = new Random();
```

```
        final int size = random.nextInt(100);
        final Integer list[] = new Integer[100000];
        for (int i = 0; i < 100000; i++) {
            list[i] = random.nextInt(size + (size - 1)) - (size - 1);
        }

        System.out.print("\n" + "Input.length = " + list.length + '\n');
        Integer[] arr1 = Arrays.copyOf(list, list.length);
        long t = System.currentTimeMillis();
        Arrays.sort(arr1, (a, b) -> a > b ? 1 : a == b ? 0 : -1);
        t = System.currentTimeMillis() - t;
        System.out.println("Time spent for system based Arrays.sort(): " + t + "ms");

        Integer[] arr2 = Arrays.copyOf(list, list.length);
        t = System.currentTimeMillis();
        MergeSort.mergeSort(arr2, 0, arr2.length - 1);
        t = System.currentTimeMillis() - t;
        System.out.println("Time spent for custom single threaded recursive merge_sort(): " + t + "ms");

        Integer[] arr = Arrays.copyOf(list, list.length);
        MergeSort.threadedSort(arr);
    }
}
```

5. Parallelreduction=filename-ParallelReduction.java

```
import java.util.stream.LongStream;

public class ParallelReduction {
   public static void main(String[] args) {
      // Generate a smaller range of data to avoid OutOfMemoryError
      long[] data = LongStream.rangeClosed(1, 1000000).toArray();

      long startTime = System.currentTimeMillis();
      long serialMin = serialMin(data);
      long serialMax = serialMax(data);
      long serialSum = serialSum(data);
      double serialAvg = serialAvg(data);
      long endTime = System.currentTimeMillis();
      long serialTime = endTime - startTime;
      System.out.println("Serial Min: " + serialMin + " Time: " + serialTime +
            "ms");
      System.out.println("Serial Max: " + serialMax + " Time: " + serialTime +
            "ms");
      System.out.println("Serial Sum: " + serialSum + " Time: " + serialTime +
            "ms");
      System.out.println("Serial Avg: " + serialAvg + " Time: " + serialTime +
            "ms");

      startTime = System.currentTimeMillis();
      long parallelMin = parallelReduceMin(data);
      long parallelMax = parallelReduceMax(data);
      long parallelSum = parallelReduceSum(data);
      double parallelAvg = parallelReduceAvg(data);
      endTime = System.currentTimeMillis();
      long parallelTime = endTime - startTime;
```

```
    System.out.println("Parallel Min: " + parallelMin + " Time: " +
            parallelTime + "ms");
    System.out.println("Parallel Max: " + parallelMax + " Time: " +
            parallelTime + "ms");
    System.out.println("Parallel Sum: " + parallelSum + " Time: " +
            parallelTime + "ms");
    System.out.println("Parallel Avg: " + parallelAvg + " Time: " +
            parallelTime + "ms");
}

public static long serialMin(long[] data) {
    long min = Long.MAX_VALUE;
    for (long value : data) {
        min = Math.min(min, value);
    }
    return min;
}

public static long serialMax(long[] data) {
    long max = Long.MIN_VALUE;
    for (long value : data) {
        max = Math.max(max, value);
    }
    return max;
}

public static long serialSum(long[] data) {
    long sum = 0;
    for (long value : data) {
        sum += value;
    }
```

```
        return sum;
    }

    public static double serialAvg(long[] data) {
        long sum = serialSum(data);
        return (double) sum / data.length;
    }

    public static long parallelReduceMin(long[] data) {
        return LongStream.of(data).parallel().reduce(Long.MAX_VALUE, Math::min);
    }

    public static long parallelReduceMax(long[] data) {
        return LongStream.of(data).parallel().reduce(Long.MIN_VALUE, Math::max);
    }

    public static long parallelReduceSum(long[] data) {
        return LongStream.of(data).parallel().sum();
    }

    public static double parallelReduceAvg(long[] data) {
        long sum = LongStream.of(data).parallel().sum();
        return (double) sum / data.length;
    }
}
```