HPC Practical_1

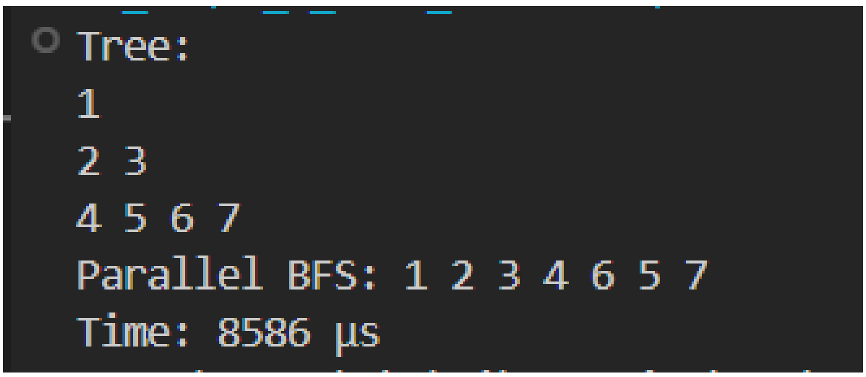
Design and implement Parallel Breadth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS

```
import java.util.*;
import java.util.concurrent.*;

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
}
public class ParallelBFS {
    // Display tree level by level
    static void printTree(TreeNode root) {
        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);
        System.out.println("Tree:");
        while (!q.isEmpty()) {
            int size = q.size();
            while (size-- > 0) {
                TreeNode node = q.poll();
                System.out.print(node.val + " ");
                if (node.left != null) q.add(node.left);
                if (node.right != null) q.add(node.right);
            }
            System.out.println();
        }
    }
    // Parallel BFS
    static void parallelBFS(TreeNode root) throws InterruptedException {
        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);
        ExecutorService ex = Executors.newFixedThreadPool(4);
        System.out.print("Parallel BFS: ");
        while (!q.isEmpty()) {
            int size = q.size();
            List<Future<List<TreeNode>>> tasks = new ArrayList<>();
            for (int i = 0; i < size; i++) {
                TreeNode node = q.poll();
                tasks.add(ex.submit(() -> {
                    System.out.print(node.val + " ");
```

```
                List<TreeNode> kids = new ArrayList<>();
                if (node.left != null) kids.add(node.left);
                if (node.right != null) kids.add(node.right);
                return kids;
            }));
        }
        for (Future<List<TreeNode>> task : tasks)
            try { q.addAll(task.get()); } catch (Exception e) {}
        }
        ex.shutdown();
    }
    public static void main(String[] args) throws InterruptedException {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2); root.right = new TreeNode(3);
        root.left.left = new TreeNode(4); root.left.right = new TreeNode(5);
        root.right.left = new TreeNode(6); root.right.right = new TreeNode(7);
        printTree(root);
        long start = System.nanoTime();
        parallelBFS(root);
        long end = System.nanoTime();
        System.out.println("\nTime: " + (end - start) / 1000 + " µs");
    }
}
```

OUTPUT:



HPC Practical_2

Design and implement Parallel Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for DFS

```
import java.util.*;
import java.util.concurrent.*;

class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
}

public class ParallelDFS {

    // Parallel DFS using ExecutorService
    static void parallelDFS(TreeNode root) throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(4);
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        System.out.print("Parallel DFS: ");
        while (!stack.isEmpty()) {
            List<Future<List<TreeNode>>> futures = new ArrayList<>();
            int size = stack.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = stack.pop();
                futures.add(executor.submit(() -> {
                    System.out.print(node.val + " ");
                    List<TreeNode> children = new ArrayList<>();
                    if (node.right != null) children.add(node.right); // Right first for stack
                    if (node.left != null) children.add(node.left);
                    return children;
                }));
            }
            for (Future<List<TreeNode>> future : futures) {
                try {
                    List<TreeNode> children = future.get();
                    for (TreeNode child : children) {
                        stack.push(child);
                    }
                } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }
    }
    executor.shutdown();
    executor.awaitTermination(1, TimeUnit.MINUTES);
  }

  public static void main(String[] args) throws InterruptedException {
    TreeNode root = new TreeNode(1);
    root.left = new TreeNode(2); root.right = new TreeNode(3);
    root.left.left = new TreeNode(4); root.left.right = new TreeNode(5);
    root.right.left = new TreeNode(6); root.right.right = new TreeNode(7);

    long start = System.nanoTime();
    parallelDFS(root);
    long end = System.nanoTime();

    System.out.println("\nTime: " + (end - start) / 1000 + " µs");
  }
}
```

OUTPUT:

```
PS C:\Users\Bhakti\OneDrive\Desktop\HPC ACTIVITY\LP_V_CODE>  & 'C:\Program Files\Java\jdk-19\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Bhakti\AppData\Roaming\Code\User\workspaceStorage\fdd1b4cadf32e7c3f294c154c730e7f5\redhat.java\jdt_ws\LP_V_CODE_4990fb89\bin' 'ParallelDFS'
Parallel DFS: 1 2 3 7 4 5 6
Time: 22091 µs
```

HPC Practical_3

Write a program to implement Parallel Bubble Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

```
import java.util.*;
import java.util.concurrent.*;

public class ParallelBubbleSort {

    // Sequential Bubble Sort
    static void sequentialBubbleSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
    // Parallel Bubble Sort using ExecutorService
    static void parallelBubbleSort(int[] arr) throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(4);
        for (int i = 0; i < arr.length - 1; i++) {
            List<Future<?>> futures = new ArrayList<>();
            for (int j = 0; j < arr.length - i - 1; j++) {
                final int idx = j;
                futures.add(executor.submit(() -> {
                    if (arr[idx] > arr[idx + 1]) {
                        synchronized (arr) {  // synchronize swap
                            if (arr[idx] > arr[idx + 1]) {
                                int temp = arr[idx];
                                arr[idx] = arr[idx + 1];
                                arr[idx + 1] = temp;
                            }
                        }
                    }
                }));
            }
            for (Future<?> future : futures) {
                try {
```

```
                future.get();  // Wait for all tasks to finish
            } catch (ExecutionException e) {
                e.printStackTrace();  // Handle exception
            }
        }
    }
    executor.shutdown();
}
// Method to print array
static void printArray(int[] arr) {
    for (int num : arr) System.out.print(num + " ");
    System.out.println();
}
public static void main(String[] args) throws InterruptedException {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Enter number of elements: ");
    int n = scanner.nextInt();

    int[] arr = new int[n];
    System.out.println("Enter " + n + " elements:");
    for (int i = 0; i < n; i++) {
        arr[i] = scanner.nextInt();
    }

    int[] arrSeq = arr.clone();
    int[] arrPar = arr.clone();

    // Sequential Bubble Sort
    long startSeq = System.nanoTime();
    sequentialBubbleSort(arrSeq);
    System.out.print("Sequential: ");
    printArray(arrSeq);
    System.out.println("Time: " + (System.nanoTime() - startSeq) / 1000 + " µs");

    // Parallel Bubble Sort
    long startPar = System.nanoTime();
    parallelBubbleSort(arrPar);
    System.out.print("Parallel: ");
    printArray(arrPar);
    System.out.println("Time: " + (System.nanoTime() - startPar) / 1000 + " µs");
    scanner.close();
}
}
```

OUTPUT:

```
Enter number of elements: 4
Enter 4 elements:
11
22
33
44
Sequential: 11 22 33 44
Time: 1115 µs
Parallel: 11 22 33 44
Time: 8087 µs
```

HPC Practical_4

Write a program to implement Parallel Merge Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

```
import java.util.*;
import java.util.concurrent.*;

public class ParallelMergeSort {
    static void mergeSort(int[] arr, boolean parallel) throws InterruptedException {
        if (arr.length < 2) return;
        int mid = arr.length / 2;
        int[] left = Arrays.copyOfRange(arr, 0, mid), right = Arrays.copyOfRange(arr, mid, arr.length);

        if (parallel) {
            ExecutorService ex = Executors.newFixedThreadPool(2);
            Future<?> f1 = ex.submit(() -> { try { mergeSort(left, true); } catch (Exception e) {} });
            Future<?> f2 = ex.submit(() -> { try { mergeSort(right, true); } catch (Exception e) {} });
            try { f1.get(); f2.get(); } catch (Exception e) {}
            ex.shutdown();
        } else {
            mergeSort(left, false);
            mergeSort(right, false);
        }

        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) arr[k++] = (left[i] <= right[j]) ? left[i++] : right[j++];
        while (i < left.length) arr[k++] = left[i++];
        while (j < right.length) arr[k++] = right[j++];
    }

    public static void main(String[] args) throws InterruptedException {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of elements: ");
        int n = sc.nextInt(), arr[] = new int[n];
        System.out.println("Enter " + n + " elements:");
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int[] arrSeq = arr.clone(), arrPar = arr.clone();

        long t1 = System.nanoTime(); mergeSort(arrSeq, false);
        long t2 = System.nanoTime(); mergeSort(arrPar, true);
        long t3 = System.nanoTime();
```

```
    System.out.println("Sequential: " + Arrays.toString(arrSeq) + "\nTime: " + (t2 - t1) / 1000 + " µs");
    System.out.println("Parallel:   " + Arrays.toString(arrPar) + "\nTime: " + (t3 - t2) / 1000 + " µs");

        sc.close();
    }
}
```

OUTPUT:

```
Enter number of elements: 5
Enter 5 elements:
12
13
14
25
24
Sequential: [12, 13, 14, 24, 25]
Time: 64 µs
Parallel:   [12, 13, 14, 24, 25]
Time: 7215 µs
```

HPC Practical_5

Implement Min, Max, Sum and Average operations using Parallel Reduction.

```
import java.util.Scanner;
import java.util.concurrent.*;

public class ParallelReduction {
    static class ReductionTask extends RecursiveTask<int[]> {
        private static final int THRESHOLD = 1000;
        private int[] arr;
        private int start, end;

        public ReductionTask(int[] arr, int start, int end) {
            this.arr = arr;
            this.start = start;
            this.end = end;
        }
        @Override
        protected int[] compute() {
            if (end - start <= THRESHOLD) {
                int min = arr[start], max = arr[start], sum = 0;
                for (int i = start; i < end; i++) {
                    min = Math.min(min, arr[i]);
                    max = Math.max(max, arr[i]);
                    sum += arr[i];
                }
                return new int[]{min, max, sum, end - start};
            } else {
                int mid = (start + end) / 2;
                ReductionTask left = new ReductionTask(arr, start, mid);
                ReductionTask right = new ReductionTask(arr, mid, end);
                left.fork();
                int[] rightResult = right.compute();
                int[] leftResult = left.join();

                int min = Math.min(leftResult[0], rightResult[0]);
                int max = Math.max(leftResult[1], rightResult[1]);
                int sum = leftResult[2] + rightResult[2];
                int count = leftResult[3] + rightResult[3];
                return new int[]{min, max, sum, count};
            }
        }
```

```
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter number of elements: ");
        int n = scanner.nextInt();
        int[] arr = new int[n];

        System.out.println("Enter " + n + " integer values:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        ForkJoinPool pool = new ForkJoinPool();

        long startTime = System.nanoTime();
        int[] result = pool.invoke(new ReductionTask(arr, 0, arr.length));
        long endTime = System.nanoTime();

        int min = result[0], max = result[1], sum = result[2], count = result[3];
        double avg = (double) sum / count;

        System.out.println("\nResults using Parallel Reduction:");
        System.out.println("Minimum: " + min);
        System.out.println("Maximum: " + max);
        System.out.println("Sum     : " + sum);
        System.out.println("Average: " + avg);
        System.out.println("Time taken: " + (endTime - startTime) + " ns");
    }
}
```

OUTPUT:

```
Enter number of elements: 4
Enter 4 integer values:
12
13
14
15

Results using Parallel Reduction:
Minimum: 12
Maximum: 15
Sum    : 54
Average: 13.5
Time taken: 2594700 ns
```