Mini Project - Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyze the performance of each algorithm for the best case and the worst case

```
import java.util.Arrays;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;
public class MergeSort {
  public static void main(String[] args) {
      int[] arr = {12, 11, 13, 5, 6, 7};

      // Single-threaded Merge Sort
      long startTime = System.currentTimeMillis();
      mergeSort(arr);
      long endTime = System.currentTimeMillis();
      System.out.println("Single-threaded Merge Sort took " + (endTime - startTime) + " ms");
      System.out.println("Sorted array: " + Arrays.toString(arr));
      int[] arr2 = {12, 11, 13, 5, 6, 7};

      // Multithreaded Merge Sort
      startTime = System.currentTimeMillis();
      multithreadedMergeSort(arr2);
      endTime = System.currentTimeMillis();
      System.out.println("Multithreaded Merge Sort took " + (endTime - startTime) + " ms");
      System.out.println("Sorted array: " + Arrays.toString(arr2));  }

   public static void mergeSort(int[] arr) {
      if (arr.length > 1) {
         int mid = arr.length / 2;
         int[] left = Arrays.copyOfRange(arr, 0, mid);
         int[] right = Arrays.copyOfRange(arr, mid, arr.length);
         mergeSort(left);
         mergeSort(right);
         int i = 0, j = 0, k = 0;
```

```
        while (i < left.length && j < right.length) {
            if (left[i] < right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }
        while (i < left.length) {
            arr[k++] = left[i++];
        }
        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }
}
public static void multithreadedMergeSort(int[] arr) {
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(new MergeSortTask(arr, 0, arr.length));
}
static class MergeSortTask extends RecursiveTask<Void> {
    private int[] arr;
    private int start, end;
    public MergeSortTask(int[] arr, int start, int end) {
        this.arr = arr;
        this.start = start;
        this.end = end;
    }
    protected Void compute() {
        if (end - start > 1) {
            int mid = (start + end) / 2;
            MergeSortTask leftTask = new MergeSortTask(arr, start, mid);
```

```
            MergeSortTask rightTask = new MergeSortTask(arr, mid, end);
            leftTask.fork();
            rightTask.fork();
            leftTask.join();
            rightTask.join();

            merge(arr, start, mid, end);
        }
        return null;
    }
  }
  private static void merge(int[] arr, int start, int mid, int end) {
    int[] left = Arrays.copyOfRange(arr, start, mid);
    int[] right = Arrays.copyOfRange(arr, mid, end);
    int i = 0, j = 0, k = start;
    while (i < left.length && j < right.length) {
      if (left[i] < right[j]) {
        arr[k++] = left[i++];
      } else {
        arr[k++] = right[j++];
      }
    }
    while (i < left.length) {
      arr[k++] = left[i++];
    }
    while (j < right.length) {
      arr[k++] = right[j++];
    }
  }
}
```

let's compare the performance of single-threaded and multithreaded Merge Sort for best and worst cases:

Performance Analysis:

1. Best Case:

- Best case for Merge Sort: When the array is already sorted, which requires minimal merging.
- Best case for Multithreaded Merge Sort: The best case is the same as the single-threaded version.
- In the best case, both algorithms have a time complexity of O(n log n).

2. Worst Case:

- Worst case for Merge Sort: When the array is in reverse order, resulting in maximum merging.
- Worst case for Multithreaded Merge Sort: The worst case is also when the array is in reverse order, but multithreading may not always provide a significant speedup due to the overhead of thread creation.
- In the worst case, both algorithms still have a time complexity of O(n log n), but the multithreaded version may not always outperform the single-threaded version due to overhead.

To compare the execution time for the best and worst cases, you can use the timeit module in java. Running both algorithms on arrays representing the best and worst cases and measure their execution time.

Conclusion:

In general, if you have a small dataset, the overhead introduced by multithreading might not be worth the potential speedup.

For larger datasets and when you have multiple CPU cores available, multithreaded Merge Sort can provide performance benefits by utilizing parallel processing.

When working in environments where parallelism is not easy to manage (e.g., certain embedded systems or constrained hardware), single-threaded Merge Sort might be a more practical choice.

Be cautious when implementing multithreaded algorithms, as they can introduce complexity and potential synchronization issues, especially in shared-memory environments.

In summary, the choice between Merge Sort and Multithreaded Merge Sort depends on the specific use case, the size of the dataset, and the available hardware resources. Both algorithms have the same time and space complexity, but the multithreaded version may offer performance benefits in certain scenarios.