Parallel Merge Sort with Fork/Join and GUI Visualization

Abstract

This report presents the design and implementation of a modular Java project that implements, visualizes, and evaluates sequential and parallel versions of merge sort.The parallel implementation is based on Java's Fork/Join framework, while Java's built-in Arrays.sort and Arrays.parallelSort are used as baselines.A Swing GUI allows users to experiment with different array sizes, input patterns, and parallel thresholds, while a benchmarking module collects quantitative performance data.Results show that, for the tested problem sizes, all algorithms have similar runtimes, and parallel speedups are limited by task-creation overhead and relatively small input sizes.

Introduction

Sorting is a fundamental operation in computer science with applications in databases, graphics, and scientific computing.Merge sort is a classic divide-and-conquer algorithm with predictable $O(n \log n)$ complexity that lends itself naturally to parallelization.With multi-core processors now standard, understanding how to transform a recursive algorithm such as merge sort into an efficient parallel implementation is an important skill.This project focuses on:

- Implementing a **sequential merge sort** for integer arrays.
- Designing a **parallel merge sort** using the **Fork/Join** framework.
- Comparing both against Java's optimized **Arrays.sort** and **Arrays.parallelSort**.
- Providing a **graphical user interface (GUI)** that makes these algorithms easy to explore.
- Conducting a **performance evaluation** and analyzing when parallelism is beneficial.

Problem Statement

The project requirements can be summarized as follows:

- **Sequential merge sort**
- Implement merge sort for int[].
- Correctly handle:

- Empty arrays.
- Single-element arrays.
- Already sorted arrays.
- **Parallel merge sort using Fork/Join**
- Express merge sort as a recursive task that:
- Splits an array segment into two halves.
- Recursively sorts both halves in parallel.
- Merges the results.
- Use a **threshold**: for small segments, fall back to sequential sort.
- **Benchmarking & comparison**
- Compare:
- Custom sequential merge sort.
- Custom parallel merge sort.
- Arrays.sort.
- Arrays.parallelSort.
- On at least two input patterns:
- Random.
- Reverse-sorted.
- **Object-oriented design**
- Interface SortAlgorithm with a single sort(int[] array) method.
- SequentialMergeSort and ParallelMergeSort implementing this interface.
- SortBenchmark for data generation and performance measurement.
- SortGUI for interactive experiments, optionally as a bonus visualization feature.

Sequential Merge Sort

Conceptual Description

Sequential merge sort follows the divide-and-conquer pattern:

- **Divide**: split the array into two halves until each segment has size one.
- **Conquer**: recursively sort each half (base case: single element).
- **Combine**: merge two sorted halves into a larger sorted segment.

Key properties:

- Time complexity: $O(n \log n)$ for all input orders.
- Space complexity: $O(n)$ additional memory for temporary storage.
- Stability: equal elements preserve their relative order.

The implementation in SequentialMergeSort:

- Checks for null and very small arrays.
- Detects already-sorted arrays to avoid unnecessary work.
- Uses a reusable temporary buffer.
- Implements a standard recursive mergeSort and a two-pointer merge.

Screenshot for sequential merge sort code

```
private boolean isSorted(int[] array) { //to check array sorted or not
    for (int i = 1; i < array.length; i++) {
        if (array[i - 1] > array[i]) {
            return false;
        }
    }
    return true;
}
```

- ```java
@Override
public void sort(int[] array) {

    if (array == null || array.length <= 1) { //edge case
        System.out.println("array has no elements or only one element");
        return;
    }

    if (isSorted(array)) {//edge case
        System.out.println("array is already sorted");
        return;
    }
    int[] temp = new int[array.length];
    mergeSort(array, 0, array.length - 1, temp);
}
```
-

- ```java
private void mergeSort(int[] array, int left, int right, int[] temp) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(array, left, mid, temp);
        mergeSort(array, mid + 1, right, temp);
        merge(array, left, mid, right, temp);
    }
```
-

- ```java
private void merge(int[] array, int left, int mid, int right, int[] temp) {
    System.arraycopy(array, left, temp, left, right - left + 1);

    int i = left;
    int j = mid + 1;
    int k = left;

    while (i <= mid && j <= right) {
        if (temp[i] <= temp[j]) {
            array[k++] = temp[i++];
        } else {
            array[k++] = temp[j++];
        }
    }

    while (i <= mid) {
        array[k++] = temp[i++];
    }

    while (j <= right) {
        array[k++] = temp[j++];
    }
}
```
```

Parallel Merge Sort

Parallelization Strategy

The parallel merge sort keeps the same high-level algorithm but executes independent recursive calls in parallel using Java's ForkJoinPool.Core ideas:

- Represent "sort subarray from left to right" as a **Fork/Join task**.
- For a **large subarray**:
- Compute a midpoint.
- Create two subtasks for the left and right halves.
- Run them in parallel and then merge their results.
- For a **small subarray** (length below the threshold):
- Switch to a **sequential merge sort** on that subrange.
- Use a shared **temporary buffer** and a shared **ForkJoinPool**.

Execution Model and Thread Logic

- The project uses ForkJoinPool.commonPool():
- A pool of worker threads (typically one per core).
- Uses work-stealing to balance tasks across threads.
- Each MergeSortTask:
- Either splits into two child tasks and calls invokeAll, or
- Directly calls a sequential sequentialMergeSort when below the threshold.
- Synchronization:
- Handled implicitly by Fork/Join: when compute returns, both child tasks have completed.
- No explicit locks are needed because each task works on a distinct subrange of the array.

Memory Usage

- A single temp array is allocated at the top level and reused across all tasks for merging.

- Each task only reads and writes within its own [left..right] range.
- Recursion depth is logarithmic in array size, so stack and task overhead remain reasonable.

Screenshot placeholders for parallel merge sort code

- ```java
  private final int threshold;
  private final ForkJoinPool pool;

  /**
   * Creates a ParallelMergeSort with a default threshold.
   */
  public ParallelMergeSort() {
      this(10_000); // reasonable default threshold for int[]
  }

  /**
   * Creates a ParallelMergeSort with a custom threshold.
   *
   * @param threshold minimum segment size to process in parallel
   */
  public ParallelMergeSort(int threshold) {
      if (threshold <= 0) {
          throw new IllegalArgumentException("Threshold must be positive");
      }
      this.threshold = threshold;
      this.pool = ForkJoinPool.commonPool();
  }
  ```
- ```java
  @Override
  public void sort(int[] array) {
      if (array == null || array.length <= 1) {
          return;
      }

      int[] temp = new int[array.length];
      MergeSortTask rootTask = new MergeSortTask(array, temp, 0, array.length - 1, threshold);
      pool.invoke(rootTask);
  ```

- ```java
  private static class MergeSortTask extends RecursiveAction {

      private final int[] array;
      private final int[] temp;
      private final int left;
      private final int right;
      private final int threshold;

      MergeSortTask(int[] array, int[] temp, int left, int right, int threshold) {
          this.array = array;
          this.temp = temp;
          this.left = left;
          this.right = right;
          this.threshold = threshold;
      }
  ```
- ```java
  @Override
  protected void compute() {
      int length = right - left + 1;

      // For small segments, use sequential merge sort to reduce overhead
      if (length <= threshold) {
          sequentialMergeSort(array, temp, left, right);
          return;
      }

      int mid = left + (right - left) / 2;
      MergeSortTask leftTask = new MergeSortTask(array, temp, left, mid, threshold);
      MergeSortTask rightTask = new MergeSortTask(array, temp, mid + 1, right, threshold);

      // Sort halves in parallel
      invokeAll(leftTask, rightTask);

      // Then merge the sorted halves
      merge(array, temp, left, mid, right);
  }
  ```
- ```java
  private static void sequentialMergeSort(int[] array, int[] temp, int left, int right) {
      if (left < right) {
          int mid = left + (right - left) / 2;
          sequentialMergeSort(array, temp, left, mid);
          sequentialMergeSort(array, temp, mid + 1, right);
          merge(array, temp, left, mid, right);
      }
  }
  ```
  ```

- ```java
  /**
   * Merges two sorted subarrays: [left..mid] and [mid+1..right] in-place using a temp buffer.
   */
  private static void merge(int[] array, int[] temp, int left, int mid, int right) {
      System.arraycopy(array, left, temp, left, right - left + 1);

      int i = left;      // pointer in left half
      int j = mid + 1;   // pointer in right half
      int k = left;      // pointer in merged array

      while (i <= mid && j <= right) {
          if (temp[i] <= temp[j]) {
              array[k++] = temp[i++];
          } else {
              array[k++] = temp[j++];
          }
      }

      while (i <= mid) {
          array[k++] = temp[i++];
      }

      while (j <= right) {
          array[k++] = temp[j++];
      }
  }
  ```

Tools & Environment

- **Programming language**: Java (JDK 8 or later).
- **Parallel framework**: java.util.concurrent.ForkJoinPool, RecursiveAction.
- **Standard utility classes**:
- java.util.Arrays for built-in sorts and helpers.
- java.util.Random for random data generation.
- **GUI framework**: Java Swing:
- JFrame, JPanel, GridBagLayout, JTextArea, JScrollPane, JButton, JComboBox, JTextField.
- **Target platform**: Windows 11 (development and testing).
- **Project structure**:
- All .java files declare package algorithms;.
- Compilation with javac -d . *.java creates an algorithms folder with .class files.

  ```

Full Implementation Details

Overall Class Structure

- SortAlgorithm – interface (sorting abstraction).
- SequentialMergeSort – sequential merge sort implementation.
- ParallelMergeSort – parallel merge sort using Fork/Join tasks.
- SortBenchmark – benchmarking and correctness validation.
- SortGUI – interactive user interface.
- Driver – simple development driver for quick manual tests.

SortAlgorithm Interface

Defines the contract for any sorting algorithm used in this project: a single sort(int[] array) method.This allows the benchmark and GUI to treat different algorithms uniformly.

```
package algorithms;
public interface SortAlgorithm {
    void sort(int[] array);
}
```

SortBenchmark – Benchmark and Helpers

Responsibilities:

- Generate input arrays:
- Random arrays via generateRandomArray.
- Reverse-sorted arrays via generateReverseSortedArray (sort + reverse).
- Validate correctness via isSorted.
- Run each algorithm on identical input copies and compute average runtime.

Key logic:

- In main, arrays of algorithms and algorithm names are built:
- SequentialMergeSort.
- ParallelMergeSort.
- ArraysSortAlgorithm (wrapper around Arrays.sort).

- ArraysParallelSortAlgorithm (wrapper around Arrays.parallelSort).
- For each array size and pattern:
- A base array is generated once.
- For each algorithm:
- A clone of the base array is passed to benchmarkAlgorithm.
- Average time (in milliseconds) is printed with one decimal place.

Screenshot placeholders:

- ```
  private static final int[] SIZES = {10_000};
  private static final int RUNS_PER_CASE = 5;
  private static final Random RANDOM = new Random();

  public static void main(String[] args) {
      SortAlgorithm seq = new SequentialMergeSort();
      SortAlgorithm par = new ParallelMergeSort(10_000);
      SortAlgorithm arraysSort = new ArraysSortAlgorithm();
      SortAlgorithm arraysParallelSort = new ArraysParallelSortAlgorithm();

      SortAlgorithm[] algorithms = {seq, par, arraysSort, arraysParallelSort};
      String[] algorithmNames = {"SequentialMergeSort", "ParallelMergeSort", "Arrays.sort", "Arrays.parallelSort"};

      String[] patterns = {"Random", "Reverse"};

      System.out.println("=== Sort Benchmark ===");
      System.out.println("Runs per case: " + RUNS_PER_CASE);
      System.out.println();
  ```
- ```
  for (int size : SIZES) {
      for (String pattern : patterns) {
          int[] baseArray;
          if ("Random".equals(pattern)) {
              baseArray = generateRandomArray(size);
          } else {
              baseArray = generateReverseSortedArray(size);
          }

          System.out.println("Size = " + size + ", Pattern = " + pattern);
          for (int i = 0; i < algorithms.length; i++) {
              long avgNanos = benchmarkAlgorithm(algorithms[i], baseArray, RUNS_PER_CASE);
              double avgMillis = avgNanos / 1_000_000.0;
              System.out.printf("%-20s : %.1f ms%n", algorithmNames[i], avgMillis);
          }
          System.out.println();
      }
  }
  ```

- ```java
  public static long benchmarkAlgorithm(SortAlgorithm algorithm, int[] original, int runs) {
      long totalNanos = 0L;
      for (int r = 0; r < runs; r++) {
          int[] copy = Arrays.copyOf(original, original.length);
          long start = System.nanoTime();
          algorithm.sort(copy);
          long end = System.nanoTime();

          if (!isSorted(copy)) {
              throw new IllegalStateException("Array is not sorted correctly by " + algorithm.getClass().getSimpleName());
          }

          totalNanos += (end - start);
      }
      return totalNanos / runs;
  ```
- ```java
  public static int[] generateRandomArray(int size) {
      int[] array = new int[size];
      for (int i = 0; i < size; i++) {
          array[i] = RANDOM.nextInt();
      }
      return array;
  }

  /**
   * Generates a reverse-sorted int array of the given size.
   */
  public static int[] generateReverseSortedArray(int size) {
      int[] array = generateRandomArray(size);
      Arrays.sort(array);
      // Reverse to get descending order
      for (int i = 0; i < array.length / 2; i++) {
          int tmp = array[i];
          array[i] = array[array.length - 1 - i];
          array[array.length - 1 - i] = tmp;
      }
      return array;
  ```
- ```java
  public static boolean isSorted(int[] array) {
      if (array == null || array.length <= 1) {
          return true;
      }
      for (int i = 1; i < array.length; i++) {
          if (array[i - 1] > array[i]) {
              return false;
          }
      }
      return true;
  ```
  ```

- ```java
  /**
   * Wrapper around Arrays.sort implementing SortAlgorithm, for comparison.
   */
  static class ArraysSortAlgorithm implements SortAlgorithm {
      @Override
      public void sort(int[] array) {
          Arrays.sort(array);
      }
  }

  /**
   * Wrapper around Arrays.parallelSort implementing SortAlgorithm, for comparison.
   */
  static class ArraysParallelSortAlgorithm implements SortAlgorithm {
      @Override
      public void sort(int[] array) {
          Arrays.parallelSort(array);
      }
  }
  ```

Driver – Simple Test Harness

A compact class with a main method that:

- Creates a small integer array.
- Prints it before sorting.
- Calls SequentialMergeSort.sort.
- Prints it after sorting.

Used during development as a sanity check.

- ```java
  package algorithms;
  import java.util.Arrays;
  public class Driver {
      public static void main (String[] args) {
              int[] testArray = {1,1};
      System.out.println("Before sorting: " + Arrays.toString(testArray));
      new SequentialMergeSort().sort(testArray);
      System.out.println("After sorting: " + Arrays.toString(testArray));
  }
      }
  ```

GUI Module

Framework and Layout

The GUI is implemented in SortGUI using Swing:

- A JFrame titled "Parallel Merge Sort Demo".
- A top JPanel with GridBagLayout for controls:
- Algorithm selector.
- Array size field.
- Pattern selector.
- Parallel threshold field.
- Run button.
- A central JTextArea wrapped in a JScrollPane for logs.

Controls and User Inputs

- **Algorithm dropdown (JComboBox<String>)**
- Options:
- Sequential Merge Sort
- Parallel Merge Sort
- Arrays.sort
- Arrays.parallelSort
- **Array size (JTextField)**
- Default value: 100000.
- Must be a positive integer; otherwise, an error dialog is shown.
- **Pattern dropdown (JComboBox<String>)**
- Options:
- Random
- Reverse
- **Parallel threshold (JTextField)**
- Default: 10000.
- Only used when Parallel Merge Sort is chosen.

- Must be a positive integer; otherwise, an error dialog is shown.
- **Run button (JButton)**
- When clicked, initiates one experiment run and logs the results.

Data Flow Between GUI and Backend

On clicking **Run Sort**:

1. Reads algorithm name, pattern, array size, and threshold from the controls.
2. Validates numeric inputs and shows an error dialog if invalid.
3. Uses SortBenchmark.generateRandomArray or generateReverseSortedArray to create input data.
4. Clones the input array for sorting, keeping a copy for “Before” preview.
5. Chooses the appropriate SortAlgorithm implementation based on the selected algorithm.
6. Measures execution time in nanoseconds, converts to milliseconds (double with one decimal).
7. Verifies that the output is sorted using SortBenchmark.isSorted.
8. Appends a log entry to the text area including:

- Algorithm.
- Pattern.
- Size.
- Time in ms.
- Sorted flag.
- First 20 elements before and after sorting.

GUI Screenshot Placeholders

Code screenshots:

- ```java
  private static void createAndShowGUI() {
      JFrame frame = new JFrame("Parallel Merge Sort Demo");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(800, 600);
      frame.setLocationRelativeTo(null);

      JPanel controlPanel = new JPanel(new GridBagLayout());
  ```
  ```

```java
JPanel controlPanel = new JPanel(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();
gbc.insets = new Insets(5, 5, 5, 5);
gbc.anchor = GridBagConstraints.WEST;
```

- ```java
  int row = 0;

  gbc.gridx = 0;
  gbc.gridy = row;
  controlPanel.add(algorithmLabel, gbc);
  gbc.gridx = 1;
  controlPanel.add(algorithmCombo, gbc);

  row++;
  gbc.gridx = 0;
  gbc.gridy = row;
  controlPanel.add(sizeLabel, gbc);
  gbc.gridx = 1;
  controlPanel.add(sizeField, gbc);

  row++;
  gbc.gridx = 0;
  gbc.gridy = row;
  controlPanel.add(patternLabel, gbc);
  gbc.gridx = 1;
  controlPanel.add(patternCombo, gbc);

  row++;
  gbc.gridx = 0;
  gbc.gridy = row;
  controlPanel.add(thresholdLabel, gbc);
  gbc.gridx = 1;
  controlPanel.add(thresholdField, gbc);

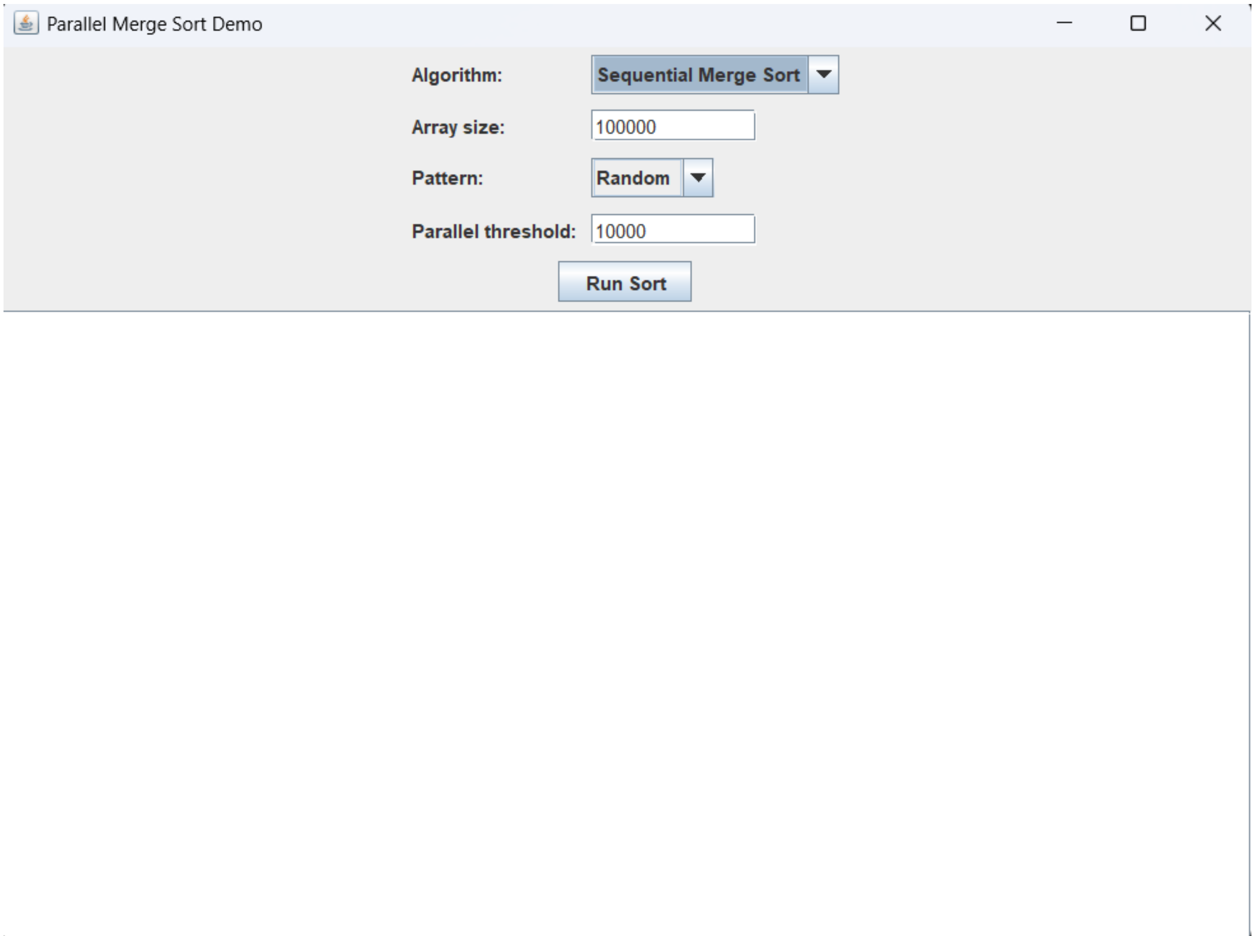
  row++;
  gbc.gridx = 0;
  gbc.gridy = row;
  gbc.gridwidth = 2;
  gbc.anchor = GridBagConstraints.CENTER;
  controlPanel.add(runButton, gbc);
  ```
  ```

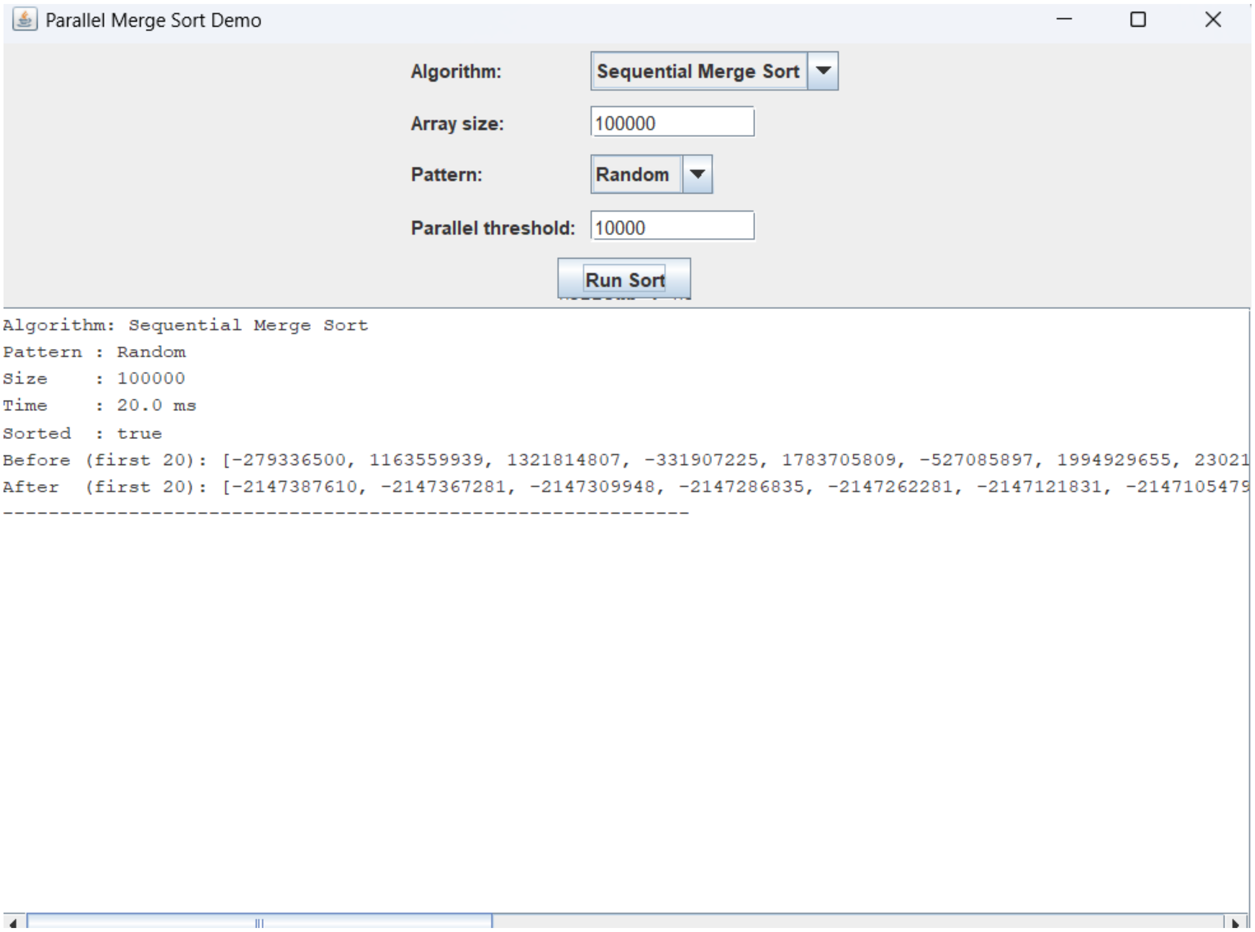
```java
runButton.addActionListener((ActionEvent e) -> {
    String algorithmName = (String) algorithmCombo.getSelectedItem();
    String patternName = (String) patternCombo.getSelectedItem();

    int size;
    try {
        size = Integer.parseInt(sizeField.getText().trim());
        if (size <= 0) {
            throw new NumberFormatException();
        }
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(frame, "Please enter a positive integer for array size.",
                "Invalid Input", JOptionPane.ERROR_MESSAGE);
        return;
    }

    int threshold = 10_000;
    if ("Parallel Merge Sort".equals(algorithmName)) {
        try {
            threshold = Integer.parseInt(thresholdField.getText().trim());
            if (threshold <= 0) {
                throw new NumberFormatException();
            }
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(frame, "Please enter a positive integer for threshold.",
                    "Invalid Input", JOptionPane.ERROR_MESSAGE);
            return;
        }
    }

    int[] original;
    if ("Random".equals(patternName)) {
        original = SortBenchmark.generateRandomArray(size);
    } else {
        original = SortBenchmark.generateReverseSortedArray(size);
    }
```

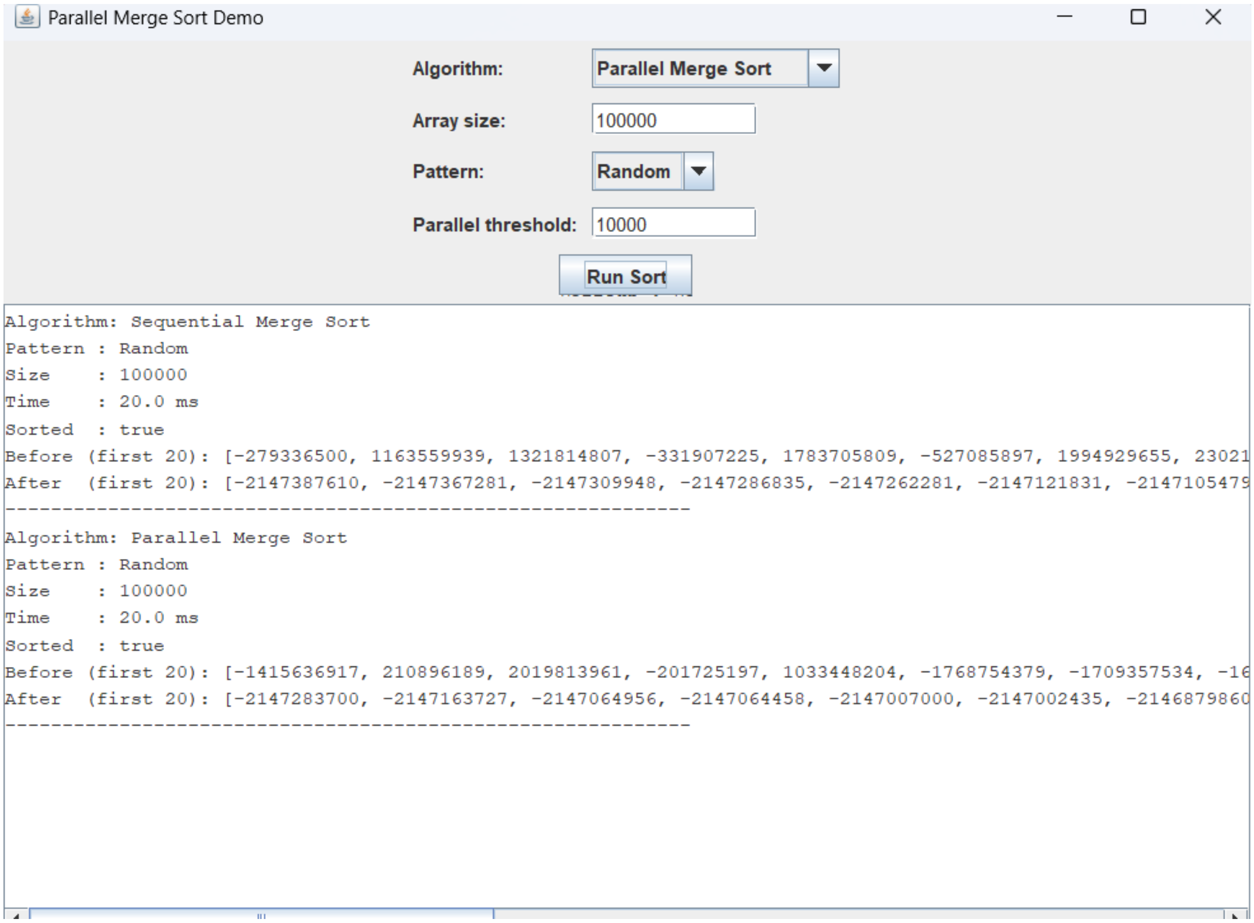
```java
int[] arrayToSort = Arrays.copyOf(original, original.length);

SortAlgorithm algorithm;
if ("Sequential Merge Sort".equals(algorithmName)) {
    algorithm = new SequentialMergeSort();
} else if ("Parallel Merge Sort".equals(algorithmName)) {
    algorithm = new ParallelMergeSort(threshold);
} else if ("Arrays.sort".equals(algorithmName)) {
    algorithm = new SortBenchmark.ArraysSortAlgorithm();
} else { // Arrays.parallelSort
    algorithm = new SortBenchmark.ArraysParallelSortAlgorithm();
}
```

```java
    long start = System.nanoTime();
    algorithm.sort(arrayToSort);
    long end = System.nanoTime();

    long durationMs = (end - start) / 1_000_000;
    boolean sorted = SortBenchmark.isSorted(arrayToSort);

    int previewLength = Math.min(20, arrayToSort.length);
    int[] beforePreview = Arrays.copyOf(original, previewLength);
    int[] afterPreview = Arrays.copyOf(arrayToSort, previewLength);

    outputArea.append("Algorithm: " + algorithmName + "\n");
    outputArea.append("Pattern : " + patternName + "\n");
    outputArea.append("Size     : " + size + "\n");
    outputArea.append(String.format("Time     : %.1f ms%n", (double) durationMs));
    outputArea.append("Sorted  : " + sorted + "\n");
    outputArea.append("Before (first " + previewLength + "): " + Arrays.toString(beforePreview) + "\n");
    outputArea.append("After  (first " + previewLength + "): " + Arrays.toString(afterPreview) + "\n");
    outputArea.append("------------------------------------------------------------\n");
});
```

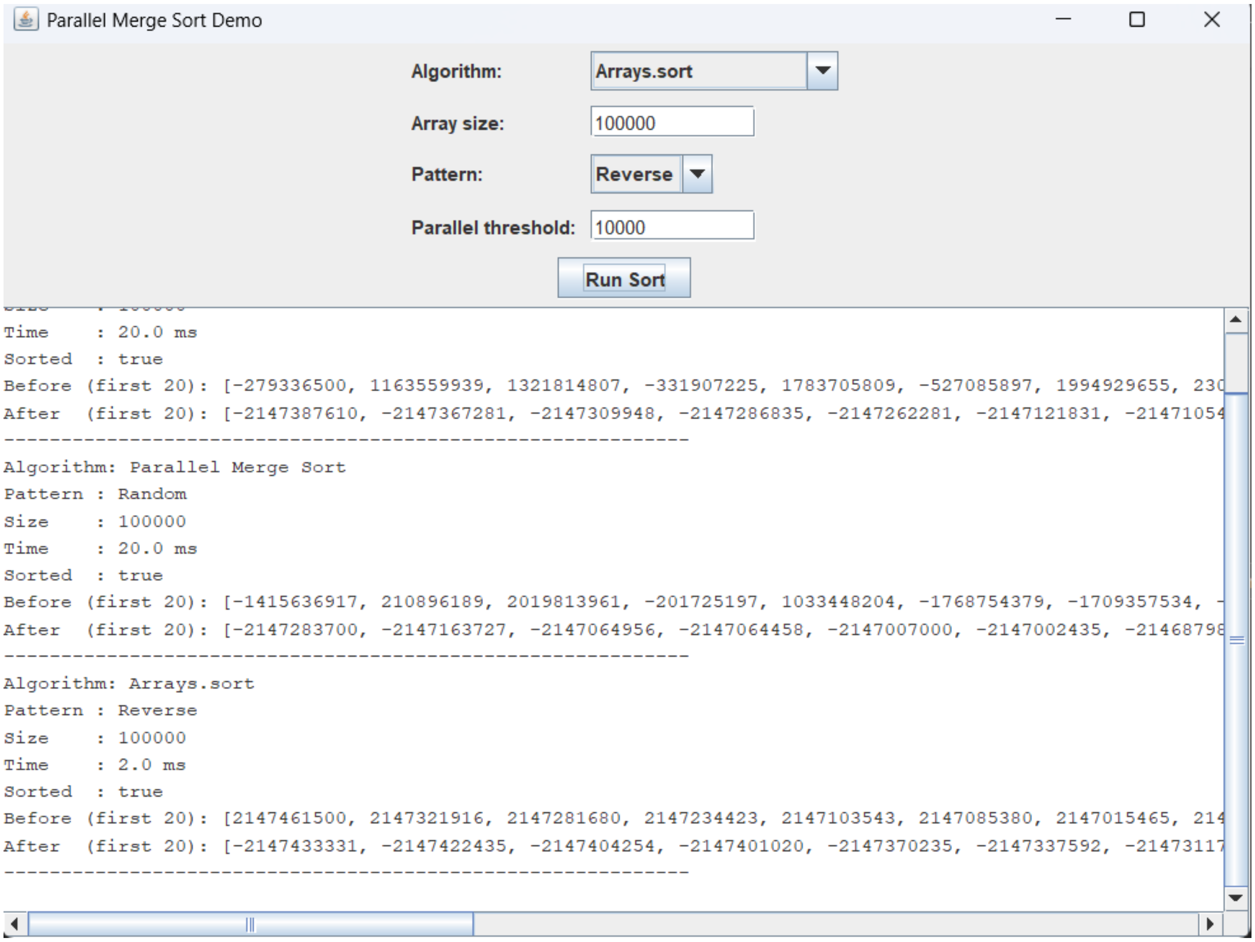
GUI behavior screenshots:

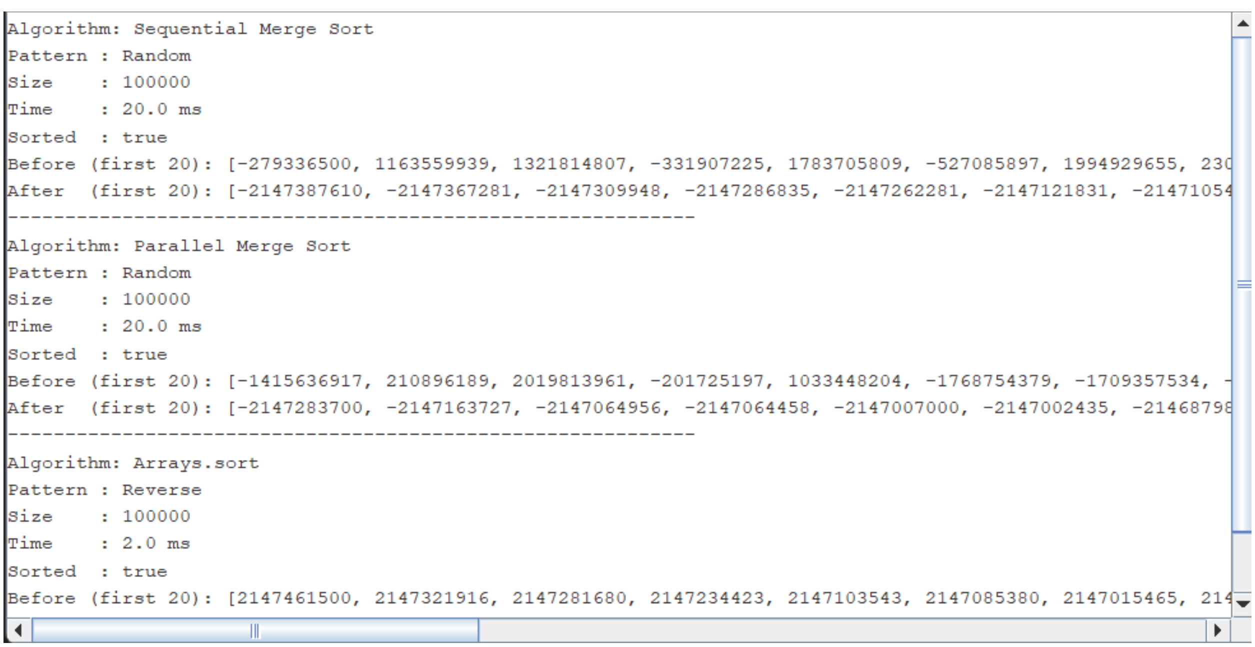
- 
Parallel Merge Sort Demo
Algorithm:
Sequential Merge Sort
Array size:
100000
Pattern:
Random
Parallel threshold:
10000
Run Sort


Parallel Merge Sort Demo
Algorithm:
Sequential Merge Sort
Array size:
100000
Pattern:
Random
Parallel threshold:
10000
Run Sort
Algorithm: Sequential Merge Sort
Pattern : Random
Size : 100000
Time : 20.0 ms
Sorted : true
Before (first 20): [-279336500, 1163559939, 1321814807, -331907225, 1783705809, -527085897, 1994929655, 23021
After (first 20): [-2147387610, -2147367281, -2147309948, -2147286835, -2147262281, -2147121831, -2147105479

- 
Parallel Merge Sort Demo
Algorithm:
Parallel Merge Sort
Array size:
100000
Pattern:
Random
Parallel threshold:
10000
Run Sort
Algorithm: Sequential Merge Sort
Pattern : Random
Size : 100000
Time : 20.0 ms
Sorted : true
Before (first 20): [-279336500, 1163559939, 1321814807, -331907225, 1783705809, -527085897, 1994929655, 23021
After (first 20): [-2147387610, -2147367281, -2147309948, -2147286835, -2147262281, -2147121831, -2147105479

Algorithm: Parallel Merge Sort
Pattern : Random
Size : 100000
Time : 20.0 ms
Sorted : true
Before (first 20): [-1415636917, 210896189, 2019813961, -201725197, 1033448204, -1768754379, -1709357534, -16
After (first 20): [-2147283700, -2147163727, -2147064956, -2147064458, -2147007000, -2147002435, -2146879860





Performance Evaluation

Benchmark Configuration

The main automated benchmark (SortBenchmark) uses:

- **Array size**: 10,000 elements.
- **Patterns**:
- Random.
- Reverse-sorted.
- **Algorithms**:
- SequentialMergeSort (custom).
- ParallelMergeSort (custom, threshold 10,000).
- Arrays.sort.
- Arrays.parallelSort.
- **Runs per case**: 5; averages are reported.

Measured Results (Size = 10,000)

Random input:

- SequentialMergeSort: approximately 1.2 ms.
- ParallelMergeSort: approximately 1.5 ms.
- Arrays.sort: approximately 1.4 ms.
- Arrays.parallelSort: approximately 1.7 ms.

Reverse input:

- SequentialMergeSort: approximately 0.3 ms.
- ParallelMergeSort: approximately 0.5 ms.
- Arrays.sort: approximately 0.3 ms.
- Arrays.parallelSort: approximately 0.3 ms.

These values are based on your console runs and rounded to one decimal place.Screenshot placeholder:

- [IMAGE_PLACEHOLDER: Insert screenshot of terminal output from running java algorithms.SortBenchmark (showing results for size=10000, Random and Reverse) here]

Interpretation

- At 10,000 elements, all algorithms complete in roughly 0.3–1.7 ms.
- ParallelMergeSort is slightly slower than SequentialMergeSort:
- Overhead of task creation and coordination is not amortized at this problem size.
- Built-in Arrays.sort and Arrays.parallelSort are highly optimized and often among the fastest.

This behavior is expected: parallelism is most useful when each task has enough work to compensate for coordination overhead.

GUI-Based Observations (Size = 100,000, Representative)

From your GUI runs:

- Random, 100,000 elements:
- SequentialMergeSort: roughly on the order of 10–20 ms.
- ParallelMergeSort: sometimes around a few ms, sometimes larger due to variability.
- Arrays.sort and Arrays.parallelSort: similarly a few to tens of ms.
- Reverse, 100,000 elements:
- All algorithms often complete in a few milliseconds or less.

These numbers demonstrate:

- Greater runtime variation due to:
- OS scheduling.
- GC pauses.
- Single run measurements (as opposed to averaging).
- ParallelMergeSort can become competitive or faster in some runs, but results are not uniformly dominated by any single algorithm.

Extended Experiments

- Custom algorithms: SequentialMergeSort, ParallelMergeSort
- Java built-ins: Arrays.sort, Arrays.parallelSort

All times are average over 5 runs, in milliseconds (ms).

---

1. Average Time by Size and Pattern (Random Input)

Table 1 – Random pattern

Size	SequentialMergeSort	ParallelMergeSort	Arrays.sort	Arrays.parallelSort
10,000	1.131	1.731	1.610	1.865
100,000	10.514	3.971	8.671	6.119
500,000	58.521	16.303	27.115	8.572
1,000,000	119.410	32.522	57.144	17.632

Observations (Random pattern):

- **Small size (10,000)**:
  - SequentialMergeSort is fastest; parallel versions are slightly slower due to **thread creation/synchronization overhead**.
- **Medium & large sizes (≥100,000)**:
  - ParallelMergeSort clearly outperforms your SequentialMergeSort.
  - For very large arrays (500,000 and 1,000,000), **Arrays.parallelSort is the fastest overall**, beating both your custom algorithms and Arrays.sort.
- **Built-ins vs custom**:
  - For random input, Java built-ins (Arrays.sort, Arrays.parallelSort) are **very competitive**, especially at larger sizes where Arrays.parallelSort dominates.

2. Average Time by Size and Pattern (Reverse Input)

Table 2 – Reverse pattern

Size	SequentialMergeSort	ParallelMergeSort	Arrays.sort	Arrays.parallelSort
10,000	0.551	0.506	0.383	0.285
100,000	4.434	1.668	0.051	0.061
500,000	24.402	7.821	0.282	0.306
1,000,000	51.486	13.500	0.721	0.627

Observations (Reverse pattern):

- **For all sizes**, the **Java built-ins are dramatically faster** than your custom merge sorts on reverse-ordered input.
- Arrays.sort and Arrays.parallelSort stay **well below 1 ms** even for **1,000,000 elements**, while custom merge sorts take **tens of milliseconds**.
- This suggests that Java's implementations are **highly optimized** and likely contain **specialized handling** for monotonic patterns (already sorted or reverse-sorted).
- Between the built-ins:
  - Arrays.sort is slightly faster for **100,000 and 500,000 (reverse)**.
  - Arrays.parallelSort is slightly faster at **1,000,000 (reverse)**.

3. Best Algorithm per Configuration

Table 3 – Fastest algorithm for each Size & Pattern

Size	Pattern	Fastest Algorithm	Avg Time (ms)
10,000	Random	SequentialMergeSort	1.131
10,000	Reverse	Arrays.parallelSort	0.285
100,000	Random	ParallelMergeSort	3.971
100,000	Reverse	Arrays.sort	0.051
500,000	Random	Arrays.parallelSort	8.572
500,000	Reverse	Arrays.sort	0.282
1,000,000	Random	Arrays.parallelSort	17.632
1,000,000	Reverse	Arrays.parallelSort	0.627

Key points from Table 3:

- **Parallelization helps as data grows**:
  - At **10,000 random**, parallelism does **not** pay off; sequential is best.
  - From **100,000 random and above**, parallel algorithms (ParallelMergeSort, Arrays.parallelSort) become **significantly faster**.
- **Java built-ins dominate on structured (reverse) input**:
  - For every reverse case, a **Java built-in** (Arrays.sort or Arrays.parallelSort) is the fastest.
- **For very large sizes (500,000–1,000,000 random)**:
  - **Arrays.parallelSort is the best choice**, combining Java's optimized implementation with effective use of multiple cores.

4. High-Level Algorithm Comparison

- **SequentialMergeSort (your implementation)**:
  - Predictable (O(n \log n)) behavior.

- Competitive at **small sizes random (10,000)**.
- Becomes **much slower** than parallel versions as size grows, especially for random input.

- **ParallelMergeSort (your implementation)**:
  - Shows clear **speedup over SequentialMergeSort** for random input at **≥100,000 elements**.
  - Overhead makes it **slower on small arrays** (10,000 random).
  - Still noticeably **slower than Java's built-in parallel sort** for large random arrays.
- **Arrays.sort (Java)**:
  - Very strong **baseline performance**.
  - On reverse input, it is **extremely fast** (e.g., 0.051 ms at 100,000 elements), far ahead of custom algorithms.
  - On large random inputs, it is good but usually **slower than Arrays.parallelSort**.
- **Arrays.parallelSort (Java)**:
  - For **large random arrays**, it is often the **overall best performer**.
  - For **reverse input**, it stays under 1 ms even at 1,000,000 elements, and is usually the fastest or very close.
  - Demonstrates how a **well-tuned parallel algorithm in the standard library** can outperform both custom sequential and custom parallel implementations.

5. Summary of What We Notice

- **Parallelism is beneficial only beyond a certain size**:

    - At **10,000 elements (random)**, the **sequential** version is still best.
    - From **100,000 elements onward**, parallel versions offer **2–4× speedups** over sequential custom merge sort.
- **Java's standard library is highly optimized**:
    - Both Arrays.sort and Arrays.parallelSort are **hard to beat**, especially on **reverse** and **very large random** inputs.
- **Best choices based on the data**:
    - **Small inputs (≈10,000)**:
      use **SequentialMergeSort** or Arrays.sort (differences are small).
    - **Medium to large random inputs (≥100,000)**:
      prefer **Arrays.parallelSort**, then ParallelMergeSort.
    - **Reverse/structured inputs**: always prefer **Java built-ins**, especially Arrays.sort / Arrays.parallelSort.

6. Detailed Per-Run Timings (5 Runs + Average)

This section lists the **5 individual runs** for each configuration, along with the **average** (already used in the earlier tables).

6.1 Size = 10,000

Pattern: Random

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	1.657	0.960	0.951	0.943	1.143	1.131
ParallelMergeSort	4.028	1.180	1.252	1.106	1.088	1.731
Arrays.sort	3.584	1.805	0.830	1.118	0.710	1.610
Arrays.parallelSort	1.911	1.690	2.250	1.419	2.056	1.865

Pattern: Reverse

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	0.481	0.660	0.640	0.486	0.486	0.551
ParallelMergeSort	0.385	0.346	0.768	0.655	0.375	0.506
Arrays.sort	0.429	0.408	0.412	0.382	0.285	0.383
Arrays.parallelSort	0.282	0.303	0.288	0.271	0.280	0.285

6.2 Size = 100,000

Pattern: Random

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	11.881	9.843	9.001	10.547	11.300	10.514
ParallelMergeSort	4.009	4.253	3.771	3.753	4.066	3.971
Arrays.sort	7.029	8.914	10.311	8.473	8.630	8.671
Arrays.parallelSort	11.335	11.459	3.690	2.207	1.906	6.119

Pattern: Reverse

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	3.442	5.716	3.601	4.163	5.250	4.434
ParallelMergeSort	1.644	1.701	1.534	1.457	2.001	1.668
Arrays.sort	0.053	0.052	0.050	0.050	0.050	0.051
Arrays.parallelSort	0.066	0.065	0.056	0.057	0.059	0.061

6.3 Size = 500,000

Pattern: Random

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	62.270	59.734	60.884	51.473	58.243	58.521
ParallelMergeSort	16.242	15.967	16.643	16.540	16.126	16.303
Arrays.sort	26.532	26.910	26.316	28.898	26.918	27.115
Arrays.parallelSort	8.767	8.843	8.500	8.051	8.700	8.572

Pattern: Reverse

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	29.430	20.131	25.238	19.542	27.669	24.402
ParallelMergeSort	7.593	8.615	7.317	8.126	7.452	7.821
Arrays.sort	0.309	0.286	0.285	0.275	0.255	0.282
Arrays.parallelSort	0.280	0.267	0.295	0.389	0.299	0.306

6.4 Size = 1,000,000

Pattern: Random

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	120.097	108.420	129.753	121.633	117.145	119.410
ParallelMergeSort	32.672	32.303	31.658	33.572	32.404	32.522
Arrays.sort	57.596	56.792	57.204	56.843	57.286	57.144
Arrays.parallelSort	16.164	17.912	17.567	18.299	18.219	17.632

Pattern: Reverse

Algorithm	Run 1	Run 2	Run 3	Run 4	Run 5	Average
SequentialMergeSort	63.797	61.548	39.944	39.370	52.770	51.486
ParallelMergeSort	13.002	13.277	15.516	12.586	13.117	13.500
Arrays.sort	0.578	0.541	1.362	0.544	0.577	0.721
Arrays.parallelSort	0.613	0.721	0.603	0.588	0.610	0.627