# Project 2 – Parallel Merge Sort with Fork/Join

## 1. Introduction

This project implements and evaluates sorting of integer arrays (int[]) using: (1) a custom **sequential merge sort**, (2) a custom **parallel merge sort** built on Java's **Fork/Join** framework, and (3) Java standard-library baselines (**Arrays.sort** and **Arrays.parallelSort**). The purpose is to quantify when task-based parallelism improves performance over a straightforward sequential algorithm, and to compare custom implementations against highly optimized built-in methods.

Parallelism is motivated by modern multi-core CPUs: large sorting workloads can be decomposed into independent subproblems, executed concurrently, then combined. The project includes a console benchmark (SortBenchmark) and an interactive Swing GUI (SortGUI) to visualize time-vs-size behavior.

## 2. Merge Sort Algorithm Overview

Merge sort is a **divide-and-conquer** algorithm:

- **Divide**: recursively split the array into two halves until subarrays are of size 0 or 1.
- **Conquer**: sort each half (recursively).
- **Combine (merge)**: merge the two sorted halves into one sorted segment by repeatedly selecting the smallest next element from the two halves.

**Merge phase**: given two sorted subarrays ([left..mid]) and ([mid+1..right]), merging uses two pointers (one per half) and writes the smaller element into the output. In this project, merging is implemented using a temporary buffer to avoid overwriting values needed for future comparisons.

**Time complexity**: merge sort runs in (O(n \log n)) in the **best**, **average**, and **worst** cases. **Space complexity**: (O(n)) additional space for the temporary buffer.

Merge sort is suitable for parallel execution because the two recursive calls that sort the left and right halves are independent until the merge step, enabling concurrent execution of subproblems.

## 3. Sequential Merge Sort

The sequential implementation is provided by algorithms.SequentialMergeSort, which implements the project interface algorithms.SortAlgorithm (void sort(int[] array)).

**Recursive structure and merging**:

- The sort method allocates a single temp array of the same length as the input and calls a recursive mergeSort(array, left, right, temp).
- The recursion computes mid = left + (right - left) / 2, sorts both halves, and calls merge(array, left, mid, right, temp).
- The merge routine copies the segment into temp and merges back into array using two pointers; the comparison uses <=, yielding stable behavior for equal keys (deterministic ordering during merge).

**Edge cases**:

- **Empty array / size 1 / null**: sort returns immediately for null or length <= 1.
- **Already sorted input**: SequentialMergeSort includes an isSorted check and exits early if the array is already in non-decreasing order.

## 4. Parallel Merge Sort Using Fork/Join

The parallel implementation is algorithms.ParallelMergeSort, which uses Java's Fork/Join framework:

- A ForkJoinPool is used via ForkJoinPool.commonPool().
- Work is represented by an inner RecursiveAction called MergeSortTask, because sorting is performed in-place and does not need to return a value.

Parallelism is achieved by expressing merge sort subproblems ("sort subarray ([left..right])") as tasks. Large tasks split into two subtasks (left half and right half), which are executed concurrently; after both complete, the parent task merges the two sorted halves.

### 4.1 Split and Merge Strategy

This section explicitly states the split/merge rationale implemented in the code.

**How the array is split**:

- Each task operates on a contiguous segment ([left..right]).
- The segment is divided at: [ mid = left + \frac{(right - left)}{2} ]
- Two child tasks are constructed for:
    - Left half: ([left..mid])

- Right half: ([mid+1..right])

**How subtasks are forked and joined**:

- The parent task executes both subtasks using invokeAll(leftTask, rightTask).
- invokeAll schedules the tasks in the Fork/Join pool and waits (joins) until both subtasks finish, ensuring the two halves are fully sorted before merging.

**How merging is performed after completion**:

- Once both subtasks complete, the parent performs merge(array, temp, left, mid, right).
- The merge copies array[left..right] into a shared temporary buffer temp[left..right] and merges back into array.

**Rationale**: the two halves are independent sorting subproblems; executing them concurrently maximizes available parallelism. The merge is performed only after both halves are sorted, because correctness requires merging sorted subarrays. Using a temporary buffer prevents overwrite hazards when reading and writing within the same segment.

### 4.2 Threshold Choice and Rationale

**Exact threshold used in the implementation**:

- ParallelMergeSort default constructor sets the threshold to **10,000** elements.
- The console benchmark algorithms.SortBenchmark explicitly uses **parallelThreshold = 10,000** for its benchmarked sizes.
- The provided results table also includes a **10,000-element** case where ParallelMergeSort was run with **threshold = 1,000** (as documented in sorting_results_comparison.md).

**Why a threshold is necessary**: Fork/Join parallelism introduces overhead (task allocation, scheduling, context coordination, and join synchronization). If tasks are too small, overhead dominates and can make parallel execution slower than sequential sorting. Therefore, the project uses a threshold rule:

- If segmentLength <= threshold, the task performs a sequential merge sort on that segment.
- Otherwise, the task splits into subtasks and continues in parallel.

**Trade-offs**:

- **Threshold too small**: creates many fine-grained tasks, increasing overhead and contention; performance may degrade (especially for small arrays).
- **Threshold too large**: reduces available parallelism and may underutilize CPU cores, approaching sequential behavior.

**Rationale for 10,000**: a threshold of **10,000** provides a practical granularity for large arrays (100,000–1,000,000 in this project's console benchmark), balancing task overhead against parallel speedup. The 10,000-element experiment using threshold **1,000** illustrates that for smaller inputs, task overhead can outweigh benefits (as seen in the Random 10,000 results).

## 5. Object-Oriented Design

The project uses a modular OOP structure to separate algorithm logic from benchmarking and visualization:

- **SortAlgorithm (interface)**: defines a uniform API void sort(int[] array).
- **SequentialMergeSort**: implements SortAlgorithm with a standard recursive merge sort and explicit edge-case handling.
- **ParallelMergeSort**: implements SortAlgorithm using Fork/Join tasks (RecursiveAction) and a configurable threshold for granularity control.
- **SortBenchmark**: generates input arrays (Random and Reverse), runs multiple algorithms, times them, verifies sorted output, and reports average runtimes.

This design improves:

- **Modularity**: algorithms are interchangeable behind the same interface.
- **Extensibility**: new algorithms can be added without changing benchmark/GUI logic.
- **Separation of concerns**: sorting, benchmarking, and visualization responsibilities are kept distinct.

## 6. Experimental Setup

Benchmarking in this project evaluates runtime (ms) under controlled input generation:

- **Array sizes tested (from provided results)**: 10,000; 100,000; 500,000; 1,000,000.

    - The console benchmark SortBenchmark iterates sizes: 100,000; 500,000; 1,000,000.
    - The 10,000 case is included in the provided results table (see Section 7) and used ParallelMergeSort with threshold 1,000 as stated in that file.

- **Input patterns**: Random and Reverse (reverse-sorted).
- **Runs and averaging**: all reported times are **averages over 5 runs**.
- **Timing method**: uses System.nanoTime() and converts to milliseconds.
- **Fairness and correctness**: for each run, the benchmark sorts a fresh copy of the same base array and validates that the output is sorted.

The GUI (SortGUI) supports interactive experiments with algorithm selection, array size, pattern, and a user-specified parallel threshold, and plots time vs. size on the "Performance Chart" tab.

## 7. Results and Performance Analysis

### 7.1 Results Tables

The following tables are taken from sorting_results_comparison.md. Times are **average of 5 runs**, in **milliseconds (ms)**.

**Table 1 — Random input (avg ms)**:

| Size | SequentialMergeSort | ParallelMergeSort | Arrays.sort | Arrays.parallelSort |
|---|---|---|---|---|
| 10,000 | 1.157 | 2.246 | 1.528 | 2.067 |
| 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 |

**Table 2 — Reverse input (avg ms)**:

| Size | SequentialMergeSort | ParallelMergeSort | Arrays.sort | Arrays.parallelSort |
|---|---|---|---|---|
| 10,000 | 0.441 | 0.328 | 0.344 | 0.327 |
| 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 |

### 7.2 Graph Analysis

The GUI's "Performance Chart" tab plots **time (ms)** vs **array size (n)** as points colored by algorithm. The following trends are consistent with the results tables:

- **Impact of input size (Random)**: as (n) grows, the parallel approaches become increasingly beneficial. At 10,000 elements, parallel overhead dominates (custom ParallelMergeSort is slower than SequentialMergeSort), but from 100,000 onward the parallel methods outperform the sequential custom implementation.
- **Effect of parallelism (custom vs built-in)**: ParallelMergeSort improves substantially over SequentialMergeSort for Random inputs at larger sizes, confirming that fork/join parallelism amortizes overhead when tasks are sufficiently large. Arrays.parallelSort provides the best results at the largest Random sizes, indicating further optimizations beyond the custom implementation.
- **Comparison on Reverse input**: the graphs and Table 2 show Java built-ins (Arrays.sort / Arrays.parallelSort) are extremely fast on reverse input across all tested sizes, while both custom merge sorts remain in the millisecond-to-tens-of-milliseconds range. This suggests the standard library benefits from highly tuned implementations and/or adaptive optimizations for structured patterns.

## 8. Discussion

The experiments highlight core trade-offs in task-parallel divide-and-conquer algorithms:

- **When parallel merge sort helps**: for sufficiently large Random arrays (≥ 100,000 in this dataset), dividing work across cores reduces total runtime relative to sequential merge sort. The custom ParallelMergeSort demonstrates clear speedups over the custom sequential version at 100,000–1,000,000 elements.
- **When overhead outweighs gains**: for small arrays (e.g., 10,000 Random), Fork/Join overhead can dominate, making parallel execution slower than sequential sorting. This directly motivates a threshold-based granularity control.
- **Role of threshold**: the threshold is a performance tuning parameter that balances concurrency against overhead. In this project, **10,000** is used as the primary benchmark threshold and aligns with improved performance on large arrays; the 10,000-element case using **1,000** demonstrates that more aggressive task splitting does not necessarily improve performance at small sizes.
- **Effect of input pattern**: merge sort's (O(n \log n)) work is largely pattern-independent, so the custom implementations do not achieve the extreme improvements seen in Java built-ins on Reverse inputs. Built-in algorithms likely

incorporate low-level optimizations and adaptive behavior that are outside the scope of this educational implementation.

**9. Conclusion**

This project implemented a modular sorting framework in Java and compared a custom sequential merge sort, a custom Fork/Join parallel merge sort, and Java's built-in sorting methods. The results show that:

- Fork/Join parallelism can significantly improve merge sort performance for large Random inputs when task granularity is controlled by an appropriate threshold (**10,000** in the benchmarked implementation).
- For small inputs, parallel overhead can outweigh benefits, reinforcing the necessity of a threshold that switches to sequential processing for small segments.
- Java's Arrays.sort and Arrays.parallelSort remain strong baselines; Arrays.parallelSort is typically best for large Random arrays, while both built-ins dominate on Reverse input in the provided results.