In **Java 8**, **parallel array sorting** is a feature introduced in the java.util.Arrays class to leverage **multi-core processors** for faster sorting of large arrays.

**🔍 What Is Parallel Array Sorting?**

Parallel sorting uses **multithreading** internally to divide the array into chunks, sort them **concurrently**, and then **merge** them — all handled by the **Fork/Join framework** under the hood.

**📌 Syntax**

Arrays.parallelSort(array);

**✅ Available Overloads**

Java 8 provides parallel sort for:

Arrays.parallelSort(int[] a);

Arrays.parallelSort(long[] a);

Arrays.parallelSort(double[] a);

Arrays.parallelSort(T[] a); // generic with Comparable

Arrays.parallelSort(T[] a, Comparator<? super T> cmp);

Arrays.parallelSort(array, fromIndex, toIndex);

**⚙️ Internal Working**

Java 8 internally uses the **Fork/Join framework** introduced in Java 7:

1. The array is split into subarrays (typically if size > 8192).
2. Each subarray is sorted using **parallel threads**.
3. The sorted subarrays are **merged** using an efficient merge strategy.

**📊 When to Use**

| **Use Case** | **Recommendation** |
| --- | --- |
| Arrays with >10,000 elements | ✅ Use parallelSort |
| Arrays with <1,000 elements | ❌ Use regular Arrays.sort() (less overhead) |
| Single-core machines | ❌ No benefit |
| Multi-core machines | ✅ Parallel sort leverages threads |

**🧠 Difference Between Arrays.sort() vs Arrays.parallelSort()**

| **Feature** | **Arrays.sort()** | **Arrays.parallelSort()** |
| --- | --- | --- |
| Threaded? | No | Yes (multi-threaded) |
| Uses Fork/Join? | No | Yes |
| Suitable for Large? | Less Efficient | More Efficient |
| Overhead? | Low | Higher for small arrays |

**🚧 Caveats**

* Overhead of managing threads can outweigh benefits for small arrays.
* Not suitable for **real-time** systems needing predictability.
* Not available for List, only for **arrays**.

**🔎 Bonus: Custom Comparator Example**

String[] names = {"Zara", "Bob", "Alex"};

Arrays.parallelSort(names, (a, b) -> b.compareTo(a)); // reverse sort

System.out.println(Arrays.toString(names));

Here’s the **performance comparison** of sorting a large array (1 million integers):

| **Method** | **Time Taken (seconds)** |
| --- | --- |
| Arrays.sort() (sequential / sorted()) | **0.57 sec** |
| Arrays.parallelSort() (simulated with multiprocessing) | **1.31 sec** |

**⚠️ Observations**

* In this Python simulation, **sequential sorting was faster** than the parallel version.
* This is because:
  + **Parallel merge overhead** is significant.
  + Python’s multiprocessing has **higher inter-process communication cost**.
  + Java's parallelSort() is **better optimized with Fork/Join**.

**✅ Conclusion (For Java 8)**

* Use Arrays.parallelSort() **only for large arrays** (e.g., >10,000 elements).
* Performance benefit depends on:
  + Number of CPU cores
  + Array size
  + Type of data
* Always **benchmark in your environment** before choosing.

Here’s a complete **Java benchmarking example** comparing Arrays.sort() and Arrays.parallelSort() on a large array:

**📊 Key Insights**

* **parallelSort() outperforms sort()** for large arrays on multi-core CPUs.
* For small arrays (e.g., <10,000), sort() may be faster due to **low thread management overhead**.
* Always test with **actual data sizes** in your application.

Here’s a version of the benchmark using **custom objects** and a **custom Comparator**, demonstrating both Arrays.sort() and Arrays.parallelSort() with Java 8 features.

**✅ Java Code: Benchmark with Custom Objects + Comparator**

**💡 Summary**

| **Feature** | **Arrays.sort()** | **Arrays.parallelSort()** |
| --- | --- | --- |
| Threading | Single-threaded | Multi-threaded (Fork/Join) |
| Custom Comparator | ✅ Supported | ✅ Supported |
| Performance | Slower on large datasets | Faster on large datasets |
| Overhead | Low | High for small datasets |