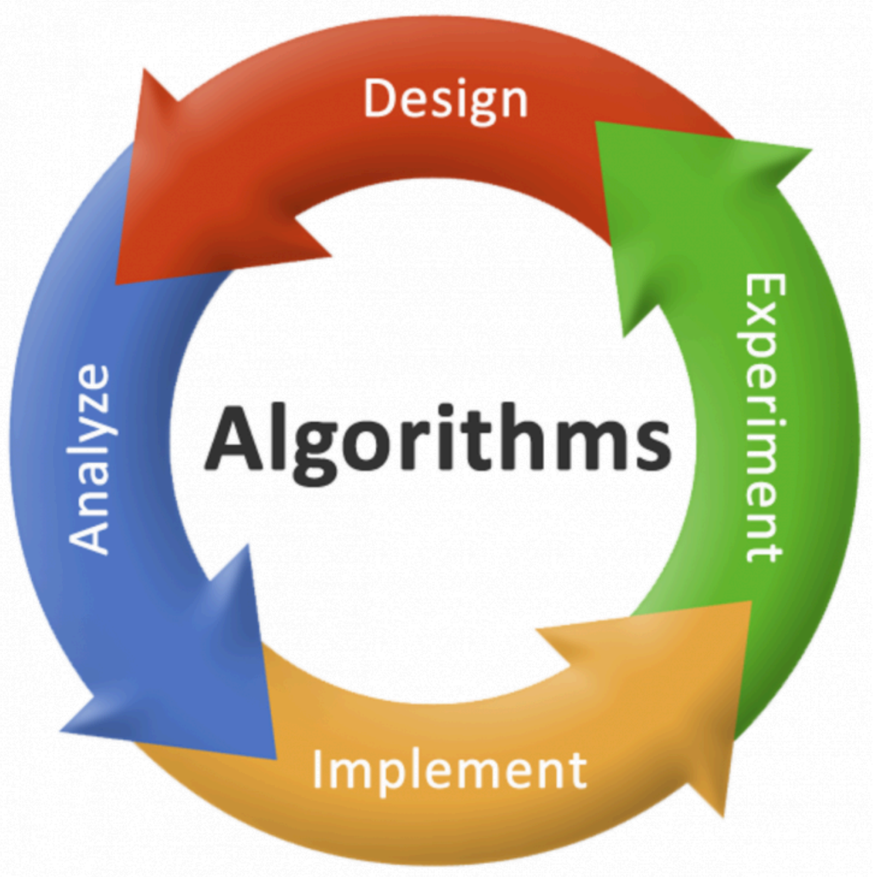
OPTIMIZING SORTING PERFORMANCE WITH PARALLEL ALGORITHMS IN JAVA

Introduction

In today's world of ever-growing data, sorting is a fundamental operation used across many applications—from databases and search engines to scientific simulations and financial systems. While traditional sequential sorting algorithms like Quick Sort and Merge Sort are efficient for moderate-sized data, they can become bottlenecks when dealing with large datasets.

This project aims to explore and compare the performance of sequential and parallel sorting algorithms using Java. By leveraging multithreading and the Fork/Join framework, we demonstrate how parallelism can significantly improve sorting efficiency, especially on multi-core processors.



Divide and Conquer in Sorting Algorithms

How it Works in Sorting:

Divide

Split the array into two or more smaller sub-arrays

Conquer

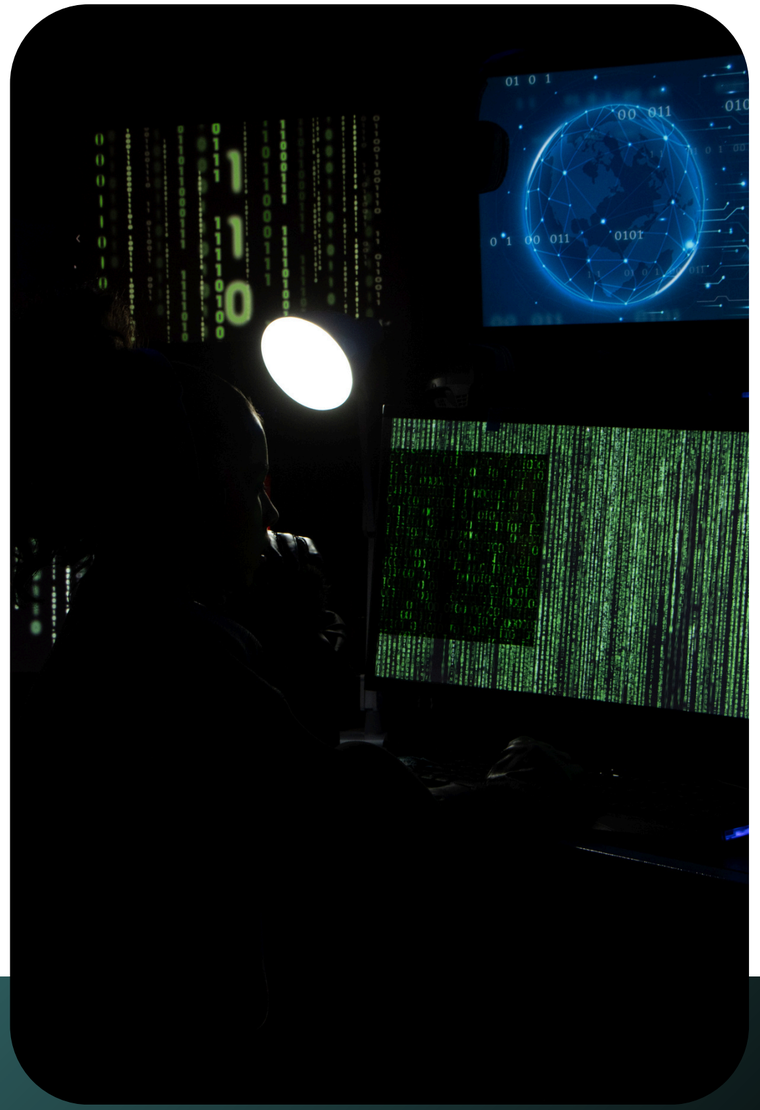
Recursively sort each sub-array

Combine

Merge the sorted sub-arrays into a final sorted array

Why it's Important in Sorting:

- Efficiency: Reduces time complexity compared to brute-force sorting (like Bubble Sort).
- Recursion-Friendly: Naturally fits recursive strategies like in Merge Sort and Quick Sort.
- Parallelization: Sub-arrays can be sorted independently, making it ideal for parallel processing.
- Scalability: Performs well with large datasets when combined with multi-threading.



Project Idea

Project Idea
The main idea of our project is to develop a Java-based application that demonstrates and compares the performance of sequential and parallel sorting algorithms using a simple and interactive Graphical User Interface (GUI).

1

Implement multiple sorting algorithms (Merge Sort & Quick Sort).

2

Use multithreading to parallelize sorting using Java Fork/Join Framework.

3

Allow users to choose between sequential or parallel execution.

4

Enable users to upload unsorted files and visualize sorted results with timing comparisons.

Code Comparasion

Parallel QuickSort

```java
package parallel_sort;

import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class ParallelQuickSort {
    public static void sort(int[] arr) {
        ForkJoinPool forkJoinPool = new ForkJoinPool(6);
        forkJoinPool.invoke(new QuickSortTask(arr, 0, arr.length - 1));
    }

    private static class QuickSortTask extends RecursiveTask<Void> {
        private int[] arr;
        private int low, high;

        public QuickSortTask(int[] arr, int low, int high) {
            this.arr = arr;
            this.low = low;
            this.high = high;
        }

        @Override
        protected Void compute() {
            if (low < high) {
                int pivotIndex = partition(arr, low, high);
                QuickSortTask leftTask = new QuickSortTask(arr, low, pivotIndex - 1);
                QuickSortTask rightTask = new QuickSortTask(arr, pivotIndex + 1, high);

                leftTask.fork();
                rightTask.fork();

                leftTask.join();
                rightTask.join();
            }
            return null;
        }

        private int partition(int[] arr, int low, int high) {
            int pivot = arr[high];
            int i = low - 1;
            for (int j = low; j < high; j++) {
                if (arr[j] <= pivot) {
                    i++;
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
            int temp = arr[i + 1];
            arr[i + 1] = arr[high];
            arr[high] = temp;
            return i + 1;
        }
    }
}
```

Parallel MergeSort

```java
package parallel_sort;

import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class ParallelMergeSort {
    public static void sort(int[] arr) {
        ForkJoinPool forkJoinPool = new ForkJoinPool(6);
        forkJoinPool.invoke(new MergeSortTask(arr, 0, arr.length - 1));
    }

    private static class MergeSortTask extends RecursiveTask<Void> {
        private int[] arr;
        private int left, right;

        public MergeSortTask(int[] arr, int left, int right) {
            this.arr = arr;
            this.left = left;
            this.right = right;
        }

        @Override
        protected Void compute() {
            if (right - left < 2) {
                if (arr[left] > arr[right]) {
                    int temp = arr[left];
                    arr[left] = arr[right];
                    arr[right] = temp;
                }
                return null;
            }
            int middle = (left + right) / 2;
            MergeSortTask leftTask = new MergeSortTask(arr, left, middle);
            MergeSortTask rightTask = new MergeSortTask(arr, middle + 1, right);

            leftTask.fork();
            rightTask.fork();

            leftTask.join();
            rightTask.join();

            merge(arr, left, middle, right);
            return null;
        }

        private void merge(int[] arr, int left, int middle, int right) {
            int[] temp = new int[right - left + 1];
            int i = left, j = middle + 1, k = 0;

            while (i <= middle && j <= right) {
                if (arr[i] < arr[j]) {
                    temp[k++] = arr[i++];
                } else {
                    temp[k++] = arr[j++];
                }
            }
            while (i <= middle) {
                temp[k++] = arr[i++];
            }
            while (j <= right) {
                temp[k++] = arr[j++];
            }
            System.arraycopy(temp, 0, arr, left, temp.length);
        }
    }
}
```

Seq MergeSort

```java
package parallel_sort;

public class SequentialMergeSort {
    public static void sort(int[] arr) {
        if (arr.length < 2) return;
        mergeSort(arr, 0, arr.length - 1);
    }

    private static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    private static void merge(int[] arr, int left, int mid, int right) {
        int[] leftArr = new int[mid - left + 1];
        int[] rightArr = new int[right - mid];

        System.arraycopy(arr, left, leftArr, 0, leftArr.length);
        System.arraycopy(arr, mid + 1, rightArr, 0, rightArr.length);

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

Seq QuickSort

```java
package parallel_sort;

public class SequentialQuickSort {
    public static void sort(int[] arr) {
        quickSort(arr, 0, arr.length - 1);
    }

    private static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
}
```

1Mill Num File

Algorithm	Sequential Time (ms)	Parallel Time (ms)
Merge Sort	119 ms	76 ms
Quick Sort	132 ms	50 ms

2Mill Num File

Algorithm	Sequential Time (ms)	Parallel Time (ms)
Merge Sort	320 ms	59 ms
Quick Sort	127 ms	106 ms