



Mini Project Report on

“IMPLEMENT MERGE SORT AND MULTITHREADED MERGE SORT.”

Submitted *By*

Mr. Anshul Jitendra Jadhav [Roll no: 40]

*In partial fulfilment of the requirements for The
award of the degree of*

**Bachelor in
COMPUTER ENGINEERING
For Academic Year 2023 – 2024**

DEPARTMENT OF COMPUTER ENGINEERING

K.K.Wagh Institute of Engineering Education and Research,
Nashik – 422003.

Certificate

This is to Certify That

Mr. Anshul Jitendra Jadhav [Roll no: 40]

Has completed the necessary

Mini Project Work and Prepared the Report on

**“IMPLEMENT MERGE SORT AND
MULTITHREADED MERGE SORT.”**

*in Satisfactorily manner as a fulfilment of the requirement of the award of
degree of the Bachelor in Computer Engineering in the Academic Year*

2023 – 2024

Prof.K.P.Birla

Project Guide

Prof.Dr.S.S.Sane

H.O.D

OBJECTIVE:

- To understand the Implementation of Merge Sort and Multithreaded Merge Sort.
- To Understand how Merge Sort and Multi threaded Merge Sort Works.
- To Analyse Time Complexity of Both Algorithms.

MERGE SORT:

Merge sort is a highly efficient, stable, and comparison-based sorting algorithm that uses the divide-and-conquer approach to sort an array or list of elements. The algorithm operates by breaking the unsorted array into smaller subarrays until each subarray consists of only one or zero elements (the base case), which are trivially sorted. It then recursively merges these smaller subarrays to produce larger, sorted subarrays until the entire array is sorted.

The process begins by dividing the original array into two halves, a "left" and a "right" subarray. Each subarray is then recursively sorted using the same merge sort algorithm, effectively dividing the problem into smaller and smaller pieces. Once the base case is reached (single-element subarrays), the merging phase begins. In this phase, the algorithm takes the two sorted subarrays and combines them to produce a single, larger sorted array. This merging process continues until the entire array is sorted.

The key characteristic of merge sort is its remarkable stability; it preserves the relative order of equal elements, making it suitable for a wide range of applications where maintaining the initial order of equivalent elements is important. While merge sort's time complexity is $O(n \log n)$ in the worst, average, and best cases, making it highly efficient for large datasets, it does require additional memory space for the merging process. Nonetheless, merge sort remains a popular choice for sorting tasks where stability and efficiency are paramount.

How Merge Sort Works?

1. Divide:

- The input array is divided into two halves.
- This process is repeated recursively for each half until each subarray contains only one element, essentially making them sorted.

2. Conquer:

- Once the array is divided into single-element subarrays, the merging process begins.
- Pairs of subarrays are merged back together to form larger sorted subarrays.
- The merging process continues until there is only one sorted subarray, which is the fully sorted array.

3. Merge Operation:

- During the merge operation, two subarrays are combined to create a single, larger sorted subarray.
- The elements of the two subarrays are compared one by one, and the smaller element is placed in the resulting merged array.
- This process continues until all elements from both subarrays are placed in the merged array.
- The key to merge sort's efficiency is that it can merge two sorted subarrays quickly.

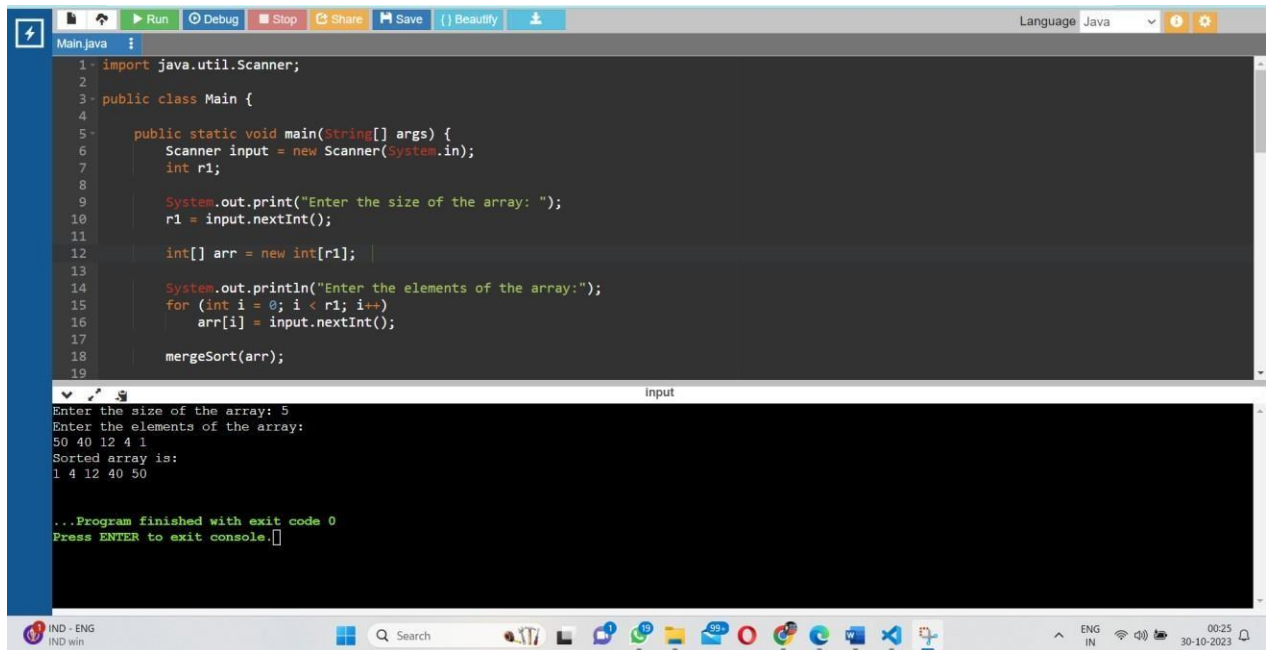
4. Recursion and Backtracking:

- Merge sort uses recursion to divide the array into smaller subarrays.
- Once the smallest subarrays are sorted, the recursion backtracks and the merge operation combines these sorted subarrays until the entire array is sorted.

Time Complexity:

- Merge sort is known for its stable time complexity, making it an efficient sorting algorithm.
- Best-case time complexity: $O(n \log n)$
- Average-case time complexity: $O(n \log n)$
- Worst-case time complexity: $O(n \log n)$
- Merge sort exhibits the same time complexity in all cases, which makes it a reliable choice for sorting large datasets.

OUTPUT:



The screenshot displays an IDE window with a Java file named 'Main.java'. The code implements a merge sort algorithm. The console output shows the program's execution, including prompts for array size and elements, the resulting sorted array, and a completion message.

```
1 import java.util.Scanner;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Scanner input = new Scanner(System.in);
7         int r1;
8
9         System.out.print("Enter the size of the array: ");
10        r1 = input.nextInt();
11
12        int[] arr = new int[r1];
13
14        System.out.println("Enter the elements of the array:");
15        for (int i = 0; i < r1; i++)
16            arr[i] = input.nextInt();
17
18        mergeSort(arr);
19    }
20 }
```

Input

```
Enter the size of the array: 5
Enter the elements of the array:
50 40 12 4 1
Sorted array is:
1 4 12 40 50

...Program finished with exit code 0
Press ENTER to exit console.
```

MULTITHREADED MERGE SORT:

Multithreaded merge sort is a parallelized version of the traditional merge sort algorithm, designed to take advantage of multi-core processors and enhance sorting performance on large datasets. In multithreaded merge sort, the sorting process is divided into multiple threads or concurrent tasks that run simultaneously, making better use of available hardware resources. Here's an overview of the key features and benefits of multithreaded merge sort:

1. **Parallelism:** Instead of sorting the entire dataset sequentially, multithreaded merge sort splits the data into smaller segments and assigns each segment to a separate thread. These threads can then sort their assigned portions concurrently.
2. **Improved Efficiency:** By distributing the sorting work among multiple threads, the algorithm can take full advantage of multi-core processors, significantly reducing the time required to sort a large dataset.
3. **Load Balancing:** To ensure equal work distribution among threads, load-balancing techniques are often applied. This helps prevent situations where some threads complete their tasks much faster than others, maximizing overall efficiency.
4. **Merging Threads:** Once the individual segments are sorted, they are merged back together concurrently, usually in a hierarchical fashion. This merging process is similar to the traditional merge step in merge sort but is executed in parallel, further enhancing performance.
5. **Complexity:** The time complexity of multithreaded merge sort remains $O(n \log n)$ for sorting, as in the traditional merge sort, but with a significantly reduced constant factor. However, managing threads and synchronization may introduce some overhead.
6. **Scalability:** Multithreaded merge sort can scale effectively with the number of available cores, which is particularly valuable for modern processors with multiple cores.
7. **Thread Safety:** Proper synchronization mechanisms, such as mutexes or semaphores, are essential to ensure thread safety and avoid race conditions during parallel execution.

Multithreaded merge sort is a powerful sorting algorithm for exploiting the full potential of modern hardware, making it well-suited for sorting extensive datasets in applications like big data processing, scientific computing, and real-time data analysis where performance optimization is critical. However, it does require careful implementation to manage threads and ensure data integrity.

How Multithreaded Merge Sort Works?

1. Divide:

- The input array is divided into smaller subarrays in a recursive manner until each subarray contains a manageable number of elements or reaches a base case.
- Each subarray is assigned to a separate thread for parallel sorting.

2. Conquer:

- Multiple threads sort their respective subarrays independently using standard merge sort or, in the case of a further optimized multithreaded merge sort, a similar parallel merge sort algorithm.

3. Merge Operation (Parallel Merge):

- After the subarrays are sorted, the final merging step combines these sorted subarrays.
- Parallel merging is challenging and requires careful synchronization to maintain the order and avoid race conditions.
- In a highly optimized multithreaded merge sort, techniques such as parallel merging networks or other parallel sorting algorithms are employed for the merge step.

4. Thread Management:

- Thread management is crucial to efficiently utilize available cores and resources. It typically involves creating a thread pool and assigning tasks to individual threads.

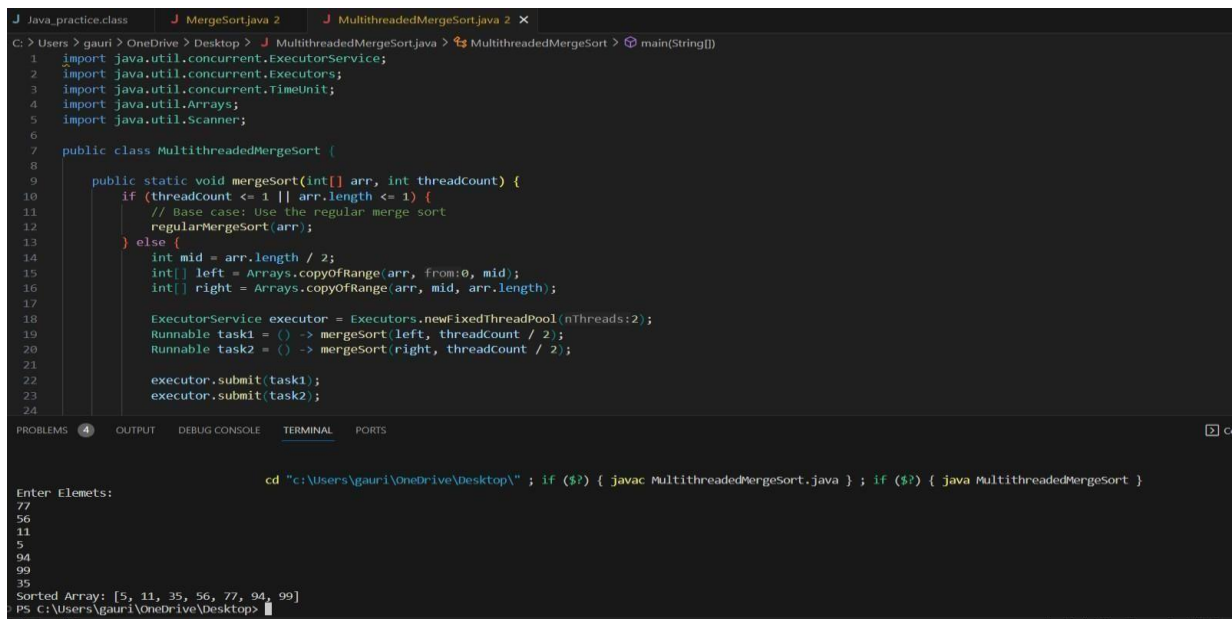
5. Base Case and Recursion:

- The process continues recursively until the subarrays are small enough to be sorted efficiently by a single thread.

Time Complexity:

- Multithreaded Merge Sort has the same time complexity as standard Merge Sort
- Best-case time complexity: $O(n \log n)$
- Average-case time complexity: $O(n \log n)$
- Worst-case time complexity: $O(n \log n)$

Output:



```
J Java_practice.class J MergeSort.java 2 J MultithreadedMergeSort.java 2 X
C:\Users\gaury> OneDrive\OneDrive\Desktop\J MultithreadedMergeSort.java > MultithreadedMergeSort > main(String[])
1 import java.util.concurrent.ExecutorService;
2 import java.util.concurrent.Executors;
3 import java.util.concurrent.TimeUnit;
4 import java.util.Arrays;
5 import java.util.Scanner;
6
7 public class MultithreadedMergeSort {
8
9     public static void mergeSort(int[] arr, int threadCount) {
10         if (threadCount <= 1 || arr.length <= 1) {
11             // Base case: Use the regular merge sort
12             regularMergeSort(arr);
13         } else {
14             int mid = arr.length / 2;
15             int[] left = Arrays.copyOfRange(arr, 0, mid);
16             int[] right = Arrays.copyOfRange(arr, mid, arr.length);
17
18             ExecutorService executor = Executors.newFixedThreadPool(nThreads:2);
19             Runnable task1 = () -> mergeSort(left, threadCount / 2);
20             Runnable task2 = () -> mergeSort(right, threadCount / 2);
21
22             executor.submit(task1);
23             executor.submit(task2);
24
25         }
26     }
27
28     private static void regularMergeSort(int[] arr) {
29         // Regular merge sort implementation
30     }
31 }
32
33 Enter Elements:
34 77
35 56
36 11
37 5
38 94
39 99
40 35
41 Sorted Array: [5, 11, 35, 56, 77, 94, 99]
42 PS C:\Users\gaury\OneDrive\Desktop>
```

Analysing the performance of both the regular Merge Sort and the Multithreaded Merge Sort in the best-case and worst-case scenarios:

Best Case:

The best-case scenario for both regular and multithreaded Merge Sort occurs when the input array is already sorted or nearly sorted. In this case, the performance characteristics are as follows:

-Regular Merge Sort:

- In the best case, regular Merge Sort's time complexity remains $O(n \log n)$.
- The divide-and-conquer nature of the algorithm ensures that it consistently performs well, even when the input is already partially sorted.
- The number of comparisons and swaps is minimal, making it efficient in the best-case scenario.

-Multithreaded Merge Sort:

- The best-case time complexity for multithreaded Merge Sort is also $O(n \log n)$ because the core algorithm remains the same.
- Parallelizing the sorting process doesn't change the fundamental time complexity; it only improves execution time.
- However, the improvement may be less noticeable in the best-case scenario compared to the worst-case scenario, due to the overhead of thread management and synchronization.

Worst Case:

The worst-case scenario for both algorithms occurs when the input array is in reverse order, which results in the maximum number of comparisons and swaps. Here's the analysis for the worst-case scenario:

- Regular Merge Sort:

- In the worst case, regular Merge Sort's time complexity is still $O(n \log n)$.
- While the number of comparisons and swaps is higher in the worst-case scenario, the divide-and-conquer approach ensures that the time complexity remains consistent.
- The algorithm is stable and guarantees the same time complexity, irrespective of the input.

- Multithreaded Merge Sort:

- The worst-case time complexity for multithreaded Merge Sort is also $O(n \log n)$ due to its reliance on the merge sort algorithm.
- Parallelization doesn't affect the worst-case time complexity but can still provide improvements in execution time by taking advantage of multiple cores.
- The overhead associated with thread management and synchronization remains a consideration in the worst-case scenario.

The Difference Between Merge Sort and Multithreaded Merge Sort:

Parameters	Merge Sort	Multithreaded Merge Sort
Parallelism	Single-threaded	Multithreaded
Concurrency	Sequential	Concurrent
Time Complexity	$O(n \log n)$	$O(n \log n)$
Parallelism Overhead	None	Introduces thread management and Synchronization overhead
Scalability	Limited, not effectively scalable with the number of processor cores.	Effectively Scalable with the number of processor mechanism
Complexity	Simpler to implement	Requires handling multiple threads and synchronization mechanism.
Use Cases	Smaller datasets or situations where parallel processing is not a requirement	Large datasets, performance optimization on multi-core processor.

Conclusion:

Both regular and multithreaded Merge Sort algorithms maintain a consistent and reliable time complexity of $O(n \log n)$ in the best-case and worst-case scenarios. Regular Merge Sort is straightforward and efficient, especially in the best-case scenario. Multithreaded Merge Sort can offer performance benefits on multi-core systems, particularly for large datasets and worst-case scenarios, although the improvements might be less noticeable in the best-case scenario. In practical use, the choice between regular and multithreaded Merge Sort should consider factors such as the nature of the input data, hardware capabilities, and the overhead associated with parallelism. For many real-world sorting tasks, regular Merge Sort is often sufficient and simpler to implement, while multithreaded Merge Sort is more suitable for large, unsorted datasets on multi-core systems.