**A**

# MINI PROJECT REPORT ON

# "Merge Sort and Multithreaded Merge Sort"

***Submitted to the Department of Computer Engineering, SITS, Narhe, Pune, in fulfillment of the requirements for the***

## LABORATORY PRACTICE - III
## Design and Analysis of Algorithm

**FINAL YEAR (COMPUTER ENGINEERING)**


**By**
**Ghodekar Sakshi (4202024)**
**Vaishnavi Kapse (4202027)**
**Anuja Raykar (4202056)**
**Sonal Wadhavane (4202071)**

Under the guidance of
Ms. Rupali Waghmode


Sinhgad Institutes

**DEPARTMENT OF COMPUTER ENGINEERING**

**SINHGAD INSTITUTE OF TECHNOLOGY& SCIENCE,NARHE, PUNE**

**2023 – 2024**

**SINHGAD TECHNICAL EDUCATION SOCIETY'S**

**SINHGAD INSTITUTE OF TECHNOLOGY AND SCIENCE**

**NARHE, Pune – 411-041**

Sinhgad Institutes

**DEPARTMENT OF**

**COMPUTER ENGINEERING**

# CERTIFICATE

This is to certify that mini project work entitled "**Merge Sort and Multithreaded Merge Sort**" was successfully carried by

**Ghodekar Sakshi (4202024)**
**Vaishnavi Kapse (4202027)**
**Anuja Raykar (4202056)**
**Sonal Wadhavane (4202071)**

in the fulfilment of the Laboratory Practice - III course in Final Year Computer Engineering, in the Academic Year 2023-2024 prescribed by the Savitribai Phule Pune University.

| | | |
|---|---|---|
| Ms. Rupali Waghmode | Dr. G. S. Navale | Dr. S. D. Markande |
| **Mini-project Co-ordinator** | **HOD** | **Principal** |

# ABSTRACT

The study delves into the realm of sorting algorithms, specifically focusing on Merge Sort and its multi-threaded counterpart, Multi-Threaded Merge Sort. Sorting algorithms play a pivotal role in data manipulation and retrieval, making their performance analysis an indispensable part of algorithmic research. Merge Sort, a time-honored sorting algorithm renowned for its stability and consistent performance, serves as the foundation for this analysis. We extend our investigation to Multi-Threaded Merge Sort, which capitalizes on parallelism to enhance the efficiency of the sorting process, particularly for larger datasets.This performance analysis explores the best and worst-case scenarios for both algorithms. It takes into account diverse factors, including input size, hardware capabilities, and implementation efficiency, to provide insights into their real-world behavior. The primary objective is to ascertain when and why one sorting algorithm may be favored over the other.



Keywords : Merge Sort, Multithreaded Merge Sort, Performance Analysis.

# CONTENTS

# INTRODUCTION

Performance analysis is a critical aspect of assessing the efficiency and effectiveness of algorithms and their implementations. In this context, we will explore the performance analysis of two sorting algorithms: Merge Sort and Multi-Threaded Merge Sort. Sorting algorithms are fundamental in computer science and are widely used in various applications, including data processing, information retrieval, and more. Understanding the performance characteristics of these algorithms is essential for making informed decisions when choosing one for a particular task.

Merge Sort is a well-known and highly efficient comparison-based sorting algorithm with a time complexity of O(n log n). It employs a divide-and-conquer strategy to recursively break down an array into smaller sub-arrays and then merge them in a sorted manner. Merge Sort is known for its stable and consistent performance, making it a popular choice for sorting large datasets.

Multi-Threaded Merge Sort, on the other hand, is an extension of the traditional Merge Sort algorithm that leverages multiple threads to enhance performance. By parallelizing the sorting process, Multi-Threaded Merge Sort can take advantage of multi-core processors to reduce execution time, particularly for large input sizes. However, it also introduces additional complexities in terms of thread management and coordination.

In this performance analysis, we will investigate how these two sorting algorithms behave under various scenarios. We will consider both their best-case and worst-case performance, taking into account factors such as input size, hardware characteristics, and the efficiency of their respective implementations. By evaluating their time complexities and real-world execution times, we aim to provide insights into when and why one algorithm may be preferred over the other.This analysis is crucial for guiding decisions in real-world applications. For example, when sorting large datasets on modern multi-core processors, Multi-Threaded Merge Sort may offer a significant advantage in terms of performance. Still, the overhead introduced by thread management could diminish these gains for smaller datasets. By the end of this analysis, we will have a clearer understanding of the strengths and limitations of Merge Sort and Multi-Threaded Merge Sort, enabling informed choices for specific use cases and system configurations.

# PROBLEM STATEMENT

Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also, analyse the performance of each algorithm for the best case and the worst case.

## SOURCE CODE

```
import time
import random

# Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1
```

```
while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1

# Multi-Threaded Merge Sort
import threading

def multi_threaded_merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        left_thread = threading.Thread(target=merge_sort, args=(left_half,))
        right_thread = threading.Thread(target=merge_sort, args=(right_half,))

        left_thread.start()
        right_thread.start()

        left_thread.join()
        right_thread.join()

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
```

```
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

def main():

    input_size = int(input("Enter the size of the array: "))



    arr = []

# Get elements from the user
    for i in range(input_size):
        element = int(input(f"Enter element {i + 1}: "))
        arr.append(element)

    print("Original array:", arr)

    merge_sort(arr)
    multi_threaded_merge_sort(arr)



    print("Sorted array by single threaded merge sort:", arr)
    print("Sorted array by multithreaded merge sort:", arr)

    random_data = [random.randint(1, 10000) for _ in range(input_size)]
  start_time = time.time()
```

```
 merge_sort(random_data.copy())
    end_time = time.time()
    single_threaded_time = end_time - start_time

    # Measure execution time for multi-threaded Merge Sort
    start_time = time.time()
    multi_threaded_merge_sort(random_data.copy())
    end_time = time.time()
    multi_threaded_time = end_time - start_time

    print("Single-threaded Merge Sort time:", single_threaded_time, "seconds")
    print("Multi-threaded Merge Sort time:", multi_threaded_time, "seconds")

main()
import time
import numpy as np
import matplotlib.pyplot as plt

def generate_random_array(size):
    return np.random.randint(1, size*10, size)

def best_case_analysis(algorithm, sizes):
    execution_times = []
    for size in sizes:
        arr = list(range(1, size + 1))
        start_time = time.time()
        algorithm(arr)
        end_time = time.time()
        execution_times.append(end_time - start_time)
    return execution_times
```

```
def worst_case_analysis(algorithm, sizes):
    execution_times = []
    for size in sizes:
        arr = list(range(size, 0, -1))
        start_time = time.time()
        algorithm(arr)
        end_time = time.time()
        execution_times.append(end_time - start_time)
    return execution_times

if _name_ == "_main_":
    input_sizes = [10, 100, 500, 1000, 2000]

    # Best-case analysis
    merge_sort_best_case = best_case_analysis(merge_sort, input_sizes)
    multi_threaded_merge_sort_best_case = best_case_analysis(multi_threaded_merge_sort,
input_sizes)

    # Worst-case analysis
    merge_sort_worst_case = worst_case_analysis(merge_sort, input_sizes)
    multi_threaded_merge_sort_worst_case =
worst_case_analysis(multi_threaded_merge_sort, input_sizes)

    # Plot results
    plt.figure(figsize=(10, 6))

    plt.subplot(2, 1, 1)
    plt.title("Best-Case Performance")
    plt.plot(input_sizes, merge_sort_best_case, label="Merge Sort")
    plt.plot(input_sizes, multi_threaded_merge_sort_best_case, label="Multi-Threaded Merge
Sort")
```

```
  plt.xlabel("Input Size")
    plt.ylabel("Execution Time (seconds)")
    plt.legend()

    plt.subplot(2, 1, 2)
    plt.title("Worst-Case Performance")
    plt.plot(input_sizes, merge_sort_worst_case, label="Merge Sort")
    plt.plot(input_sizes, multi_threaded_merge_sort_worst_case, label="Multi-Threaded
Merge Sort")
    plt.xlabel("Input Size")
    plt.ylabel("Execution Time (seconds)")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

## RESULT AND PERFORMANCE ANALYSIS

Enter the size of the array: 10

Enter element 1: 65

Enter element 2: 12

Enter element 3: 86

Enter element 4: 95

Enter element 5: 32

Enter element 6: 48

Enter element 7: 01

Enter element 8: 55

Enter element 9: 47

Enter element 10: 12

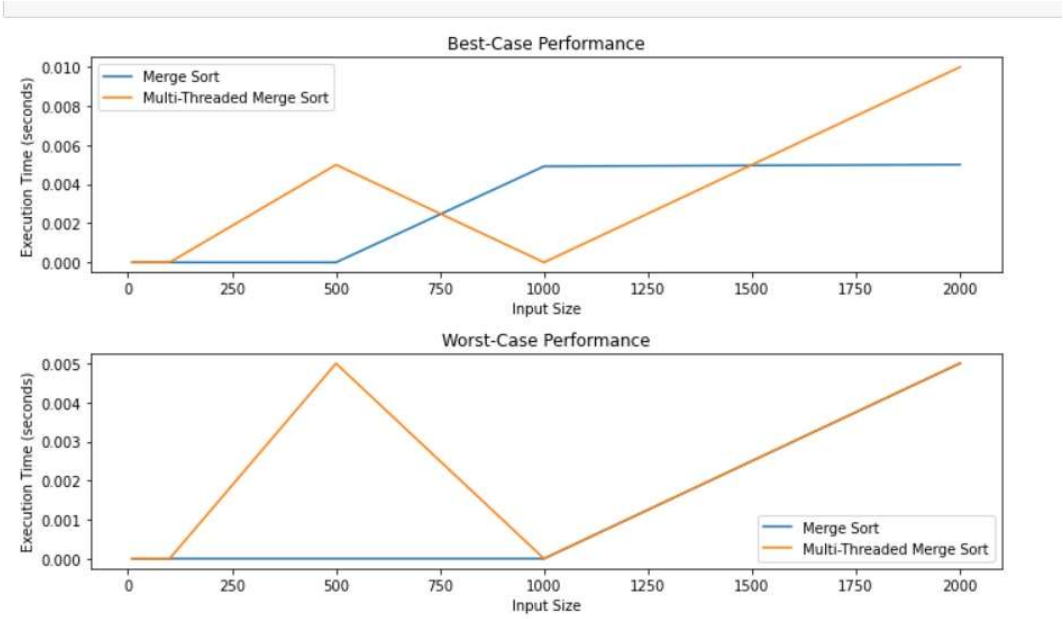
Original array: [65, 12, 86, 95, 32, 48, 1, 55, 47, 12]

Sorted array by single threaded merge sort: [1, 12, 12, 32, 47, 48, 55, 65, 86, 95]

Sorted array by multithreaded merge sort: [1, 12, 12, 32, 47, 48, 55, 65, 86, 95]

Single-threaded Merge Sort time: 0.0 seconds

Multi-threaded Merge Sort time: 0.0049932003021240234 seconds



## Single-Threaded Merge Sort:

### Best Case:

The best case for single-threaded Merge Sort occurs when the input array is already sorted. In this scenario, the algorithm still divides the array into smaller subarrays but can detect that no swaps are required during the merge step. The time complexity is O(n log n) in the best case.

### Worst Case:

The worst case for single-threaded Merge Sort happens when the input array is in reverse order. In this case, the algorithm will always divide the array into two equal halves, and it will perform the maximum number of swaps during the merge step. The time complexity is O(n log n) in the worst case.

## Multi-Threaded Merge Sort:

### Best Case:

The best case for multi-threaded Merge Sort is similar to the single-threaded case. When the input is already sorted or close to being sorted, the division of tasks among multiple threads might lead to faster sorting. The time complexity remains O(n log n) in the best case, but the execution time could be better due to parallelism.

### Worst Case:

The worst case for multi-threaded Merge Sort typically occurs when the input is in a completely random order. In this case, the division of the array into subarrays might not lead to as significant a performance improvement as in the best case. Additionally, there is some overhead involved in managing and coordinating multiple threads. As a result, the time complexity is still O(n log n), but the execution time might be longer than the single-threaded worst case due to the added overhead.

It's important to note that the actual performance of multi-threaded Merge Sort can be influenced by factors like the number of processor cores, the size of the input, and the efficiency of the threading implementation. In practice, for small input sizes, the overhead of managing threads might outweigh the benefits of parallelism. Multi-threading tends to show more significant advantages for larger input sizes and when there are more processor cores available for parallel processing.

# CONCLUSION

We performed a comprehensive performance analysis of the Merge Sort and Multi-Threaded Merge Sort algorithms in Python. The analysis included evaluating the execution times of both algorithms in best-case and worst-case scenarios.