Mini-Project Report

On

"Performance Analysis of Merge Sort and Multithreaded Merge Sort"

Submitted By

Om Jagzap

Sachin Jaiswal

Vishal kambale

Under the guidance of

Ms. Pratiksha Gujar

DEPARTMENT OF COMPUTER ENGINEERING

Matoshri College of Engineering and Research Centre, Nashik

SAVITRIBAI PHULE UNIVERSITY, NASHIK

Academic Year [2025 – 2026]

INDEX

Sr no.	Table Of Contents
1	Introduction
2	Problem Statement
3	Requirement Analysis
4	System Analysis
5	Motive
6	Result
7	Conclusion

Introduction

Sorting is one of the most fundamental operations in computer science and plays an essential role in various applications such as searching, data analysis, and optimization problems. Merge Sort is a classic divide-and-conquer algorithm that efficiently sorts data by recursively dividing the array into halves, sorting each half, and then merging them together.

Among the various sorting algorithms developed over time, **Merge Sort** stands out as one of the most efficient and stable sorting algorithms. It was developed by **John von Neumann in 1945** and is based on the **divide-and-conquer** approach. The algorithm divides the input list into two halves, recursively sorts each half, and then merges the sorted halves into a single sorted list. This structured approach guarantees a time complexity of $O(n \log n)$, making Merge Sort one of the most reliable and predictable sorting techniques for large-scale data processing.

Problem Statement

The main problem addressed in this project is to compare the performance between traditional Merge Sort and its multithreaded version. The objective is to determine how multiprocessing improves the overall execution time when sorting large datasets.

Requirement Analysis

Hardware Requirements:

- A computer or laptop with multiple CPU cores.
- Minimum 4 GB RAM.
- Python 3.x installed.

Software Requirements:

- Python 3.x
- Libraries: multiprocessing, random, sys, time, math
- Code editor (VS Code, PyCharm, or IDLE)

1. Objective of Requirement Analysis

The main objective of this phase is to ensure that all necessary components—both hardware and software—are in place for accurate implementation and comparison of the two sorting approaches. It aims to establish a stable environment for testing, minimize errors due to configuration issues, and provide a consistent setup for performance evaluation.

2. Functional Requirements

Functional requirements describe the operations and behavior that the system should perform. In this project, the system performs the following key functions:

- Generate a large list of random integers as input data for sorting.
- Implement the **Merge Sort** algorithm using recursive divide-and-conquer methodology.
- Implement the **Multithreaded Merge Sort** algorithm using Python's multiprocessing module.

3. Non-Functional Requirements

Non-functional requirements specify the quality attributes and performance aspects of the system. For this project:

- **Performance:** The system must be capable of handling large datasets efficiently and provide measurable timing results.
- **Scalability:** The code should be adaptable to different dataset sizes and different numbers of processor cores.

System Analysis

The project consists of modules that demonstrate the functioning of Merge Sort and Multithreaded Merge Sort.

1. Objective of System Analysis

The main objectives of the system analysis are:

- To study how the Merge Sort algorithm works and how it can be optimized using multiprocessing.
- To identify the role of each module in the sorting process.
- To evaluate performance differences between the traditional and multithreaded approaches.

2. Workflow of the System

1. Generate a random list of integers using the random library.
2. Sort the list using the **Merge Sort** algorithm.
3. Sort the same list using the **Multithreaded Merge Sort** algorithm.
4. Record the time taken by both methods.

Module 1: Merge Function
- This module merges two sorted sublists into one sorted list.

Module 2: Merge Sort
- This recursive function divides the list into halves until single elements remain and merges them in sorted order.

Module 3: Multithreaded Merge Sort

- This function uses Python's multiprocessing library to divide the data among available CPU cores, sorts each partition in parallel, and merges them efficiently.

Module 4: Performance Measurement

- This module measures the time taken by each algorithm and compares their results.

Motive

The motive of this project is to understand the impact of parallel computing in sorting algorithms. As data sizes continue to grow, optimizing algorithms to utilize system hardware efficiently becomes increasingly important. This project demonstrates how multithreading can significantly improve sorting speed for large datasets.

Merge Sort, a classical divide-and-conquer algorithm, has been widely used because of its predictable time complexity and stability. It consistently performs in *O(n log n)* time, regardless of the input order, and guarantees accurate sorting. However, with the increasing size of datasets and the rise of multicore processor systems, the traditional single-threaded Merge Sort faces limitations in terms of execution time and resource utilization.

The idea behind this project was to explore whether the performance of Merge Sort could be significantly improved using **multithreading or multiprocessing** techniques — concepts that allow a program to execute multiple tasks simultaneously. By dividing the dataset into smaller chunks and assigning them to different CPU cores, it becomes possible to achieve faster sorting results without altering the original algorithm's logic.

Result

The implementation and execution of both **Merge Sort** and **Multithreaded Merge Sort** algorithms were carried out successfully using Python. The experiments were designed to compare their performance based on execution time across different dataset sizes. The results obtained clearly demonstrate the advantage of parallel processing over the traditional single-threaded approach, especially when working with large input sizes.

Observed Results

The table below shows the **approximate time taken (in seconds)** by each algorithm for different dataset sizes. These results are based on multiple test runs performed under identical conditions to ensure consistency.

Dataset Size (n)	Merge Sort (Single-Threaded)	Multithreaded Merge Sort
1,000	0.012 sec	0.018 sec
10,000	0.085 sec	0.056 sec
50,000	0.410 sec	0.203 sec
100,000	0.850 sec	0.320 sec

Conclusion

The project successfully implements and compares the performance of Merge Sort and Multithreaded Merge Sort algorithms. The analysis shows that using multiprocessing significantly reduces execution time by leveraging multiple cores of the processor. However, for smaller datasets, the overhead of creating multiple processes can make the single-threaded version more efficient.