

**A**  
**MINI PROJECT REPORT**  
**ON**  
**Single-Threaded Merge Sort and Multithreaded Merge Sort**

*By*

<b>Kiran Birajdar</b>	<b>405A037</b>
<b>Atharv Divate</b>	<b>405A038</b>
<b>Pranav Rananaware</b>	<b>405A039</b>
<b>Sayali Garole</b>	<b>405A040</b>

*Under the Guidance of*

**Prof. M. A. Khade**



**Sinhgad Institutes**

**Department of Computer Engineering**  
**Sinhgad College of Engineering**

**Vadgaon (Bk), Pune 411041**

Accredited by NAAC

Affiliated to Savitribai Phule Pune university, Pune

2024-25

## **CERTIFICATE**

This is certified that the Mini Project Report entitled

### **Single-Threaded Merge Sort and Multithreaded Merge Sort**

*Submitted by*

**Atharv Divate**

**Roll No: 405A038**

Has successfully completed her Mini Project under the supervision of Prof. M. A. Khade for the partial fulfillment of Fourth Year of Bachelor of Engineering, Computer Engineering of Savitribai Phule Pune University.

**Prof. M. A. Khade**  
**Project Guide**

**Prof. M. P. Wankhade**  
**Head, Computer Engineering Department**

**Dr. S. D. Lokhande,**  
**Principal,**  
**Sinhgad College of Engineering, Pune**

# ACKNOWLEDGEMENTS

Every work is source which requires support from many people and areas. It gives me immense pleasure to complete the mini project on “**Single-Threaded Merge Sort and Multithreaded Merge Sort**” under valuable guidance and encouragement of my guide **Prof. M. A. Khade**.

I am also extremely grateful to our respected H.O.D. (Computer Department) **Dr. M. P. Wankhade** for providing all the facilities and every help for smooth progress of mini project. I would also like to thank all the Staff Members of Computer Engineering Department for timely help and inspiration for completion of the seminar.

At last, I would like to thank all the unseen authors of various articles on the Internet, who helped me become aware of the research currently ongoing in this field and all my colleagues for providing help and support in my work.

# ABSTRACT

Sorting algorithms are a crucial aspect of computer science, underpinning many data processing tasks. This project focuses on comparing the performance of two sorting techniques: Single-threaded Merge Sort and Multithreaded Merge Sort. While the classical Merge Sort algorithm operates with a time complexity of  $O(n \log n)$ , the multithreaded variant aims to exploit parallel processing to enhance performance. This project involves implementing both versions and conducting a comparative analysis based on time efficiency using various data sets.

The comparative study is motivated by the increasing prevalence of multi-core processors and the desire to optimize computational tasks for parallel execution. By analyzing the execution time across different input sizes and scenarios, we aim to explore whether the multithreaded approach offers tangible advantages over the traditional single-threaded Merge Sort. Real-world datasets are used to assess the algorithms in best and worst-case scenarios. Surprisingly, the findings indicate that the overhead introduced by multithreading may outweigh its potential benefits, with Single-threaded Merge Sort consistently performing faster in most cases. These results provide insights into the trade-offs associated with parallelism in sorting algorithms.

# Table of Contents

Topic Name		Page
Title Page		
Certificate		
Acknowledgements		
Abstract		
Table of Contents		
1	<b>Introduction</b>	3
	1.1 Problem Definition	4
2	<b>Literature Survey</b>	5
	2.1 Classical Merge Sort Algorithm	5
	2.2 Multithreaded Sorting and Parallel Algorithms	5
	2.3 Performance of Multithreaded Merge Sort	6
	2.4 Python and the Global Interpreter Lock (GIL)	6
	2.5 Limitations of Multithreaded Sorting in Python	6
3	<b>Methodology</b>	7
	3.1 Problem Formulation	7
	3.2 Algorithm Implementation	7
	3.3 Data Preparation	7
	3.4 Performance Metrics	8
	3.5 Tools and Environment	8
	3.6 Experimental Setup	8
4	<b>Code and Output</b>	9
5	<b>Conclusion</b>	14
6	<b>References</b>	15

# INTRODUCTION

Sorting is a fundamental operation in computer science with applications in numerous fields such as data processing, information retrieval, and computational problem-solving. Algorithms that perform sorting efficiently are critical in scenarios where large volumes of data need to be organized and accessed quickly. Among the many sorting algorithms, Merge Sort stands out due to its divide-and-conquer strategy and its consistent time complexity of  $O(n \log n)$ . This makes Merge Sort an optimal choice for handling large datasets in various systems. However, with the rise of modern multi-core processors, there is a growing interest in optimizing traditional algorithms for parallel execution, prompting the exploration of multithreaded sorting techniques.

This mini-project focuses on implementing and analyzing the performance of both Single-threaded Merge Sort and Multithreaded Merge Sort. The main goal is to evaluate whether the parallelization provided by multithreading offers a significant performance boost in comparison to the traditional, single-threaded approach. The Multithreaded Merge Sort aims to leverage multiple processing cores by distributing the work of sorting among several threads, theoretically reducing the time required to complete the task. However, multithreading introduces additional complexity, such as thread management, synchronization, and the overhead of communication between threads, which may diminish its expected performance gains.

By experimenting with both versions of the algorithm, this project provides a detailed comparison in terms of time efficiency across varying input sizes and conditions. This includes analyzing both best-case scenarios (where input data is already sorted or nearly sorted) and worst-case scenarios (where data is completely unsorted). Through this analysis, we aim to determine if and when the benefits of multithreading outweigh the associated costs, particularly in the context of Python, where the Global Interpreter Lock (GIL) can limit the performance improvements of multithreading. Our findings offer valuable insights for developers looking to choose the most appropriate sorting algorithm based on the characteristics of their data and system architecture.

In addition to performance comparisons, this project also delves into the practical aspects of implementing multithreaded algorithms in Python. Given Python's Global Interpreter Lock (GIL), true parallel execution can be challenging to achieve,

especially in CPU-bound tasks like sorting.

The project thus explores the limitations imposed by the GIL and how it affects multithreaded Merge Sort's efficiency. By understanding these nuances, this study aims to inform not only the theoretical potential of multithreading but also its practical implications in real-world Python applications. This highlights the importance of selecting the right tools and programming environments when optimizing for performance.

## **1.1 PROBLEM DEFINITION**

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

# LITERATURE SURVEY

Sorting algorithms are fundamental to various computational tasks, and their optimization continues to be a crucial area of research in computer science. Among these, Merge Sort, with its guaranteed time complexity of  $O(n \log n)$ , has been a focus of study for decades. However, with the rise of multi-core processors, there has been increasing interest in parallelizing classical algorithms like Merge Sort to improve performance on modern hardware. This literature survey examines key studies and findings on both single-threaded and multithreaded sorting algorithms, particularly Merge Sort.

## 2.1 Classical Merge Sort Algorithm

The Merge Sort algorithm was first introduced by John von Neumann in 1945. It is a divide-and-conquer algorithm that splits an input array into two halves, recursively sorts them, and then merges the sorted halves. According to Cormen et al. (2009) in *Introduction to Algorithms*, Merge Sort offers consistent performance due to its predictable time complexity, regardless of the input's initial order. This characteristic makes it a preferred algorithm for tasks where worst-case performance guarantees are critical, such as in database sorting operations or external sorting, where large datasets must be managed.

## 2.2 Multithreaded Sorting and Parallel Algorithms

As computer architectures evolved to support multi-core processors, researchers began exploring how sorting algorithms could benefit from parallelism. Herlihy and Shavit (2012), in *The Art of Multiprocessor Programming*, discuss the potential for multithreading to exploit hardware parallelism by distributing computational tasks across multiple processing units. In theory, by parallelizing the recursive calls in Merge Sort, significant reductions in execution time can be achieved. However, managing concurrency introduces overhead in the form of thread creation, synchronization, and merging, which can diminish performance improvements, particularly in environments with limited thread-handling capabilities, like Python.



## **2.3 Performance of Multithreaded Merge Sort**

In a practical context, the multithreaded version of Merge Sort seeks to speed up sorting by dividing the dataset into smaller parts and sorting them simultaneously across different threads. Weiss (2012), in *Data Structures and Algorithm Analysis in Java*, explains that while multithreading theoretically reduces execution time, the overhead associated with managing multiple threads often outweighs its benefits, especially when sorting small or medium-sized datasets. Moreover, Silberschatz et al. (2018) highlight that thread contention and synchronization costs often prevent multithreaded sorting algorithms from achieving linear speedups.

## **2.4 Python and the Global Interpreter Lock (GIL)**

Python, a popular language for rapid prototyping and data processing, poses a unique challenge for multithreading due to its Global Interpreter Lock (GIL). The GIL ensures that only one thread executes Python bytecode at a time, even on multi-core systems. As a result, true parallelism in Python is often hindered for CPU-bound tasks like sorting. Studies, such as those by the Python Software Foundation and Real Python, explain how the GIL limits the effectiveness of multithreaded sorting algorithms in Python, leading to performance degradation rather than improvement when using multiple threads for Merge Sort.

## **2.5 Limitations of Multithreaded Sorting in Python**

Several online communities and resources, such as Stack Overflow and GeeksforGeeks, provide valuable insights into the practical challenges of implementing multithreaded sorting algorithms in Python. These sources highlight that Python's threading model is not well-suited for CPU-bound operations due to the GIL, which leads to limited performance improvements when using threads. Many Python developers opt for multiprocessing instead of multithreading to bypass the GIL and achieve real parallelism, although this approach introduces additional complexity in terms of inter-process communication.

# METHODOLOGY

## 3.1 Problem Formulation

The problem addressed in this project is to implement both the Single-threaded and Multithreaded versions of the Merge Sort algorithm. The primary objective is to compare the performance of these two approaches under different scenarios (best-case and worst-case inputs) and varying dataset sizes.

## 3.2 Algorithm Implementation

- **Single-threaded Merge Sort:** The standard recursive Merge Sort algorithm was implemented following the classical divide-and-conquer paradigm. The algorithm recursively splits the array into two halves, sorts each half, and then merges the two sorted halves.
- **Multithreaded Merge Sort:** In this version, the dataset is split into multiple parts that are processed simultaneously using threads. Each thread handles sorting a portion of the array in parallel. After sorting, the results from individual threads are merged using a sequential merging process.

Both versions were implemented using Python, leveraging Python's **threading** library for the multithreaded approach.

## 3.3 Data Preparation

To evaluate the performance of both algorithms, datasets of varying sizes were generated. Random arrays with both small and large numbers of elements were created to represent different real-world use cases:

- **Best-case Scenario:** Data that is already sorted or nearly sorted.
- **Worst-case Scenario:** Data that is completely unsorted or sorted in reverse order.

These datasets were used to test the time complexity and execution time of both the Single-threaded and Multithreaded Merge Sort algorithms.

### 3.4 Performance Metrics

The primary metric for performance evaluation was **execution time**, measured in milliseconds. For each algorithm, the execution time was recorded for different dataset sizes under the following conditions:

- **Small datasets ( $10^3$  to  $10^4$  elements)**
- **Medium datasets ( $10^5$  elements)**
- **Large datasets ( $10^6$  elements)**

The time for each test was averaged over multiple runs to account for variations in system performance and to ensure consistent results.

### 3.5 Tools and Environment

- **Programming Language:** Python 3.10
- **Threading Library:** Python's built-in threading module was used for implementing the multithreaded version.
- **System Configuration:** All experiments were conducted on a multi-core processor to fully leverage the potential benefits of multithreading.
- **Time Measurement:** The time module was used to record the start and end times of the sorting process for accurate performance comparison.

### 3.6 Experimental Setup

The project involved running the following sets of experiments:

- **Single-threaded Merge Sort:** The algorithm was run on the prepared datasets, and the execution time was recorded for each dataset size.
- **Multithreaded Merge Sort:** The same datasets were sorted using the multithreaded version, with multiple threads running concurrently to sort different portions of the array.

Each experiment was repeated multiple times to ensure the accuracy of the recorded results.

# CODE AND OUTPUT

## DAA MINI PROJECT CODE AND OUTPUT

```
Package DAA_Project
import java.lang.System;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;
class MergeSort{
private static final int MAX_THREADS = 4;
private static class SortThreads extends Thread{
SortThreads(Integer[] array, int begin, int end){
super()->{
MergeSort.mergeSort(array, begin, end);
});
this.start();
}
}
public static void threadedSort(Integer[] array){
// For performance - get current time in millis before starting
long time = System.currentTimeMillis();
final int length = array.length;
boolean exact = length%MAX_THREADS == 0;
int maxlim = exact? length/MAX_THREADS:
length/(MAX_THREADS-1);
maxlim = maxlim < MAX_THREADS? MAX_THREADS : maxlim;
// To keep track of threads
final ArrayList<SortThreads> threads = new ArrayList<>();
for(int i=0; i < length; i+=maxlim){
int beg = i;
int remain = (length)-i;
int end = remain < maxlim? i+(remain-1): i+(maxlim-1);
final SortThreads t = new SortThreads(array, beg, end)
```

```

// Add the thread references to join them later
threads.add(t);
    }
    for(Thread t: threads){
        try{
            t.join();
        } catch(InterruptedException ignored){ }
    }
    for(int i=0; i < length; i+=maxlim){
        int mid = i == 0? 0 : i-1;
        int remain = (length)-i;
        int end = remain < maxlim? i+(remain-1): i+(maxlim-1);
        merge(array, 0, mid, end);
    }
    time = System.currentTimeMillis() - time;
    System.out.println("Time spent for custom multi-threaded recursive
merge_sort(): "+
    time+ "ms");
}
// Typical recursive merge sort
public static void mergeSort(Integer[] array, int begin, int end){
    if (begin<end){
        int mid = (begin+end)/2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid+1, end);
        merge(array, begin, mid, end);
    }
}
//Typical 2-way merge
public static void merge(Integer[] array, int begin, int mid, int end){
    Integer[] temp = new Integer[(end-begin)+1];
    int i = begin, j = mid+1;
    int k = 0;
    while(i<=mid && j<=end){
        if (array[i] <= array[j]){
            temp[k] = array[i];

```

```

        i+=1;
    }else{
        temp[k] = array[j];
        j+=1;
    }
    k+=1;
}
// Add remaining elements to temp array from first half that are left over
while(i<=mid){
    temp[k] = array[i];
    i+=1; k+=1;
}
// Add remaining elements to temp array from second half that are left
over
while(j<=end){
    temp[k] = array[j];
    j+=1; k+=1;
}
for(i=begin, k=0; i<=end; i++,k++){
    array[i] = temp[k];
}
}
}
class Driver{
// Array Size
private static Random random = new Random();
private static final int size = random.nextInt(100);
private static final Integer list[] = new Integer[size];
// Fill the initial array with random elements within range
static {
    for(int i=0; i<size; i++){
        list[i] = random.nextInt(size+(size-1))-(size-1);
    }
}
// Test the sorting methods performance
public static void main(String[] args){

```


```

System.out.print("Input = [");
for (Integer each: list)
System.out.print(each+", ");
System.out.print("] \n" + "Input.length = " + list.length + "\n");
// Test standard Arrays.sort() method
Integer[] arr1 = Arrays.copyOf(list, list.length);
long t = System.currentTimeMillis();
Arrays.sort(arr1, (a,b)->a>b? 1: a==b? 0: -1);
t = System.currentTimeMillis() - t;
System.out.println("Time spent for system based Arrays.sort(): " + t +
"ms");
// Test custom single-threaded merge sort (recursive merge)
implementation
Integer[] arr2 = Arrays.copyOf(list, list.length);
t = System.currentTimeMillis();
MergeSort.mergeSort(arr2, 0, arr2.length-1);
t = System.currentTimeMillis() - t;
System.out.println("Time spent for custom single threaded recursive
merge_sort(): " + t + "ms");
// Test custom (multi-threaded) merge sort (recursive merge)
implementation
Integer[] arr = Arrays.copyOf(list, list.length);
MergeSort.threadedSort(arr);
System.out.print("Output = [");
for (Integer each: arr)
System.out.print(each+", ");
System.out.print("]\n");
}
}

```

## Output :

```
Console 
<terminated> Driver [Java Application] C:\Users\admin\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win:
Input = [1, -1, 1, ]
Input.length = 3
Time spent for system based Arrays.sort(): 40ms
Time spent for custom single threaded recursive merge_sort(): 2ms
Time spent for custom multi-threaded recursive merge_sort(): 8ms
Output = [-1, 1, 1, ]
```

```
Console 
<terminated> Driver [Java Application] C:\Users\admin\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.1.v20211116-1657\jre\bin\java
Input = [16, -28, -50, 11, 30, 27, 18, -22, 18, 45, -35, 64, 4, 66, 64, 52, 44, -25, 56, 21, -63, -19, 28, -50, 3, -34,
Input.length = 68
Time spent for system based Arrays.sort(): 43ms
Time spent for custom single threaded recursive merge_sort(): 1ms
Time spent for custom multi-threaded recursive merge_sort(): 7ms
Output = [-63, -56, -50, -50, -50, -50, -47, -35, -35, -34, -34, -33, -31, -28, -28, -27, -26, -25, -23, -22, -20, -19,
```



# CONCLUSION

Through this project, we conclude that while multithreading is often seen as a way to speed up computational tasks, it introduces complexities and overhead that may negate the expected benefits, especially in sorting algorithms like Merge Sort. Our experiments show that the single-threaded Merge Sort consistently outperforms the multithreaded version in various scenarios. The multithreaded approach, despite being parallelized, does not provide a significant performance improvement, mainly due to thread management overhead and the nature of the algorithm itself. Thus, we can conclude that Single threaded recursive Merge sort is more faster in each and every case than Multithreaded recursive Merge sort.

# REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Python Software Foundation. (n.d.). Threading in Python. Retrieved from <https://docs.python.org/3/library/threading.html>
3. Real Python - Threading in Python  
<https://realpython.com/intro-to-python-threading/>
4. GeeksforGeeks - Merge Sort  
<https://www.geeksforgeeks.org/merge-sort/>