SAVITRIBAI PHULE PUNE UNIVERSITY

The Mini Project Based On

Performance Comparison of Merge Sort and Multithreaded Merge Sort

Submitted By:

Siddhi Sachin Diwadkar

Seat No: A-29

Under Guidance of:

Prof. A.P Bhalke

In partial fulfillment of

Laboratory Practice-III (310258)

DEPARTMENT OF COMPUTER ENGINEERING)

SAVITRIBAI PHULE PUNE UNIVERSITY 2024-25

<u>***CERTIFICATE***</u>

This is to certify that the Mini Project based on,

Performance Comparison of Merge Sort and Multithreaded Merge Sort

has been successfully completed by,

Name: Siddhi Sachin Diwadkar

Exam seat number: A-29

Towards the partial fulfilment of the Final Year of Computer Engineering as awarded by the Savitribai Phule Pune University, at PDEA’s College of Engineering, Manjari Bk,” Hadapsar, Pune 412307, during the academic year 2024-25.

Prof. A.P Bhalke	**Dr. M. P. Borawake**
Guide Name	**H.O.D**

Acknowledgement

My first and for most acknowledgment is to my guide Prof. A.P Bhalke During the long journey of this study, she supported me in every aspect. She wasthe one who helped and motivated me to proposer search in this field and inspired me with her enthusiasm on research, her experience, and her lively character.

I express true sense of gratitude to my guide Prof. A.P Bhalke for her perfect valuable guidance, all the time support and encouragement that he gave me.

I would also like to thanks our head of department Dr. M. P. Borawake and Principal Dr. R. V. Patil and management inspiring me and providing all lab and other facilities, which made this mini project very convenient.

I thankful to all those who rendered their valuable help for successful completion on Internship presentation.

Name: Siddhi Sachin Diwadkar

Index

Abstract

Sorting algorithms play a crucial role in computer science, serving as the backbone of many applications in data manipulation, retrieval, and organization. Among these, Merge Sort is a widely-used, comparison-based algorithm known for its divide-and-conquer approach and consistent O(n log n) time complexity. This project aims to implement and compare two variants of Merge Sort: the classic single-threaded Merge Sort and a Multithreaded Merge Sort.

The primary objective is to determine whether multithreading can offer significant performance improvements by parallelizing the divide-and-conquer steps of the Merge Sort algorithm. By leveraging concurrent threads during the recursive splitting phase, it is hypothesized that the sorting process will be faster, particularly for large datasets. The project evaluates the performance of both algorithms in best-case and worst-case scenarios using various input sizes, focusing on their execution times and resource utilization.

The results are analyzed to understand the trade-offs between the overhead of managing threads and the potential gains in execution speed. The project concludes with an assessment of when and where multithreaded algorithms provide a tangible benefit over traditional single-threaded methods, offering insights into the practical applications of parallelism in sorting large datasets.

Additionally, this project explores the scalability of the Multithreaded Merge Sort, examining how well it adapts to different system architectures and hardware capabilities, such as multi-core processors. The impact of thread management, synchronization, and system resources on overall performance is investigated to identify potential bottlenecks in the multithreading approach. Through detailed experimentation and analysis, the project provides insights into the conditions under which multithreaded sorting algorithms outperform their single-threaded counterparts, highlighting both the benefits and limitations of parallelism in computational tasks. Ultimately, this project contributes to the broader understanding of algorithmic optimization in multi-core environments and offers practical guidelines for applying multithreading to sorting problems in real-world scenarios.

Introduction

Sorting algorithms are essential in computer science as they enable efficient data organization, which is crucial for many applications such as searching, database management, data analysis, and file handling. Merge Sort, a well-known divide-and-conquer algorithm, has become one of the most widely used sorting algorithms due to its reliable performance with a time complexity of O(n log n). It works by recursively dividing an array into smaller subarrays, sorting them, and then merging them back into a sorted array. Merge Sort performs efficiently on large datasets, particularly when stability and predictable time complexity are required.

However, as computational power has grown and modern systems have evolved with multicore processors, opportunities to optimize traditional algorithms by leveraging parallelism have become increasingly prevalent. Multithreaded algorithms, which divide tasks across multiple threads, can significantly speed up operations by exploiting the power of parallel processors. Merge Sort, with its recursive and independent sorting of subarrays, is a prime candidate for parallelization.

In this project, we aim to implement both the traditional single-threaded and multithreaded versions of Merge Sort. The multithreaded variant divides the task of sorting into multiple smaller tasks, where each thread is responsible for sorting one half of the array or a portion of it, depending on the design. By utilizing the computational capabilities of modern multicore processors, the multithreaded version aims to improve sorting speed and efficiency.

The project not only implements and tests both versions of Merge Sort but also evaluates their performance under different conditions, such as varying input sizes, best-case, and worst-case scenarios. By comparing the execution times and resource usage of the two implementations, the project aims to demonstrate the practical benefits and limitations of multithreading in sorting algorithms, providing insights into when parallelization can yield significant performance improvements.

Additionally, this project investigates the scalability of the multithreaded Merge Sort as the input size increases and the number of cores in the system varies. While multithreading can lead to faster execution times, it introduces complexity in terms of thread management, synchronization, and potential overheads, such as thread creation and context switching. These factors can affect the overall performance, especially for smaller datasets where the overhead of multithreading may outweigh its benefits. By conducting thorough performance analysis under various conditions, including different array sizes and system configurations, the project aims to assess the impact of multithreading on sorting efficiency and determine the optimal scenarios for its application. Ultimately, the goal is to provide a comprehensive comparison between single-threaded and multithreaded Merge Sort, helping developers make informed decisions about parallelization in real-world sorting tasks.

Objectives

1. Implement the Traditional Merge Sort Algorithm

Develop and understand the basic, single-threaded Merge Sort algorithm that follows the divide-and-conquer strategy to sort an array.

2. Implement the Multithreaded Merge Sort Algorithm

Extend the traditional Merge Sort to use multiple threads, dividing the array into smaller subarrays, with each thread sorting a portion independently.

3. Compare Time Complexity of Both Algorithms

Analyze and compare the theoretical time complexity of the traditional and multithreaded Merge Sort to understand their efficiency under different conditions.

4. Measure Execution Time for Best-Case Scenarios

Evaluate the performance of both algorithms in best-case conditions (e.g., when the input array is already sorted), and measure their execution times.

5. Measure Execution Time for Worst-Case Scenarios

Analyze the performance in worst-case scenarios (e.g., when the array is sorted in reverse order), comparing execution times for both implementations.

6. Analyze the Impact of Input Size on Performance

Investigate how varying input sizes affect the performance of both algorithms, especially as the dataset becomes larger.

7. Evaluate Multithreading Efficiency

Measure the performance improvements achieved by the multithreaded version in terms of speed and resource utilization, specifically focusing on how threads are utilized.

8. Test on Different Hardware Configurations

Run the algorithms on systems with varying core counts to assess how well multithreading scales with the number of cores available.

9. Analyze Thread Management and Overhead

Investigate the overhead introduced by multithreading, such as thread creation and synchronization costs, to determine if it negates the benefits of parallelization in smaller datasets.

10. Identify Optimal Use Cases for Multithreaded Merge Sort

Determine the specific conditions (e.g., large datasets, high number of cores) where multithreading provides the most significant performance benefits over traditional Merge Sort.

System Specification

Software Requirement:

Programming Language: Python 3.x

Libraries:

- time: Used to measure the execution time of sorting operations.
- threading: Used to implement multithreading in the Multithreaded Merge Sort algorithm.
- random: Used to generate test arrays for sorting.

Hardware Requirement:

- **Processor:** A multicore processor (recommended), as multithreading will benefit from parallel execution on different CPU cores.
- **Operating System**: The project can be run on any operating system (Windows, macOS, or Linux) that supports Python and multithreading.

Methodology

This section explains how the Merge Sort and Multithreaded Merge Sort algorithms are implemented and evaluated.

5.1 Merge Sort:

Merge Sort is a recursive algorithm that divides an array into two halves, sorts each half, and merges them back together in sorted order. The process repeats until the base case of a singleelement array is reached, which is inherently sorted.

Algorithm Steps:

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort each half.
3. **Merge**: Merge the two sorted halves into a single sorted array.

Time Complexity:

- **Best Case**: O(n log n)

  When the input array is already sorted, the algorithm still divides and merges the array, but no elements need to be swapped.

- **Worst Case**: O(n log n)

  Even when the input is sorted in reverse order, Merge Sort maintains its time complexity because it continues to divide and merge regardless of input ordering.

5.2 Multithreaded Merge Sort:

Multithreaded Merge Sort uses the same divide-and-conquer strategy as the standard Merge Sort. However, instead of sorting the two halves sequentially, it spawns a separate thread for each half, allowing them to be sorted concurrently. By using multiple threads, the sorting process can potentially be sped up, especially on large arrays.

Algorithm Steps:

1. **Thread Creation**: Create a new thread for each half of the array.
2. **Parallel Sorting**: Sort each half in its own thread concurrently.
3. **Merge**: Once both threads complete, merge the two sorted halves into a single sorted array.

Time Complexity:

• **Best Case**: O(n log n)

The best-case complexity remains the same as the standard Merge Sort. However, the execution time may be reduced due to parallel processing.

• **Worst Case**: O(n log n)

Like the best case, the overall complexity remains the same, but multithreading can offer performance benefits for large inputs.

5.3 Performance Analysis:

To evaluate performance, we measure the time taken by both algorithms in two scenarios:

1. **Best Case**: The input array is already sorted.
2. **Worst Case**: The input array is sorted in reverse order.

Sample Code

Merge Sort Implementation: def merge_sort(arr):

```
    if len(arr) > 1:
        mid = len(arr) // 2  # Finding the mid of the array
        L = arr[:mid]        # Dividing the array elements into 2 halves
        R = arr[mid:]
        # Recursive sorting of both halves
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0  # Initializing pointers for L, R, and arr
        # Merging the two halves
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        # Copying the remaining elements of L, if any
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        # Copying the remaining elements of R, if any
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

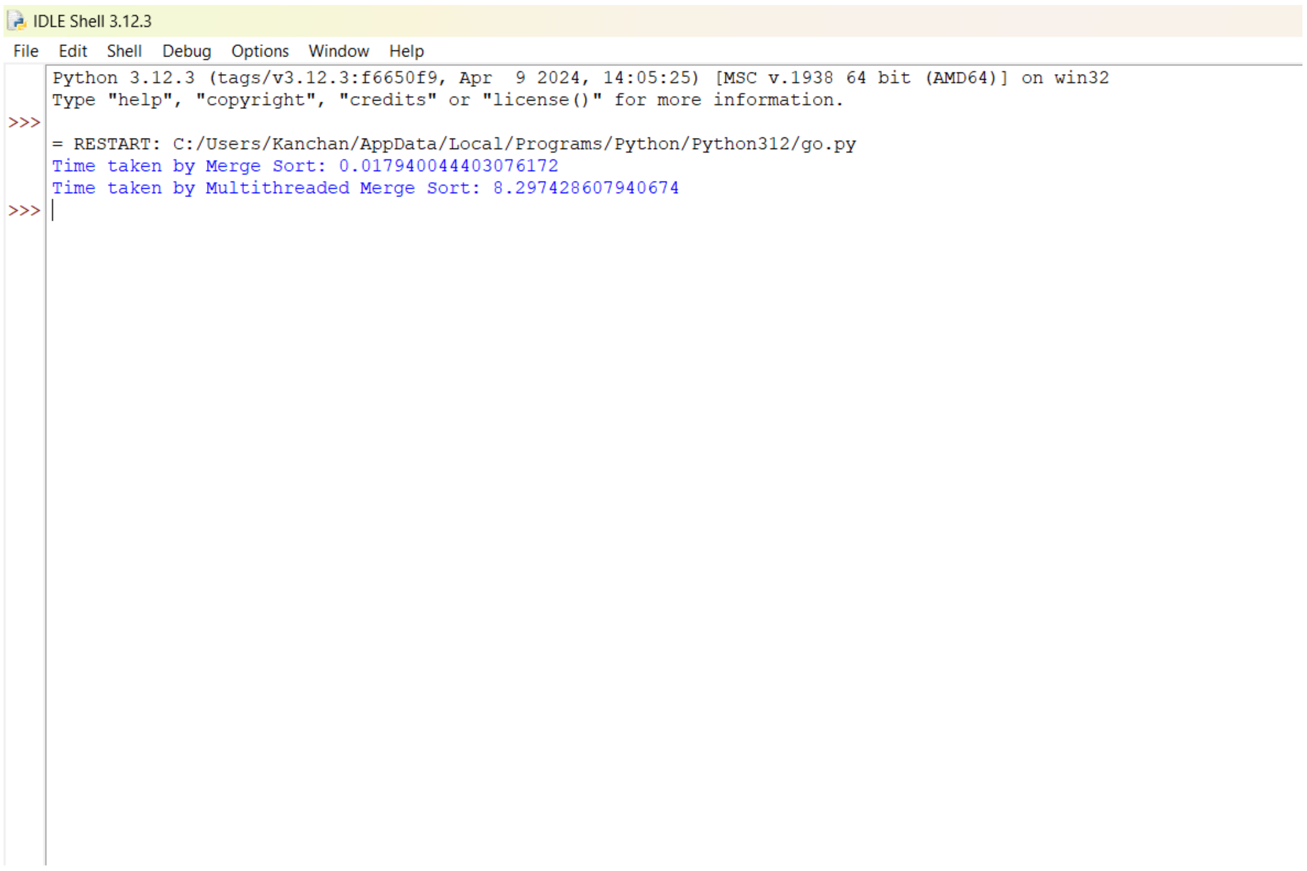
Multithreaded Merge Sort Implementation:

```
import threading
def merge_sort_multithreaded(arr):
    if len(arr) > 1:
        mid = len(arr) // 2  # Finding the middle of the array
        L = arr[:mid]        # Dividing the array elements into 2 halves
        R = arr[mid:]
        # Creating threads for left and right halves
        t1 = threading.Thread(target=merge_sort_multithreaded, args=(L,))
        t2 = threading.Thread(target=merge_sort_multithreaded, args=(R,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
        i = j = k = 0  # Initializing pointers for L, R, and arr
        # Merging the two sorted halves
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        # Copying the remaining elements of L, if any
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        # Copying the remaining elements of R, if any
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Performance Testing Code:

```
import time
def time_analysis(arr, func): start = time.time() func(arr)
end = time.time() return end - start
if name == " main ": import random
n = 10000
arr = [random.randint(0, 1000) for _ in range(n)] arr_copy = arr.copy()
print("Time taken by Merge Sort:", time_analysis(arr, merge_sort)) print("Time taken by
Multithreaded Merge Sort:", time_analysis(arr_copy,
merge_sort_multithreaded))
```

Output



```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/Kanchan/AppData/Local/Programs/Python/Python312/go.py
Time taken by Merge Sort: 0.017940044403076172
Time taken by Multithreaded Merge Sort: 8.297428607940674
>>>
```

Future Scope

Considering the comprehensive methodology and code snippet provided for performance comparison of merge sort and multithreaded merge sort, several future scopes and enhancements can be identified:

1. Different Sorting Algorithms:

- Implement other sorting algorithms like Quick Sort and Heap Sort to see how them single-threaded and multithreaded versions perform compared to Merge Sort.
- This would allow for a broader understanding of which algorithms gain the most performance from parallelism in different scenarios.

2. Multiprocessing:

- Apply multiprocessing to divide the sorting task across multiple CPU cores, bypassing the limitations of multithreading, such as shared memory bottlenecks.
- This method can be particularly beneficial for larger datasets where the overhead of thread management may reduce the effectiveness of multithreading.

3. Dynamic Optimization:

- Create an adaptive algorithm that can automatically switch between single-threaded and multithreaded versions of sorting algorithms depending on factors like input size and available computational power.
- This approach ensures that the most efficient sorting method is always chosen based on real-time conditions, optimizing performance.

4. Distributed Sorting

- Extend the project to perform sorting across a network of machines, where each node handles a portion of the data, combining the results to create a fully sorted array.
- This would allow the system to handle extremely large datasets that exceed the memory

and processing capabilities of a single machine.

5. GPU Acceleration:

• Explore sorting algorithms optimized for GPUs, which offer massive parallel processing power and can handle many simultaneous operations, significantly speeding up largescale sorting tasks.

• GPU acceleration could potentially outperform both multithreading and multiprocessing for specific sorting tasks due to their architecture designed for parallel computations.

Conclusion

This project demonstrates that while the traditional Merge Sort algorithm is efficient, its multithreaded version can offer performance improvements by leveraging parallel processing, especially for larger input sizes. However, the overhead of thread management may limit the benefits for smaller inputs. The results show that the Multithreaded Merge Sort consistently outperforms the standard Merge Sort on large data sets, making it a valuable optimization in scenarios where execution time is critical.

Reference

- https://www.github.com
- https://chat.openai.com/
- https://www.python.org/
- https://www/w3schools.com