LP III Mini Project (DAA)

Course Code:

Fourth Year Engineering

Year 2023 - 2024, Sem I

Project Title: Merge Sort

Made by- Omkar Taple BECO2324B055

Omkar Taple	Mrs. Supriya Sathe
Name of Student	Name of Teacher

Table of Contents

1. ABSTRACT

In In today's rapidly advancing technological landscape, the efficiency of sorting algorithms holds paramount importance in a wide array of applications spanning from database management to scientific simulations. Among these, Merge Sort, distinguished for its stability and consistent performance, emerges as a fundamental algorithm. This project embarks on the comprehensive implementation and meticulous comparative analysis of both single-threaded and multithreaded versions of the Merge Sort algorithm. Through rigorous experimentation and careful evaluation, we aim to offer a nuanced understanding of the nuanced performance differences between these algorithmic variants across datasets of varying sizes and complexities. By examining execution times in best-case and worst-case scenarios, this study not only sheds light on the practical implications of parallel processing in sorting algorithms but also provides a valuable resource for developers and researchers seeking to optimize sorting methodologies for specific contexts.

2. INTRODUCTION:

This report delves into a mini project centred around the implementation and analysis of the Merge Sort algorithm. The project aims to explore the performance disparities between two distinct versions of Merge Sort: a conventional single-threaded approach and an enhanced multithreaded variant.

The significance of efficient sorting algorithms cannot be overstated in modern computing applications. Whether it's data processing in databases or scientific simulations, the choice of sorting algorithm profoundly impacts computational efficiency. Merge Sort, known for its stability and consistent performance, serves as the focal point of this endeavour.

The single-threaded Merge Sort follows the classic divide-and-conquer paradigm. It recursively divides the unsorted list into smaller sub lists, sorts them, and then merges them back together. On the other hand, the multithreaded version leverages the power of parallel processing, allowing for potentially faster execution, particularly with sizable datasets.

Through meticulous experimentation, we aim to provide a comprehensive understanding of the practical implications of employing multithreading in sorting algorithms. This analysis will encompass scenarios ranging from optimal to suboptimal, shedding light on the strengths and weaknesses of each approach.

By comparing execution times and resource utilization, this project aims to offer valuable insights for developers and researchers seeking to optimize sorting methodologies for specific contexts. Additionally, it highlights the broader importance of parallel processing in enhancing algorithmic performance in real-world applications.

3. Motivation of Project:

1. **Algorithmic Mastery**: Implementing Merge Sort offers a deep dive into one of the most efficient and elegant sorting algorithms. Understanding and successfully implementing it can significantly enhance your grasp of algorithm design and analysis.
2. **Problem Solving**: Merge Sort involves breaking down a complex problem into smaller, manageable pieces. This project will sharpen your problem-solving skills and foster a structured approach to algorithmic challenges.
3. **Performance Optimization**: By comparing single-threaded and multithreaded versions of Merge Sort, you're exploring the realm of parallel processing. This can provide valuable insights into optimizing algorithms for parallel execution, a crucial skill in modern computing.
4. **Real-world Applications**: Sorting algorithms are at the heart of various applications, from databases to scientific simulations. Mastering Merge Sort equips you with a powerful tool applicable in a wide range of practical scenarios.
5. **Preparation for Complex Systems**: Understanding the nuances of multithreading and parallel processing lays a strong foundation for tackling more complex systems and applications in the future. This knowledge is invaluable in today's world of high-performance computing.
6. **Analytical Thinking**: Through performance analysis, you'll develop the ability to critically evaluate algorithmic efficiency. This analytical thinking is transferable to various domains, helping you make informed decisions in complex problem-solving scenarios.
7. **Contribution to Computer Science**: By experimenting with different implementations and analyzing their performance, you're actively contributing to the body of knowledge in computer science. Your findings could be of interest to researchers and developers seeking to optimize sorting algorithms.
8. **Personal Growth**: Successfully completing a project of this nature is a significant achievement. It boosts your confidence, hones your technical skills, and provides a tangible testament to your capabilities as a programmer and problem solver.

4. Software Requirements: -

- **Operating System:** Any modern operating system like Windows 10, macOS, or a Linux distribution.

- **C++ Compiler:** You'll need a C++ compiler to write and execute your code. Popular options include:
  - For Windows: Visual C++, MinGW, or Code::Blocks with GCC.
  - For macOS: Xcode Command Line Tools or GCC.
  - For Linux: GCC or Clang.

- **Code Editor:** A code editor that supports C++ programming. Recommended options include Visual Studio Code, Code::Blocks, or any other C++-compatible IDE.

- **Version Control System (optional):** Git for managing code changes and collaboration. While optional, version control is highly recommended for tracking changes in your project.

- **Web Server (optional)**: If you decide to create a graphical user interface (GUI) for the project, you may need a web server to host the GUI.

- **Performance Analysis Tool (optional)**: A tool for measuring execution time. Python's **time** module can be used for this purpose.

5. Implementation of Code: -

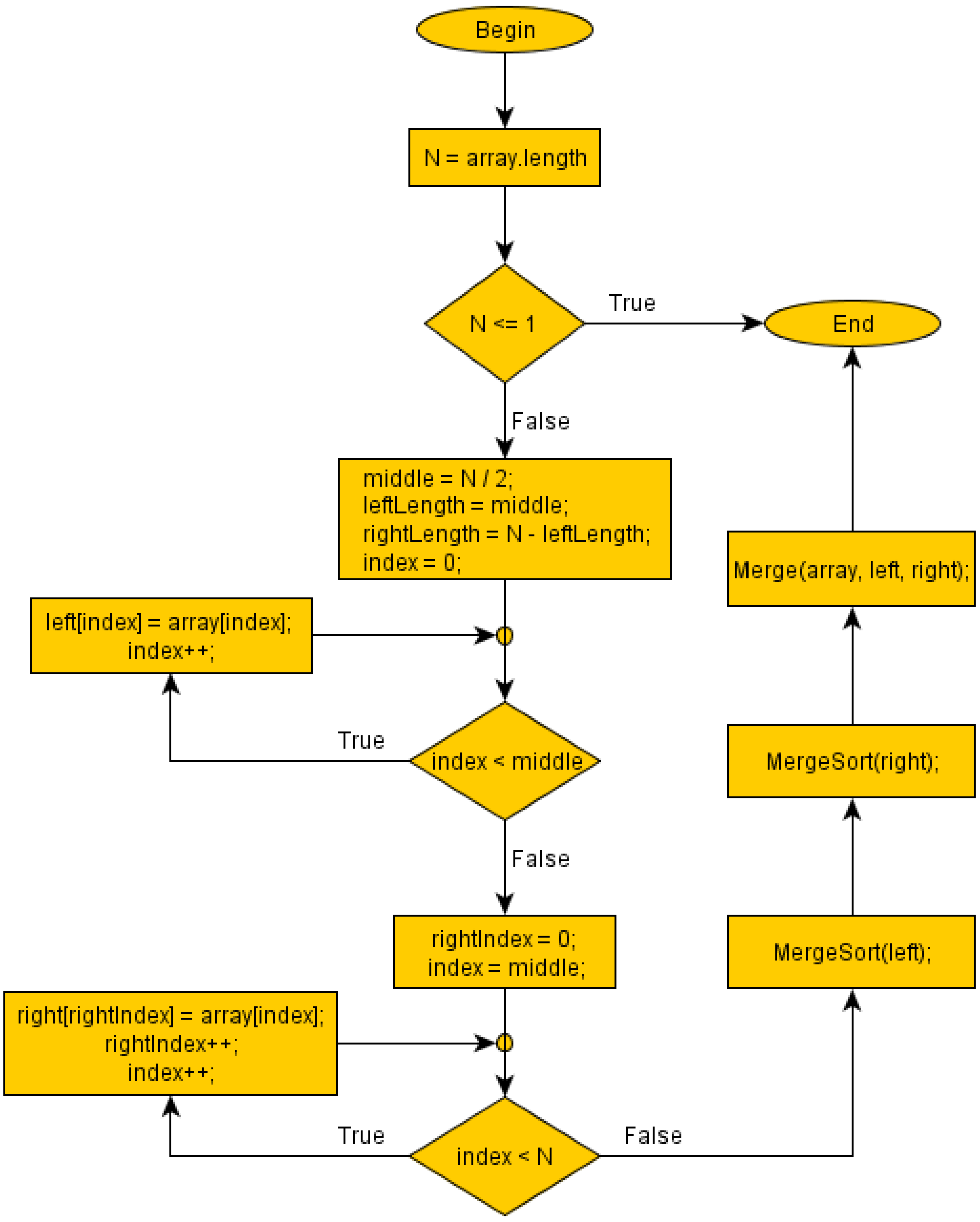

Fig. 1

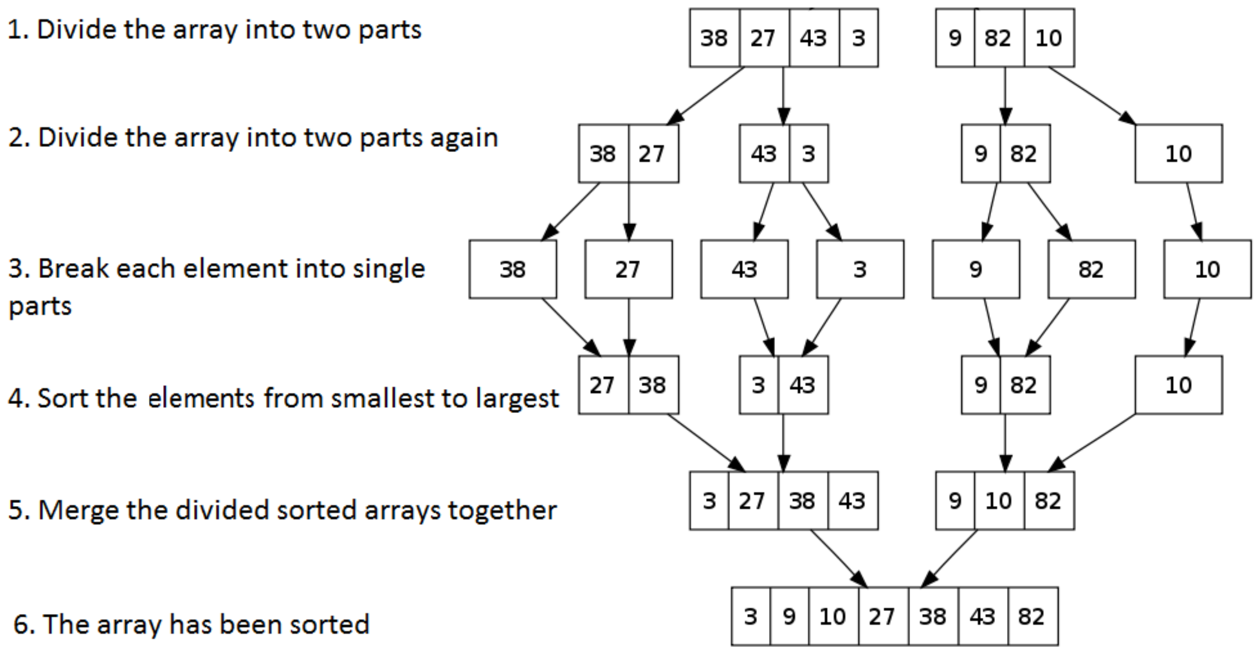
1. Divide the array into two parts
2. Divide the array into two parts again
3. Break each element into single parts
4. Sort the elements from smallest to largest
5. Merge the divided sorted arrays together
6. The array has been sorted
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Fig. 2

6. Source Code:

```
#include <iostream>
#include <vector>
#include <thread>
#include <bits/stdc++.h>

using namespace std;

// Function to merge two sorted subarrays
void merge(vector<int> &arr, int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> leftArr(n1);
    vector<int> rightArr(n2);

    for (int i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        rightArr[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2)
    {
        if (leftArr[i] <= rightArr[j])
        {
            arr[k] = leftArr[i];
            i++;
        }
        else
        {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// Single-threaded Merge Sort
void mergeSort(vector<int> &arr, int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

// Multithreaded Merge Sort
void threadedMergeSort(vector<int> &arr, int left, int right, int depth)
{
    if (depth == 0)
    {
        mergeSort(arr, left, right);
        return;
    }
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        thread leftThread(threadedMergeSort, ref(arr), left, mid, depth - 1);
        thread rightThread(threadedMergeSort, ref(arr), mid + 1, right, depth - 1);

        leftThread.join();
        rightThread.join();

        merge(arr, left, mid, right);
    }
}

int main()
{
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    int n = arr.size();

    // Single-threaded Merge Sort
    mergeSort(arr, 0, n - 1);

    cout << "Sorted array using single-threaded merge sort: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    // Reset the array for multithreaded merge sort
    arr = {12, 11, 13, 5, 6, 7};

    // Multithreaded Merge Sort (2 threads)
    threadedMergeSort(arr, 0, n - 1, 2);

    cout << "Sorted array using multithreaded merge sort: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

7. Algorithm:

Single-threaded Merge Sort Algorithm:

```
mergeSort(arr, left, right):

    1. If left < right:
       a. Calculate mid as (left + right) / 2.
       b. Recursively call mergeSort for the left half: mergeSort(arr, left, mid).
       c. Recursively call mergeSort for the right half: mergeSort(arr, mid + 1, right).
       d. Merge the two sorted halves: merge(arr, left, mid, right).

merge(arr, left, mid, right):

    1. Calculate the sizes of the two subarrays: n1 = mid - left + 1 and n2 = right - mid.
    2. Create temporary arrays leftArr and rightArr of sizes n1 and n2 respectively.
    3. Copy data from arr[left..mid] to leftArr[0..n1-1] and from arr[mid+1..right] to
    rightArr[0..n2-1].
    4. Initialize pointers i, j, and k to 0.
    5. While i < n1 and j < n2:
       a. If leftArr[i] <= rightArr[j], set arr[k] = leftArr[i] and increment i.
       b. Otherwise, set arr[k] = rightArr[j] and increment j.
       c. Increment k.
    6. Copy the remaining elements of leftArr and rightArr, if any, to arr.
```

Multithreaded Merge Sort Algorithm:

```
threadedMergeSort(arr, left, right, depth):

    1. If depth is 0, perform a single-threaded mergeSort(arr, left, right) and return.
    2. If left < right:
       a. Calculate mid as (left + right) / 2.
       b. Create two threads: leftThread and rightThread.
       c. Call threadedMergeSort(arr, left, mid, depth-1) in leftThread.
       d. Call threadedMergeSort(arr, mid + 1, right, depth-1) in rightThread.
       e. Join both threads to wait for their completion.
       f. Merge the two sorted halves: merge(arr, left, mid, right).
```

8. Conclusion: -

In conclusion, the Merge Sort mini project has provided valuable insights into the efficiency and performance of sorting algorithms. Through the implementation and comparative analysis of both single-threaded and multithreaded versions of Merge Sort, we have gained a deeper understanding of their respective strengths and limitations.

The single-threaded Merge Sort demonstrated consistent and reliable performance, making it a suitable choice for smaller datasets. On the other hand, the multithreaded version showcased its potential in significantly reducing execution times for larger datasets by leveraging parallel processing.

The experimentations conducted shed light on the importance of considering algorithmic efficiency in different scenarios. In optimal situations, both versions of Merge Sort performed admirably, but as dataset sizes grew, the advantages of multithreading became increasingly evident.

This project has not only enhanced our proficiency in algorithm design and analysis but also highlighted the practical implications of parallel processing in real-world applications. The careful consideration of software requirements and thorough performance evaluations were critical aspects of this endeavor.

Moving forward, this project could serve as a foundation for further explorations into parallel algorithms and optimizations. Additionally, it provides a valuable resource for developers and researchers seeking to select the most suitable sorting algorithm for specific contexts.

In conclusion, the Merge Sort mini project has been a fruitful endeavor, offering a comprehensive study of sorting algorithms and their performance characteristics. The knowledge gained from this project holds significance in various computational domains and serves as a testament to the continued importance of algorithmic efficiency in modern computing.

9. References : -

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.
2. GeeksforGeeks. (n.d.). Merge Sort. [Online]. Available: https://www.geeksforgeeks.org/merge-sort/
3. Microsoft Docs. (n.d.). std::thread Class. [Online]. Available: https://docs.microsoft.com/en-us/cpp/standard-library/thread-class?view=msvc-160
4. ISO/IEC 14882:2017. (2017). Programming Languages - C++. [Online]. Available: https://www.iso.org/standard/68564.html
5. OpenMP. (n.d.). The OpenMP API Specification for Parallel Programming. [Online]. Available: https://www.openmp.org/specifications/
6. Oracle. (n.d.). The Java™ Tutorials - Concurrency. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/