

A Project Report On

LP3 – Machine Learning

CLASS: BE-1

GUIDED BY

Prof. Shweta Shah



DEPARTMENT OF COMPUTER ENGINEERING

PUNE INSTITUTE OF COMPUTER TECHNOLOGY

DHANKAWADI, PUNE-43

SAVITRIBAI PHULE PUNE UNIVERSITY

2021-22

Gayatri Godbole – 41128

Pushkar Jain – 41134

Title:

Merge Sort by multithreading

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.

Hardware Requirements:

4gb Ram

64 bit OS

Software Requirements:

VS Code

C++

Windows/Ubuntu

Theory:

In computer science, merge sort (also commonly spelled as mergesort) is an efficient, general-purpose, and comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and von Neumann as early as 1948. **Threads** are lightweight processes and threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space. **Multi-threading** is way to improve parallelism by running the threads simultaneously in different cores of your processor. In this program, we'll use 4 threads but you may change it according to the number of cores your processor has.

Merge Sort

Merge sort is a sorting technique which is based on divide and conquer technique where we divide the array into equal halves and then combine them in a sorted manner.

Algorithm to implement merge sort is

check if there is one element in the list then return the element.

Else, Divide the data recursively into two halves until it can't be divided further.

Finally, merge the smaller lists into new lists in sorted order.

In merge sort, the problem is divided into two subproblems in every iteration.

Hence efficiency is increased drastically.

It follows the divide and conquer approach

Divide break the problem into 2 subproblem which continues until the problem set is left with one element only

Conquer basically merges the 2 sorted arrays into the original array

Multi-Threading

In the operating system, **Threads** are the lightweight process which is responsible for executing the part of a task. Threads share common resources to execute the task concurrently.

Multi-threading is an implementation of multitasking where we can run multiple threads on a single processor to execute the tasks concurrently. It subdivides specific operations within a single application into individual threads. Each of the threads can run in parallel.

Merge sort is a good design for multi-threaded sorting because it allocates sub-arrays during the merge procedure thereby avoiding data collisions. This implementation breaks the array up into separate ranges and then runs its algorithm on each of them, but the data must be merged (sorted) in the end by the main thread. The more threads there are, the more unsorted the second to last array is thereby causing the final merge to take longer!!

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid = (left+right)/2

mergesort(array, left, mid)

```
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

Multi-threaded Merge sort

Multi-threading is way to improve parallelism by running the threads simultaneously in different cores of your processor. In this program, we'll use 4 threads but you may change it according to the number of cores your processor has.

For Example-:

In –int arr[] = {3, 2, 1, 10, 8, 5, 7, 9, 4}

Out –Sorted array is: 1, 2, 3, 4, 5, 7, 8, 9, 10

Explanation –we are given an unsorted array with integer values. Now we will sort the array using merge sort with multithreading.

In –int arr[] = {5, 3, 1, 45, 32, 21, 50}

Out –Sorted array is: 1, 3, 5, 21, 32, 45, 50

Explanation –we are given an unsorted array with integer values. Now we will sort the array using merge sort with multithreading.

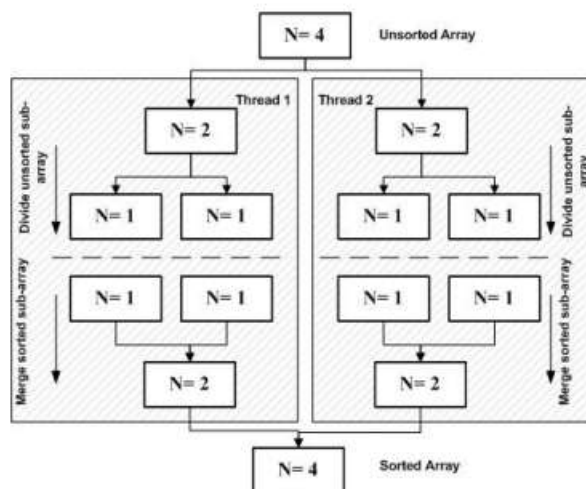


Fig. . Multithread Merge Sort

CODE :

Merge Sort

```
#include <iostream>
using namespace std;
void merge(int array[], int const left, int const mid,
           int const right)
```

```

{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;
    auto *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array
    while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]
            <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]

```

```

        = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArray(int A[], int size)
{
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}

int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    auto arr_size = sizeof(arr) / sizeof(arr[0]);

```

```

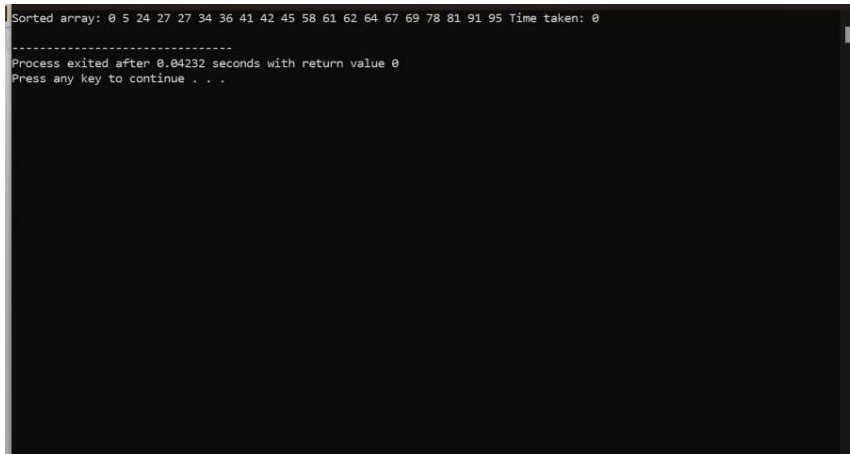
cout << "Given array is \n";
printArray(arr, arr_size);

mergeSort(arr, 0, arr_size - 1);

cout << "\nSorted array is \n";
printArray(arr, arr_size);
return 0;    }

```

OUTPUT :



```

Sorted array: 0 5 24 27 27 34 36 41 42 45 58 61 62 64 67 69 78 81 91 95 Time taken: 0
-----
Process exited after 0.04232 seconds with return value 0
Press any key to continue . . .

```

Multi-threaded Merge Sort

```

#include <iostream>
#include <pthread.h>
#include <time.h>
#define MAX 20
#define THREAD_MAX 4
using namespace std;
int a[MAX];
int part = 0;
void merge(int low, int mid, int high)
{
    int* left = new int[mid - low + 1];
    int* right = new int[high - mid];
    int n1 = mid - low + 1, n2 = high - mid, i, j;

```



```

        for (i = 0; i < n1; i++)
            left[i] = a[i + low];
    for (i = 0; i < n2; i++)
        right[i] = a[i + mid + 1];
    int k = low;
    i = j = 0;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j])
            a[k++] = left[i++];
        else
            a[k++] = right[j++];
    }
    while (i < n1) {
        a[k++] = left[i++];
    }
    while (j < n2) {
        a[k++] = right[j++];
    }
}

void merge_sort(int low, int high)
{
    int mid = low + (high - low) / 2;
    if (low < high) {
        merge_sort(low, mid);
        merge_sort(mid + 1, high);
        merge(low, mid, high);
    }
}

void* merge_sort(void* arg)
{
    int thread_part = part++;
    int low = thread_part * (MAX / 4);
    int high = (thread_part + 1) * (MAX / 4) - 1;

```

```

int mid = low + (high - low) / 2;
if (low < high) {
    merge_sort(low, mid);
    merge_sort(mid + 1, high);
    merge(low, mid, high);
}
}

int main()
{
    for (int i = 0; i < MAX; i++)
        a[i] = rand() % 100;
    clock_t t1, t2;

    t1 = clock();
    pthread_t threads[THREAD_MAX];
    for (int i = 0; i < THREAD_MAX; i++)
        pthread_create(&threads[i], NULL, merge_sort,
                                                                (void*)NULL);

    for (int i = 0; i < 4; i++)
        pthread_join(threads[i], NULL);
    merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);
    merge(MAX / 2, MAX/2 + (MAX-1-MAX/2)/2, MAX - 1);
    merge(0, (MAX - 1)/2, MAX - 1);
    t2 = clock();
    cout << "Sorted array: ";
    for (int i = 0; i < MAX; i++)
        cout << a[i] << " ";
    cout << "Time taken: " << (t2 - t1) /
        (double)CLOCKS_PER_SEC << endl;

    return 0;
}

```

OUTPUT :

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
-----
Process exited after 0.03339 seconds with return value 0
Press any key to continue . . .
```

Time Complexity and Performance

Merge Sort

1. Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n \cdot \log n)$

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **$O(n \cdot \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **$O(n \cdot \log n)$** .
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **$O(n \cdot \log n)$** .

2. Space Complexity

Space Complexity	$O(n)$
Stable	YES

- The space complexity of merge sort is $O(n)$. It is because, in merge sort, an extra variable is required for swapping.

Multi-threaded Merge Sort

Multithread merge sort, creates thread recursively, and stops work when it reaches a certain size, with each thread locally sorting its data. Then threads merge their data by joining threads into a sorted main list. The multithread merge sort that have array of 4 elements to be sorted. Merge sort in multithread is based on the fact that the recursive calls run in parallel, so there is only one $n/2$ term with the time complexity (2): $T(n) = \Theta \log(n) + \Theta(n) = \Theta(n)$

Conclusion

Thus, We have implemented and compared time complexity and analysed performance of the Merge Sort and Multi-threaded Merge Sort.

References

- <https://www.tutorialspoint.com/merge-sort-using-multithreading-in-cplusplus>
- <https://www.geeksforgeeks.org/merge-sort-using-multi-threading/>
- <https://www.geeksforgeeks.org/merge-sort/>
- https://en.wikipedia.org/wiki/Merge_sort
- <https://www.javatpoint.com/merge-sort>