**K. K. Wagh Institute of Engineering Education and Research, Nashik.**
**Department of Computer Engineering**
**Academic Year 2022-23**

**Course:** Laboratory Practice III **Course Code:** 410246
**Class:** BE **Div.:** B
**Name of Students:** Rina Sunil Gholap (05)
Deepti Depak Dabal (06)
Atharva Kalidas Pandharikar (07)
Harshada Tukaram Sonawane (08)

**Name of Faculty:** Prof. K.P.Birla

---

# Assignment No: 17 (Mini Project)

## ( Group A: Design and Analysis of Algorithms )

**Title of Mini-Project:**

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.

**Problem Statement:**

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.

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

**Objective:**

Learn how to implement algorithms that follow algorithm design strategy divide and conquer.

**Description:**

**1. Merge Sort**

Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It

divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

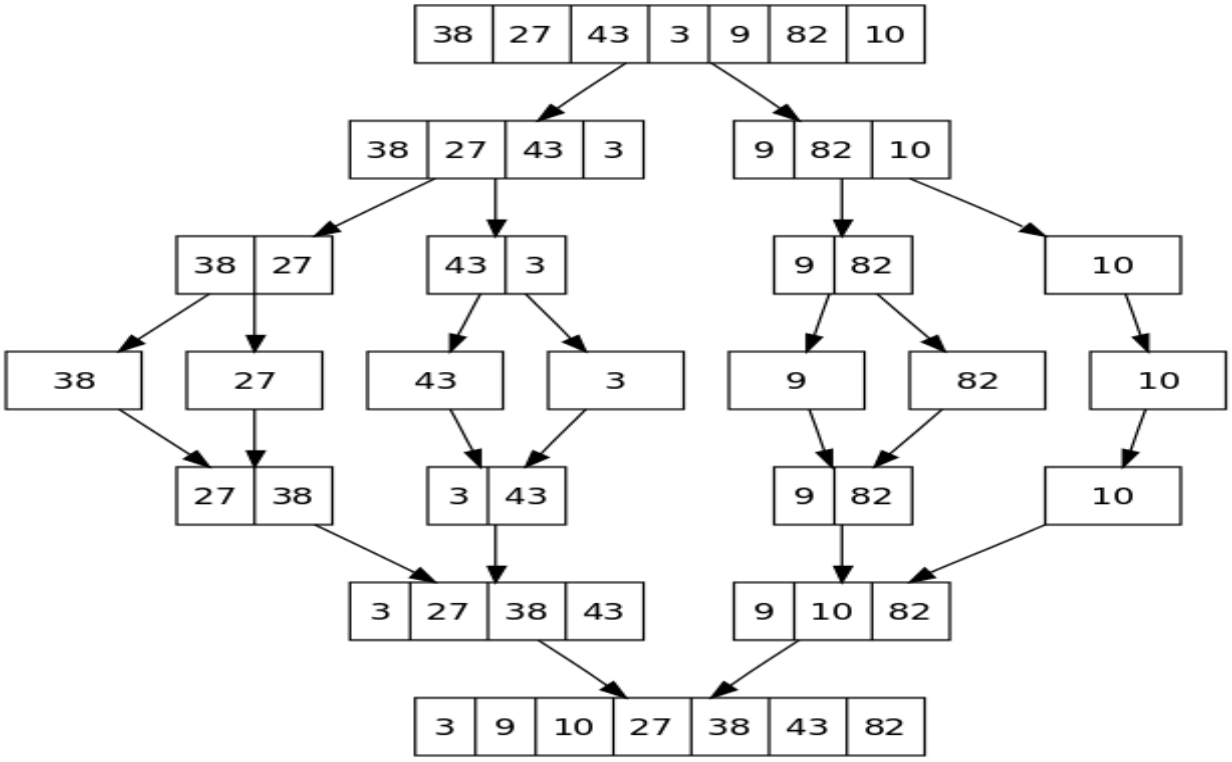


Fig: Merge Sort

**Algorithm of merge sort:**

**Step 1** – Start

**Step 2** − If it is only one element in the list it is already sorted, return.

**Step 3** − Divide the list recursively into two halves until it can no more be divided.

**Step 4** − Merge the smaller lists into new list in sorted order.

**Step 5** − Stop

**Program:**

```
// Merge sort in C++

#include <iostream>
using namespace std;

// Merge two subarrays L and M into arr
```

```
void merge(int arr[], int p, int q, int r)
{
	// Create L ? A[p..q] and M ? A[q+1..r]
	int n1 = q - p + 1;
	int n2 = r - q;

	int L[n1], M[n2];

	for (int i = 0; i < n1; i++)
	L[i] = arr[p + i];
	for (int j = 0; j < n2; j++)
	M[j] = arr[q + 1 + j];

	// Maintain current index of sub-arrays and main array
	int i, j, k;
	i = 0; j = 0; k = p;

	// Until we reach either end of either L or M, pick larger among
	// elements L and M and place them in the correct position at A[p..r]
	while (i < n1 && j < n2)
	{
		if (L[i] <= M[j])
		{
			arr[k] = L[i];
			i++;
		}
		else
		{
			arr[k] = M[j];
			j++;
		}
		k++;
	}

	// When we run out of elements in either L or M,
	// pick up the remaining elements and put in A[p..r]
	while (i < n1)
	{
		arr[k] = L[i];
		i++;	k++;
	}

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

```
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r)
{
	if (l < r)
	{   // m is the point where the array is divided into two subarrays
		int m = l + (r - l) / 2;

		mergeSort(arr, l, m);
		mergeSort(arr, m + 1, r);

		// Merge the sorted subarrays
		merge(arr, l, m, r);
	}
}

// Print the array
void printArray(int arr[], int size)
{
	for (int i = 0; i < size; i++)
	cout << arr[i] << " ";
	cout << endl;
}

// Driver program
int main()
{
	int n;
	cout<<"Enter number of Elements: ";
	cin>>n;
	int arr[n];
	cout<<"Enter Array Elements: ";
	for(int i=0; i<n; i++)
	{
		cin>>arr[i];
	}
	int size = sizeof(arr) / sizeof(arr[0]);

	mergeSort(arr, 0, size - 1);

	cout <<"\nSorted array: ";
	printArray(arr, size);
	return 0;
}
```

**Output:**

```
Enter number of Elements: 5
Enter Array Elements: 30 20 50 10 40

Sorted array: 10 20 30 40 50

--------------------------------
Process exited after 17.39 seconds with return value 0
Press any key to continue . . .
```

**Time complexity:**

| | |
|---|---|
| **Best Case** | O(n*logn) |
| **Average Case** | O(n*logn) |
| **Worst Case** | O(n*logn) |

- 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*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*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*log n).

**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.

## 2. Multithreaded Merge Sort

Merge Sort is a popular sorting technique which divides an array or list into two halves and then start merging them when sufficient depth is reached. Time complexity of merge sort is O(nlogn).

**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.

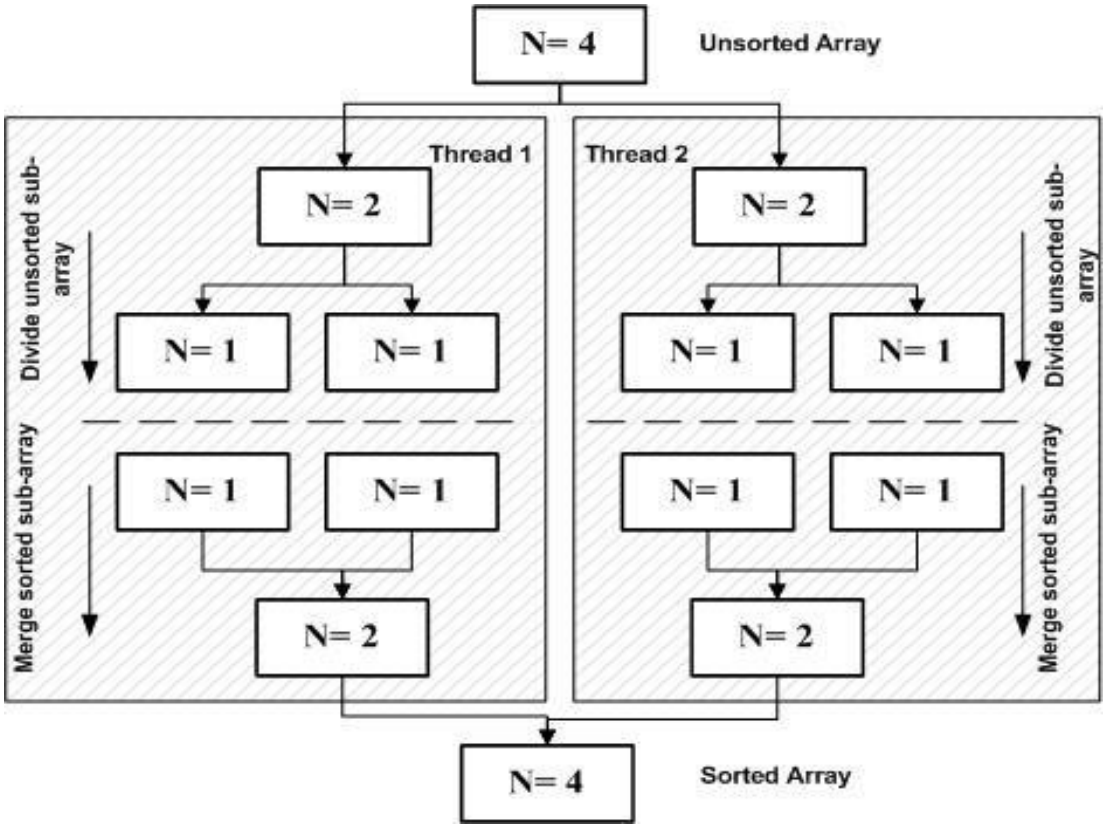


Fig: Multi-threaded Merge sort

**Algorithm:**

- We will start by generating the random numbers using the rand() method in C++ STL.
- Create an array of pthread_t type i.e. P_TH[thread_size].
- Start loop FOR from i to 0 till i is less than the size of a thread. Inside the loop, call pthread_create(&P_TH[i], NULL, Sorting_Threading, (void*)NULL) method to create the thread with given array values.

- Call function as combine_array(0, (size / 2 - 1) / 2, size / 2 - 1), combine_array(size / 2, size/2 + (size-1-size/2)/2, size - 1) and combine_array(0, (size - 1)/2, size - 1)
- Print the sorted array stored in a arr[] of integer type.
- Inside the function void* Sorting_Threading(void* arg)
    - Declare a variable as set_val to temp_val++, first to set_val * (size / 4), end to (set_val + 1) * (size / 4) - 1 and mid_val to first + (end - first) / 2
    - Check IF first less than end then call Sorting_Threading(first, mid_val), Sorting_Threading(mid_val + 1, end) and combine_array(first, mid_val, end);
- Inside the function void Sorting_Threading(int first, int end)
    - Declare variable as mid_val to first + (end - first) / 2
    - Check IF first less than end then call Sorting_Threading(first, mid_val), Sorting_Threading(mid_val + 1, end) and combine_array(first, mid_val, end)
- Inside the function void combine_array(int first, int mid_val, int end)
    - Declare variables as int* start to new int[mid_val - first + 1], int* last to new int[end - mid_val], temp_1 to mid_val - first + 1, temp_2 to end - mid_val, i, j, k to first.
    - Start loop FOR from i to 0 till i less than temp_1. Inside the loop, set start[i] to arr[i + first].
    - Start loop FOR from i to 0 till i less than temp_2. Inside the loop, set last[i] to arr[i + mid_val + 1]
    - Set i to j to 0. Start loop While i is less than temp_1 AND j less than temp_2. Inside the while, check IF start[i] less than last[j] then set arr[k++] to start[i++]. ELSE, set arr[k++] = last[j++]
    - Start WHILE i less than temp_1 then set arr[k++] = start[i++]. Start WHILE j less than temp_2 then set arr[k++] to last[j++]

**Program:**

```
#include <iostream>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#define size 20
#define thread_size 4
```

```
using namespace std;

int arr[size];
int temp_val = 0;

void combine_array(int first, int mid_val, int end)
{
    int* start = new int[mid_val - first + 1];
    int* last = new int[end - mid_val];
    int temp_1 = mid_val - first + 1;
    int temp_2 = end - mid_val;
    int i, j;
    int k = first;
    for(i = 0; i < temp_1; i++){
      start[i] = arr[i + first];
    }
    for (i = 0; i < temp_2; i++){
      last[i] = arr[i + mid_val + 1];
    }
    i = j = 0;
    while(i < temp_1 && j < temp_2){
      if(start[i] <= last[j]){
        arr[k++] = start[i++];
      }
      else{
        arr[k++] = last[j++];
      }
    }
    while (i < temp_1){
      arr[k++] = start[i++];
    }
    while (j < temp_2){
      arr[k++] = last[j++];
    }
}

void Sorting_Threading(int first, int end)
{
  int mid_val = first + (end - first) / 2;
  if(first < end)
  {
    Sorting_Threading(first, mid_val);
    Sorting_Threading(mid_val + 1, end);
    combine_array(first, mid_val, end);
  }
}
```

```
void* Sorting_Threading(void* arg)
{
  int set_val = temp_val++;
  int first = set_val * (size / 4);

  int end = (set_val + 1) * (size / 4) - 1;
  int mid_val = first + (end - first) / 2;

  if (first < end)
  {
    Sorting_Threading(first, mid_val);
    Sorting_Threading(mid_val + 1, end);
    combine_array(first, mid_val, end);
  }
}

int main()
{
  for(int i = 0; i < size; i++)
  {
    arr[i] = rand() % 100;
  }

  pthread_t P_TH[thread_size];
  for(int i = 0; i < thread_size; i++)
  {
    pthread_create(&P_TH[i], NULL, Sorting_Threading, (void*)NULL);
  }
  for(int i = 0; i < 4; i++)
  {
    pthread_join(P_TH[i], NULL);
  }

  combine_array(0, (size / 2 - 1) / 2, size / 2 - 1);
  combine_array(size / 2, size/2 + (size-1-size/2)/2, size - 1);
  combine_array(0, (size - 1)/2, size - 1);

  cout<<"Merge Sort using Multi-threading: ";

  for (int i = 0; i < size; i++)
  {
    cout << arr[i] << " ";
  }
  return 0;
}
```

**Output:**

```
Merge Sort using Multi-threading: 0 5 24 27 27 34 36 41 42 45 58 61 62 64 67 69 78 81 91 95
--------------------------------
Process exited after 0.5038 seconds with return value 0
Press any key to continue . . .
```

**Time Complexity:** O(n log2 n)

Running the algorithm in parallel does not become efficient until the size of the data sets reach 2 ^16.

**Conclusion:**

Hence, we implement merge sort and multithreaded merge sort. By Comparing time required by both the algorithms merge sort and multithreaded merge sort. We analyze the performance of each algorithm for the best case and the worst case.