**MINI PROJECT**

#include <iostream>

#include <vector>

#include <thread>

using namespace std;

void merge(std::vector<int> &arr, int left, int mid, int right)

{

int n1 = mid - left + 1;

int n2 = right - mid;

std::vector<int> left\_arr(n1);

std::vector<int> right\_arr(n2);

for (int i = 0; i < n1; i++)

{

left\_arr[i] = arr[left + i];

}

for (int i = 0; i < n2; i++)

{

right\_arr[i] = arr[mid + 1 + i];

}

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

while (i < n1 && j < n2)

{

if (left\_arr[i] <= right\_arr[j])

{

arr[k] = left\_arr[i];

i++;

}

else

{

arr[k] = right\_arr[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = left\_arr[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = right\_arr[j];

j++;

k++;

}

}

void merge\_sort(std::vector<int> &arr, int left, int right)

{

if (left < right)

{

int mid = left + (right - left) / 2;

merge\_sort(arr, left, mid);

merge\_sort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void multithreaded\_merge\_sort(std::vector<int> &arr, int left, int right, int depth)

{

if (left < right)

{

if (depth == 0)

{

merge\_sort(arr, left, right);

}

else

{

int mid = left + (right - left) / 2;

std::thread left\_thread(multithreaded\_merge\_sort, std::ref(arr), left, mid, depth - 1);

std::thread right\_thread(multithreaded\_merge\_sort, std::ref(arr), mid + 1, right, depth - 1);

left\_thread.join();

right\_thread.join();

merge(arr, left, mid, right);

}

}

}

int main()

{

std::vector<int> arr = {12, 11, 13, 5, 6, 7};

// Merge Sort

merge\_sort(arr, 0, arr.size() - 1);

std::cout << "Sorted array using Merge Sort: ";

for (int num : arr)

{

std::cout << num << " ";

}

std::cout << std::endl;

// Multithreaded Merge Sort

std::vector<int> arr2 = {12, 11, 13, 5, 6, 7};

int num\_threads = 2; // Adjust the number of threads as needed

multithreaded\_merge\_sort(arr2, 0, arr2.size() - 1, num\_threads);

std::cout << "Sorted array using Multithreaded Merge Sort: ";

for (int num : arr2)

{

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

**Regular Merge Sort:-**

Best Case:-

The best-case time complexity of regular merge sort is O(n log n), where "n" is the number of elements in the list. This is because the list is consistently divided into two halves until it reaches the base case (single-element sublists), and merging is performed efficiently.

Worst Case: -

The worst-case time complexity of regular merge sort is also O(n log n). This occurs when the list is divided into two halves in a balanced manner at each level of recursion, resulting in log n levels of recursion, and merging is still performed efficiently.

Performance Analysis:-

Regular merge sort is highly consistent in its performance, and its time complexity is optimal for general sorting tasks. It doesn't benefit significantly from parallelization, as the recursive nature of the algorithm makes it difficult to parallelize efficiently without incurring thread management overhead.

**Multithreaded Merge Sort:-**

Best Case:-

The best-case time complexity of multithreaded merge sort remains O(n log n) for the same reasons as the regular merge sort. In the best case, multithreading may provide some speedup, but it may not be substantial for smaller inputs or when the number of available CPU cores is limited.

Worst Case:-

The worst-case time complexity of multithreaded merge sort is also O(n log n), but the constant factors may be higher than in the regular merge sort due to the overhead of thread creation and synchronization. In some cases, the performance might even degrade compared to the regular merge sort, especially for small input sizes or when the number of threads used is excessive.

Performance Analysis:-

The performance of multithreaded merge sort depends heavily on factors such as the size of the input, the number of CPU cores available, and the efficiency of thread management. In general, it is most beneficial for large input sizes and when there are many available CPU cores. However, the benefits may not be noticeable for small lists or when the overhead of managing threads is significant.

In summary, both regular merge sort and multithreaded merge sort have the same theoretical time complexity of O(n log n) for both best and worst cases. Regular merge sort is more predictable and doesn't rely on parallelism, making it a reliable choice for general sorting tasks. Multithreaded merge sort can provide performance improvements but introduces complexities related to thread management, and its benefits are more pronounced for large input sizes and multicore processors. Careful benchmarking and analysis are needed to determine the actual performance gains for a specific use case.