**CODE:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <chrono>

#include <omp.h>

using namespace std;

using namespace std::chrono;

// Function to generate random numbers

void generateRandomArray(vector<int>& arr, int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 10000;  // Generate numbers in range [0, 9999]

    }

}

// Sequential Bubble Sort

void bubbleSort(vector<int>& arr, int n) {

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                swap(arr[j], arr[j + 1]);

            }

        }

    }

}

// Optimized Parallel Bubble Sort (Odd-Even Sort)

void parallelBubbleSort(vector<int>& arr, int n) {

    bool sorted = false;

    while (!sorted) {

        sorted = true;

        // Odd phase

        #pragma omp parallel for

        for (int i = 1; i < n - 1; i += 2) {

            if (arr[i] > arr[i + 1]) {

                swap(arr[i], arr[i + 1]);

                sorted = false;

            }

        }

        // Even phase

        #pragma omp parallel for

        for (int i = 0; i < n - 1; i += 2) {

            if (arr[i] > arr[i + 1]) {

                swap(arr[i], arr[i + 1]);

                sorted = false;

            }

        }

    }

}

// Merge function for Merge Sort

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

    int n1 = mid - left + 1;

    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];

    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) arr[k++] = L[i++];

        else arr[k++] = R[j++];

    }

    while (i < n1) arr[k++] = L[i++];

    while (j < n2) arr[k++] = R[j++];

}

// Sequential 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);

    }

}

// Optimized Parallel Merge Sort

void parallelMergeSort(vector<int>& arr, int left, int right) {

    if (left < right) {

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

        // Parallel execution only if data size is large

        if (right - left > 500) {

            #pragma omp parallel

            {

                #pragma omp single nowait

                {

                    #pragma omp task

                    parallelMergeSort(arr, left, mid);

                    #pragma omp task

                    parallelMergeSort(arr, mid + 1, right);

                }

            }

        } else {

            mergeSort(arr, left, mid);

            mergeSort(arr, mid + 1, right);

        }

        merge(arr, left, mid, right);

    }

}

// Function to measure execution time

template <typename Func>

double measureExecutionTime(Func sortFunction) {

    auto start = high\_resolution\_clock::now();

    sortFunction();

    auto stop = high\_resolution\_clock::now();

    return duration<double, milli>(stop - start).count();

}

int main() {

    int n = 5000;  // Adjust array size as needed

    vector<int> arr(n);

    // Sequential Bubble Sort

    generateRandomArray(arr, n);

    double timeBubbleSeq = measureExecutionTime([&]() { bubbleSort(arr, n); });

    cout << "Sequential Bubble Sort Time: " << timeBubbleSeq << " ms\n";

    // Parallel Bubble Sort (Odd-Even)

    generateRandomArray(arr, n);

    double timeBubblePar = measureExecutionTime([&]() { parallelBubbleSort(arr, n); });

    cout << "Parallel Bubble Sort Time: " << timeBubblePar << " ms\n";

    // Sequential Merge Sort

    generateRandomArray(arr, n);

    double timeMergeSeq = measureExecutionTime([&]() { mergeSort(arr, 0, n - 1); });

    cout << "Sequential Merge Sort Time: " << timeMergeSeq << " ms\n";

    // Parallel Merge Sort

    generateRandomArray(arr, n);

    double timeMergePar = measureExecutionTime([&]() { parallelMergeSort(arr, 0, n - 1); });

    cout << "Parallel Merge Sort Time: " << timeMergePar << " ms\n";

    return 0;

}

**OUTPUT**

Sequential Bubble Sort Time: 72.2472 ms

Parallel Bubble Sort Time: 457.619 ms

Sequential Merge Sort Time: 1.3395 ms

Parallel Merge Sort Time: 0.908 ms