Name: Vaishnavi Pravin Kolse

Class: BE Div: A

Roll No.37

Practical No.2

Aim: Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Program:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <omp.h>
#include <algorithm>

using namespace std;

vector<int> generateRandomVector(int size) {
    vector<int> vec(size);
    for (int &x : vec) {
        x = rand() % 1000;
    }
    return vec;
}

void bubbleSort(vector<int> &arr) {
    int n = arr.size();
    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]);
        }
    }
  }
}


void parallelBubbleSort(vector<int> &arr) {
  int n = arr.size();
  for (int i = 0; i < n - 1; i++) {
    #pragma omp parallel for
    for (int j = (i % 2); j < n - 1; j += 2) {
      if (arr[j] > arr[j + 1]) {
        swap(arr[j], arr[j + 1]);
      }
    }
  }
}


void merge(vector<int> &arr, int left, int mid, int right) {
  vector<int> temp(right - left + 1);
  int i = left, j = mid + 1, k = 0;

  while (i <= mid && j <= right) {
    if (arr[i] <= arr[j]) {
      temp[k++] = arr[i++];
    } else {
      temp[k++] = arr[j++];
```

```
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    copy(temp.begin(), temp.end(), arr.begin() + left);
}


void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}


void parallelMergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(arr, left, mid);
            #pragma omp section
            parallelMergeSort(arr, mid + 1, right);
```

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



int main() {
    srand(time(0));
    const int SIZE = 10000;
    vector<int> vec = generateRandomVector(SIZE);



    vector<int> seqBubbleSortVec = vec;
    clock_t start = clock();
    bubbleSort(seqBubbleSortVec);
    cout << "Sequential Bubble Sort Time: " << (double)(clock() - start) / CLOCKS_PER_SEC << "
seconds\n";


    vector<int> seqMergeSortVec = vec;
    start = clock();
    mergeSort(seqMergeSortVec, 0, SIZE - 1);
    cout << "Sequential Merge Sort Time: " << (double)(clock() - start) / CLOCKS_PER_SEC << "
seconds\n";



    vector<int> parBubbleSortVec = vec;
    start = clock();
    parallelBubbleSort(parBubbleSortVec);
    cout << "Parallel Bubble Sort Time: " << (double)(clock() - start) / CLOCKS_PER_SEC << "
seconds\n";
```

```
    vector<int> parMergeSortVec = vec;
    start = clock();
    parallelMergeSort(parMergeSortVec, 0, SIZE - 1);
    cout << "Parallel Merge Sort Time: " << (double)(clock() - start) / CLOCKS_PER_SEC << "
seconds\n";


    return 0;
}
```

Output:

Sequential Bubble Sort Time: 0.467943 seconds

Sequential Merge Sort Time: 0.003671 seconds

Parallel Bubble Sort Time: 0.455786 seconds

Parallel Merge Sort Time: 0.003459 seconds

=== Code Execution Successful ===