#include <iostream>

#include <ctime>

#include <cstdlib>

#include <omp.h>

using namespace std;

void bubbleSort(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]);

}

}

}

}

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int \*L = new int[n1];

int \*R = new int[n2];

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

{

L[i] = arr[l + i];

}

for (j = 0; j < n2; ++j)

{

R[j] = arr[m + 1 + j];

}

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

++i;

}

else

{

arr[k] = R[j];

++j;

}

++k;

}

while (i < n1)

{

arr[k] = L[i];

++i;

++k;

}

while (j < n2)

{

arr[k] = R[j];

++j;

++k;

}

delete[] L;

delete[] R;

}

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

int m = l + (r - l) / 2;

#pragma omp parallel sections

{

#pragma omp section

{

mergeSort(arr, l, m);

}

#pragma omp section

{

mergeSort(arr, m + 1, r);

}

}

merge(arr, l, m, r);

}

}

void printArray(int arr[], int size)

{

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

{

cout << arr[i] << " ";

}

cout << endl;

}

int main()

{

int n;

cout << "Enter the size of the array: ";

cin >> n;

int \*arr = new int[n];

srand(time(0));

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

{

arr[i] = rand() % 100;

}

// cout << "Original array: ";

// printArray(arr, n);

// Sequential Bubble Sort

clock\_t start = clock();

bubbleSort(arr, n);

clock\_t end = clock();

// cout << "Sequential Bubble Sorted array: ";

// printArray(arr, n);

double sequentialBubbleTime = double(end - start) / CLOCKS\_PER\_SEC;

// Parallel Bubble Sort

start = clock();

#pragma omp parallel

{

bubbleSort(arr, n);

}

end = clock();

// cout << "Parallel Bubble Sorted array: ";

// printArray(arr, n);

double parallelBubbleTime = double(end - start) / CLOCKS\_PER\_SEC;

// Merge Sort

start = clock();

mergeSort(arr, 0, n - 1);

end = clock();

// cout << "Sequential Merge Sorted array: ";

// printArray(arr, n);

double sequentialMergeTime = double(end - start) / CLOCKS\_PER\_SEC;

// Parallel Merge Sort

start = clock();

#pragma omp parallel

{

#pragma omp single

{

mergeSort(arr, 0, n - 1);

}

}

end = clock();

// cout << "Parallel Merge Sorted array: ";

// printArray(arr, n);

double parallelMergeTime = double(end - start) / CLOCKS\_PER\_SEC;

// Performance measurement

cout << "Sequential Bubble Sort Time: " << sequentialBubbleTime << " seconds" << endl;

cout << "Parallel Bubble Sort Time: " << parallelBubbleTime << " seconds" << endl;

cout << "Sequential Merge Sort Time: " << sequentialMergeTime << " seconds" << endl;

cout << "Parallel Merge Sort Time: " << parallelMergeTime << " seconds" << endl;

delete[] arr;

return 0;

}