**Assignment No 2**

Name: Poonam Kisan Salbande

Roll No: 20121011

Title: Write a program to implement Bubble sort and merge sort using open MP ,use existing

algorithm and measure performance of sequential and parallel algorithms.

Code:-

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

// Bubble Sort algorithm

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]) {

int temp = arr[j];

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

arr[j+1] = temp;

}

}

}

}

// Function to generate random array

void generateRandomArray(int arr[], int n) {

srand(time(NULL));

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

arr[i] = rand() % 100;

}

}

// Function to check if array is sorted

bool isSorted(int arr[], int n) {

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

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

return false;

}

}

return true;

}

// Function to measure execution time

double measureExecutionTime(void (\*sortFunction)(int[], int), int arr[], int n, int n\_threads)

{

double start\_time, end\_time;

omp\_set\_num\_threads(n\_threads);

start\_time = omp\_get\_wtime();

sortFunction(arr, n);

end\_time = omp\_get\_wtime();

return end\_time – start\_time;

}

int main() {

const int n = 10000;

int arr[n];

int n\_threads = omp\_get\_max\_threads();

// Generate random array

generateRandomArray(arr, n);

// Measure execution time of Bubble Sort

double bubbleSortSequentialTime = measureExecutionTime(bubbleSort, arr, n, 1);

double bubbleSortParallelTime = measureExecutionTime(bubbleSort, arr, n, n\_threads);

// Print results

cout << “Bubble Sort execution time with “ << n\_threads << “ threads: “ <<

bubbleSortParallelTime << “ seconds” << endl;

cout << “Bubble Sort execution time without parallelism: “ << bubbleSortSequentialTime

<< “ seconds” << endl;

// Check if array is sorted

if (isSorted(arr, n)) {

cout << “Array is sorted” << endl;

} else {

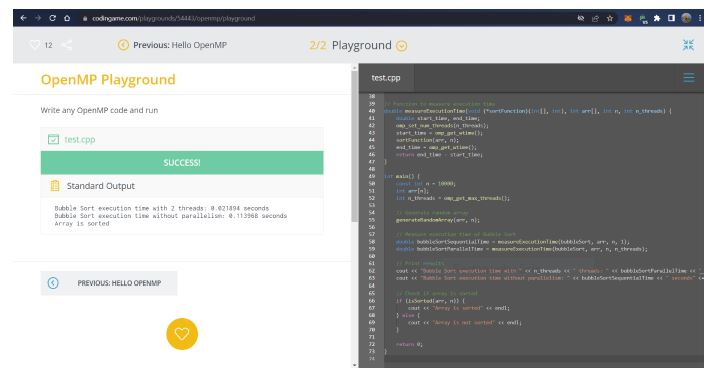
cout << “Array is not sorted” << endl;

}

return 0;

}

Output:



Merge Sort

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

// Merge Sort algorithm

void merge(int arr[], int left, int mid, int right) {

int i, j, k;

int n1 = mid – left + 1;

int n2 = right – mid;

// create temporary arrays

int L[n1], R[n2];

// copy data to temporary arrays L[] and R[]

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

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

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

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

// merge the temporary arrays back into arr[left..right]

i = 0; // initial index of first subarray

j = 0; // initial index of second subarray

k = left; // initial index of merged subarray

while (i < n1 && j < n2) {

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

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

k++;

}

// copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right – left) / 2;

// parallelize the recursive calls to mergeSort

#pragma omp parallel sections

{

#pragma omp section

mergeSort(arr, left, mid);

#pragma omp section

mergeSort(arr, mid + 1, right);

}

// merge the two sorted halves

merge(arr, left, mid, right);

}

}

// Function to generate random array

void generateRandomArray(int arr[], int n) {

srand(time(NULL));

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

arr[i] = rand() % 100;

}

}

// Function to check if array is sorted

bool isSorted(int arr[], int n) {

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

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

return false;

}

}

return true;

}

// Function to measure execution time

double measureExecutionTime(void (\*sortFunction)(int[], int, int), int arr[], int n, int

n\_threads) {

double start\_time, end\_time;

omp\_set\_num\_threads(n\_threads);

start\_time = omp\_get\_wtime();

sortFunction(arr, 0, n-1);

end\_time = omp\_get\_wtime();

return end\_time – start\_time;

}

int main() {

const int n = 10000;

int arr[n];

int n\_threads = omp\_get\_max\_threads();

// Generate random array

generateRandomArray(arr, n);

// Measure execution time of Merge Sort

double mergeSortSequentialTime = measureExecutionTime(mergeSort, arr, n, 1);

double mergeSortParallelTime = measureExecutionTime(mergeSort, arr, n, n\_threads);

// Print results

cout << “Merge Sort execution time with “ << n\_threads << “ threads: “ <<

mergeSortParallelTime << “ seconds” << endl;

cout << “Merge Sort execution time without parallelism: “ << mergeSortSequentialTime

<< “ seconds” << endl;

// Verify that array is sorted

if (isSorted(arr, n)) {

cout << “Array is sorted” << endl;

} else {

cout << “Array is not sorted” << endl;

}

return 0;}

Output:

