Assignment 2 extended

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <omp.h>

// Function to generate a random array of given size

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

srand(time(0));

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

arr[i] = rand() % 1000; // Generate random numbers between 0 and 999

}

}

// Function to print an array

void printArray(int arr[], int size) {

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

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

}

std::cout << std::endl;

}

// Sequential Bubble Sort

void bubbleSort(int arr[], int size) {

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

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

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

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

}

}

}

}

// Parallel Bubble Sort using OpenMP

void parallelBubbleSort(int arr[], int size) {

int i, j;

#pragma omp parallel for private(i, j) shared(arr)

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

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

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

#pragma omp critical

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

}

}

}

}

// Sequential Merge Sort

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

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

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

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

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

int 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;

}

}

void mergeSortHelper(int arr[], int l, int r) {

if (l < r) {

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

mergeSortHelper(arr, l, m);

mergeSortHelper(arr, m + 1, r);

merge(arr, l, m, r);

}

}

void mergeSort(int arr[], int size) {

mergeSortHelper(arr, 0, size - 1);

}

// Parallel Merge Sort using OpenMP

void parallelMergeSortHelper(int arr[], int l, int r) {

if (l < r) {

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

#pragma omp task

parallelMergeSortHelper(arr, l, m);

#pragma omp task

parallelMergeSortHelper(arr, m + 1, r);

#pragma omp taskwait

merge(arr, l, m, r);

}

}

void parallelMergeSort(int arr[], int size) {

#pragma omp parallel

{

#pragma omp single

{

parallelMergeSortHelper(arr, 0, size - 1);

}

}

}

int main() {

int size;

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

std::cin >> size;

int arr[size];

generateRandomArray(arr, size);

int arr\_copy[size]; // Copy for parallel sorting

// Sequential Bubble Sort

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

bubbleSort(arr, size);

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

std::cout << "Sequential Bubble Sort Time: " << end\_time - start\_time << " seconds\n";

// Parallel Bubble Sort

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

parallelBubbleSort(arr, size);

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

std::cout << "Parallel Bubble Sort Time: " << end\_time - start\_time << " seconds\n";

// Reset arr\_copy for Merge Sort

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

arr\_copy[i] = arr[i];

}

// Sequential Merge Sort

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

mergeSort(arr\_copy, size);

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

std::cout << "Sequential Merge Sort Time: " << end\_time - start\_time << " seconds\n";

// Parallel Merge Sort

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

parallelMergeSort(arr\_copy, size);

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

std::cout << "Parallel Merge Sort Time: " << end\_time - start\_time << " seconds\n";

return 0;

}