\*\*Implement Min, Max, Sum and Average operations using Parallel Reduction.\*\*

#include <iostream>

#include <vector>

#include <omp.h>

// Function to calculate the minimum value in parallel

template<typename T>

T parallel\_min(const std::vector<T>& values) {

T min\_value = values[0];

#pragma omp parallel for reduction(min: min\_value)

for (int i = 1; i < values.size(); i++) {

if (values[i] < min\_value) {

min\_value = values[i];

}

}

return min\_value;

}

// Function to calculate the maximum value in parallel

template<typename T>

T parallel\_max(const std::vector<T>& values) {

T max\_value = values[0];

#pragma omp parallel for reduction(max: max\_value)

for (int i = 1; i < values.size(); i++) {

if (values[i] > max\_value) {

max\_value = values[i];

}

}

return max\_value;

}

// Function to calculate the sum of values in parallel

template<typename T>

T parallel\_sum(const std::vector<T>& values) {

T sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += values[i];

}

return sum;

}

// Function to calculate the average of values in parallel

template<typename T>

double parallel\_average(const std::vector<T>& values) {

T sum = parallel\_sum(values);

int n = values.size();

return static\_cast<double>(sum) / n;

}

int main() {

std::vector<int> values = {5, 2, 8, 3, 1, 9, 4, 7, 6};

int min\_value = parallel\_min(values);

int max\_value = parallel\_max(values);

int sum = parallel\_sum(values);

double average = parallel\_average(values);

std::cout << "Min: " << min\_value << std::endl;

std::cout << "Max: " << max\_value << std::endl;

std::cout << "Sum: " << sum << std::endl;

std::cout << "Average: " << average << std::endl;

return 0;

}

\*\*\*\*\*\*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.

\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <chrono>

#include <algorithm>

#include <omp.h>

// Function to perform parallel bubble sort

template<typename T>

void parallel\_bubble\_sort(std::vector<T>& arr) {

int n = arr.size();

bool swapped = true;

#pragma omp parallel default(none) shared(arr, n, swapped)

{

while (swapped) {

#pragma omp single

swapped = false;

#pragma omp for

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

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

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

#pragma omp atomic write

swapped = true;

}

}

}

}

}

// Function to perform parallel merge sort

template<typename T>

void parallel\_merge\_sort(std::vector<T>& arr, int left, int right) {

if (left < right) {

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

#pragma omp parallel sections

{

#pragma omp section

{

parallel\_merge\_sort(arr, left, middle);

}

#pragma omp section

{

parallel\_merge\_sort(arr, middle + 1, right);

}

}

std::vector<T> temp(right - left + 1);

int i = left;

int j = middle + 1;

int k = 0;

while (i <= middle && j <= right) {

if (arr[i] <= arr[j]) {

temp[k] = arr[i];

i++;

} else {

temp[k] = arr[j];

j++;

}

k++;

}

while (i <= middle) {

temp[k] = arr[i];

i++;

k++;

}

while (j <= right) {

temp[k] = arr[j];

j++;

k++;

}

for (int p = 0; p < k; p++) {

arr[left + p] = temp[p];

}

}

}

// Function to generate random values in the vector

void generate\_random\_values(std::vector<int>& arr, int size) {

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

arr.push\_back(rand() % size);

}

}

// Function to measure the execution time of a given function

template<typename Func, typename... Args>

double measure\_execution\_time(Func&& func, Args&&... args) {

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

func(std::forward<Args>(args)...);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end - start;

return duration.count();

}

int main() {

int size;

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

std::cin >> size;

std::vector<int> arr;

generate\_random\_values(arr, size);

// Sequential Bubble Sort

std::vector<int> seqBubbleArr(arr);

double seqBubbleTime = measure\_execution\_time(std::sort<std::vector<int>::iterator>, seqBubbleArr.begin(), seqBubbleArr.end());

// Parallel Bubble Sort

std::vector<int> parallelBubbleArr(arr);

double parallelBubbleTime = measure\_execution\_time(parallel\_bubble\_sort<int>, parallelBubbleArr);

// Sequential Merge Sort

std::vector<int> seqMergeArr(arr);

double seqMergeTime = measure\_execution\_time(std::sort<std::vector<int>::iterator>, seqMergeArr.begin(), seqMergeArr.end());

// Parallel Merge Sort

std::vector<int> parallelMergeArr(arr);

double parallelMergeTime = measure\_execution\_time(parallel\_merge\_sort<int>, parallelMergeArr, 0, parallelMergeArr.size() - 1);

std::cout << "Sequential Bubble Sort Time: " << seqBubbleTime << " seconds" << std::endl;

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

std::cout << "Sequential Merge Sort Time: " << seqMergeTime << " seconds" << std::endl;

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

std::cout << "Sorted Array (Sequential Bubble Sort): ";

for (int num : seqBubbleArr) {

std::cout << num << " ";

}

std::cout << std::endl;

std::cout << "Sorted Array (Parallel Bubble Sort): ";

for (int num : parallelBubbleArr) {

std::cout << num << " ";

}

std::cout << std::endl;

std::cout << "Sorted Array (Sequential Merge Sort): ";

for (int num : seqMergeArr) {

std::cout << num << " ";

}

std::cout << std::endl;

std::cout << "Sorted Array (Parallel Merge Sort): ";

for (int num : parallelMergeArr) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*Design and implement Parallel Breadth First Search and Depth First Search based on existing

algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Code –

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

void parallel\_bfs(int start, vector<vector<int>> adj\_list, vector<bool>& visited)

{

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty()) {

int curr = q.front();

q.pop();

#pragma omp parallel for shared(adj\_list, visited, q)

for (int neighbor : adj\_list[curr]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

void parallel\_dfs(int curr, vector<vector<int>> adj\_list, vector<bool>& visited)

{

visited[curr] = true;

#pragma omp parallel for shared(adj\_list, visited)

for (int neighbor : adj\_list[curr]) {

if (!visited[neighbor]) {

#pragma omp critical

parallel\_dfs(neighbor, adj\_list, visited);

}

}

}

int main() {

int n, m;

cout << "Enter number of vertices and edges: ";

cin >> n >> m;

vector<vector<int>> adj\_list(n);

vector<bool> visited(n, false);

cout << "Enter edges:" << endl;

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

int a, b;

cin >> a >> b;

adj\_list[a].push\_back(b);

adj\_list[b].push\_back(a);

}

cout << "BFS: ";

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

if (!visited[i]) {

parallel\_bfs(i, adj\_list, visited);

}

cout << i << " ";

}

cout << endl;

visited.assign(n, false);

cout << "DFS: ";

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

if (!visited[i]) {

parallel\_dfs(i, adj\_list, visited);

}

cout << i << " ";

}

cout << endl;

return 0;

}