**Sum,min,max:**

Min, Max, Sum and Avg using parallel reduction

Code-

#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

using namespace std;

void min\_reduction(vector<int>& arr) {

int min\_value = INT\_MAX;

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

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

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

min\_value = arr[i];

}

}

cout << "Minimum value: " << min\_value << endl;

}

void max\_reduction(vector<int>& arr) {

int max\_value = INT\_MIN;

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

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

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

max\_value = arr[i];

}

}

cout << "Maximum value: " << max\_value << endl;

}

void sum\_reduction(vector<int>& arr) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += arr[i];

}

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

}

void average\_reduction(vector<int>& arr) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += arr[i];

}

cout << "Average: " << (double)sum / arr.size() << endl;

}

int main() {

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

min\_reduction(arr);

max\_reduction(arr);

sum\_reduction(arr);

average\_reduction(arr);

}

**Merge Sort:**

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

void merge(vector<int>& arr, int low, int mid, int high) {

int i, j, k;

int left\_side = mid - low + 1;

int right\_side = high - mid;

vector<int> left(left\_side), right(right\_side); // Create temporary arrays for the left and right halves

// Copy data to the temporary arrays

for (i = 0; i < right\_side; i++) {

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

}

for (j = 0; j < right\_side; j++) {

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

}

i = 0; // Index for the left subarray

j = 0; // Index for the right subarray

k = low; // Index for the merged array

// Merge the temporary arrays back into the original array

while (i < left\_side && j < right\_side) {

if (left[i] <= right[j]) {

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

} else {

arr[k++] = right[j++];

}

}

}

void merge\_sort(vector<int>& arr, int low, int high) {

if (low < high) {

int mid = low + (high - low) / 2;

#pragma omp task

merge\_sort(arr, low, mid);

#pragma omp task

merge\_sort(arr, mid + 1, high);

merge(arr, low, mid, high);

}

}

void parallel\_merge\_sort(vector<int>& arr) {

#pragma omp parallel

{

#pragma omp single

merge\_sort(arr, 0, arr.size() - 1);

}

}

int main() {

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

double start, end;

// Measure performance of sequential merge sort

start = omp\_get\_wtime();

merge\_sort(arr, 0, arr.size() - 1);

end = omp\_get\_wtime();

cout << "Sequential merge sort time: " << end - start <<endl;

// Measure performance of parallel merge sort

arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

start = omp\_get\_wtime();

parallel\_merge\_sort(arr);

end = omp\_get\_wtime();

cout << "Parallel merge sort time: " << end - start <<endl;

return 0;

}

**Bubble Sort:**

#include<iostream>

#include<omp.h>

#include<vector>

using namespace std;

void bubble\_sort(vector<int>& arr){

bool isSorted = false;

while(!isSorted){

isSorted = true;

#pragma omp parallel for

for(int i=0;i<arr.size()-1,i+=2){

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

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

isSorted=false;

}

}

#pragma omp parallel for

for(int i=1;i<arr.size()-1,i+=2){

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

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

isSorted=false;

}

}

}

}

int main(){

vector<int> arr = {5,2,3,9,8,6,1,3,5,6};

double start,end;

start=omp\_get\_wtime();

bubble\_sort(arr);

end=omp\_get\_wtime();

cout<<"time: "<<endl;

return 0;

}

**DFS:**

#include <iostream>

#include <vector>

#include <stack>

#include <omp.h>

using namespace std;

const int MAXN = 1e5;

vector<int> adj[MAXN + 5]; // adjacency list

bool visited[MAXN + 5]; // mark visited nodes

void dfs(int start\_node) {

stack<int> stack;

stack.push(start\_node);

while (!stack.empty()) {

int current\_node = stack.top();

stack.pop();

if (!visited[current\_node]) {

visited[current\_node] = true;

cout << "Visited node: " << current\_node << endl;

#pragma omp parallel for

for (int i = 0; i < adj[current\_node].size(); i++) {

int next\_node = adj[current\_node][i];

if (!visited[next\_node]) {

#pragma omp critical

stack.push(next\_node);

}

}

}

}

}

int main() {

cout << "Please enter the number of nodes: ";

int n; // number of nodes

cin >> n;

cout << "Please enter the number of edges: ";

int m; // number of edges

cin >> m;

for (int i = 1; i <= m; i++) {

cout << "Enter edge " << i << ": ";

int u, v; // edge between u and v

cin >> u >> v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

cout << "Enter the starting node of DFS: ";

int start\_node; // start node of DFS

cin >> start\_node;

dfs(start\_node);

    return 0;

}

**BFS:**

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

const int MAXN = 1e5;

vector<int> adj[MAXN + 5]; // adjacency list

bool visited[MAXN + 5]; // mark visited nodes

void bfs(int start\_node) {

queue<int> queue;

queue.push(start\_node);

while (!queue.empty()) {

int current\_node = queue.front();

queue.pop();

if (!visited[current\_node]) {

visited[current\_node] = true;

cout << "Visited node: " << current\_node << endl;

#pragma omp parallel for

for (int i = 0; i < adj[current\_node].size(); i++) {

int next\_node = adj[current\_node][i];

if (!visited[next\_node]) {

#pragma omp critical

queue.push(next\_node);

}

}

}

}

}

int main() {

cout << "Please enter the number of nodes: ";

int n; // number of nodes

cin >> n;

cout << "Please enter the number of edges: ";

int m; // number of edges

cin >> m;

for (int i = 1; i <= m; i++) {

cout << "Enter edge " << i << ": ";

int u, v; // edge between u and v

cin >> u >> v;

adj[u].push\_back(v);

adj[v].push\_back(u);

}

cout << "Enter the starting node of BFS: ";

int start\_node; // start node of BFS

cin >> start\_node;

bfs(start\_node);

    return 0;

}