**Ass – DFS/BFS**

#include <iostream>

#include <vector>

#include <stack>

#include <queue>

#include <omp.h>

#include <chrono>

using namespace std;

void BFS(const vector<vector<int>> &adjacencylist, vector<bool> &visited, queue<int> &q)

{

    while (!q.empty())

    {

        int vertex = q.front();

        q.pop();

        cout << vertex << " ";

#pragma omp parallel for // Parallelize the loop

        for (int i = 0; i < adjacencylist[vertex].size(); ++i)

        {

            int neighbour = adjacencylist[vertex][i];

            if (!visited[neighbour])

            {

#pragma omp critical // Ensure only one thread modifies visited

                {

                    visited[neighbour] = true;

                    q.push(neighbour);

                }

            }

        }

    }

}

void DFS(const vector<vector<int>> &adjacencylist, vector<bool> &visited, stack<int> &s)

{

    while (!s.empty())

    {

        int vertex = s.top();

        s.pop();

        cout << vertex << " ";

#pragma omp parallel for // Parallelize the loop

        for (int i = 0; i < adjacencylist[vertex].size(); ++i)

        {

            int neighbour = adjacencylist[vertex][i];

            if (!visited[neighbour])

            {

#pragma omp critical // Ensure only one thread modifies visited

                {

                    visited[neighbour] = true;

                    s.push(neighbour);

                }

            }

        }

    }

}

int main()

{

    int nvertices, nedges;

    cout << "Enter the number of vertices: ";

    cin >> nvertices;

    cout << "Enter the number of edges: ";

    cin >> nedges;

    vector<vector<int>> adjacencylist(nvertices);

    cout << "Enter the edges in (source destination) format: ";

    for (int i = 0; i < nedges; i++)

    {

        int u, v;

        cin >> u >> v;

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

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

    }

    int startvertex;

    cout << "Enter the starting vertex: ";

    cin >> startvertex;

    // Measure BFS traversal time

    cout << "BFS traversal is: ";

    vector<bool> visitedBFS(nvertices, false);

    queue<int> q;

    visitedBFS[startvertex] = true;

    q.push(startvertex);

    auto start\_bfs = chrono::steady\_clock::now();

    BFS(adjacencylist, visitedBFS, q);

    auto end\_bfs = chrono::steady\_clock::now();

    cout << endl;

    auto duration\_bfs = chrono::duration\_cast<chrono::microseconds>(end\_bfs - start\_bfs);

    cout << "Time taken for BFS traversal: " << duration\_bfs.count() << " microseconds" << endl;

    cout << "Speedup BFS: " << static\_cast<double>(start\_bfs.time\_since\_epoch().count()) / end\_bfs.time\_since\_epoch().count() << endl;

    // Measure DFS traversal time

    cout << "DFS traversal is: ";

    vector<bool> visitedDFS(nvertices, false);

    stack<int> s;

    visitedDFS[startvertex] = true;

    s.push(startvertex);

    auto start\_dfs = chrono::steady\_clock::now();

    DFS(adjacencylist, visitedDFS, s);

    auto end\_dfs = chrono::steady\_clock::now();

    cout << endl;

    auto duration\_dfs = chrono::duration\_cast<chrono::microseconds>(end\_dfs - start\_dfs);

    cout << "Time taken for DFS traversal: " << duration\_dfs.count() << " microseconds" << endl;

    cout << "Speedup DFS: " << static\_cast<double>(start\_dfs.time\_since\_epoch().count()) / end\_dfs.time\_since\_epoch().count() << endl;

    return 0;

}

**ASSIGNMET – MERGE/BUBBLE SORT**

#include<iostream>

#include<bits/stdc++.h>

using namespace std;

void merge(int arr[] , int start , int mid , int end)

{

    int leftsize = mid - start+1;

    int rightsize = end - mid;

    int left[leftsize];

    int right[rightsize];

    for(int i=0;i<leftsize;i++)

    {

        left[i] = arr[start+i];

    }

    for(int j=0;j<rightsize;j++)

    {

        right[j]=arr[mid+1+j];

    }

int i,j , k;

    while(i<leftsize && j<rightsize)

    {

       if(left[i]<right[j])

       {

        arr[k] = left[i];

        i++;

       }

       else{

        arr[k] = right[j];

        j++;

       }

    }

    while(i<leftsize)

    {

        arr[k] = left[i];

        i++;

        k++;

    }

    while(j<rightsize)

    {

        arr[j] = right[j];

        j++;

        k++;

    }

}

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

{

    if (l < r)

    {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

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

    if (l < r)

    {

        int m = l + (r - l) / 2;

        #pragma omp parallel sections

        {

            #pragma omp section

            {

                merge\_para(arr, l, m);

            }

            #pragma omp section

            {

                merge\_para(arr, m + 1, r);

            }

        }

        merge(arr, l, m, r);

    }

}

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

    int i, j;

    for (i = 0; i < n - 1; i++) {

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

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

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

            }

        }

    }

}

void swap\_para(int \*a, int \*b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

void bubble\_para(int arr[], int n)

{

    int i = 0, j = 0;

    int f;

    for (i = 0; i < n - 1; i++)

    {

        f = i % 2;

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

        for (j = f; j < n - 1; j++)

        {

            if (arr[j] > arr[j + 1])

            {

                swap\_para(&arr[j], &arr[j + 1]);

            }

        }

    }

}

int main() {

    int n;

    cout << "Enter no of elements in array:";

    cin >> n;

    int arr[n];

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % n;

    }

    auto start = chrono::steady\_clock::now();

    mergeSort(arr, 0, n - 1);

    auto end = chrono::steady\_clock::now();

    chrono::duration<double, micro> fp = end - start;

    cout << "Sequential Merge Sort Time: " << fp.count() << " microseconds" << endl;

    auto start1 = chrono::steady\_clock::now();

    //merge\_para(arr2, 0, n - 1);

    auto end1 = chrono::steady\_clock::now();

    chrono::duration<double, micro> fp1 = end1 - start1;

    cout << "Parallel Merge Sort Time: " << fp1.count() << " microseconds" << endl;

    auto start2 = chrono::steady\_clock::now();

    bubbleSort(arr, n);

    auto end2 = chrono::steady\_clock::now();

    chrono::duration<double, micro> fp2 = end2 - start2;

    cout << "Sequential Bubble Sort Time: " << fp2.count() << " microseconds" << endl;

    auto start3 = chrono::steady\_clock::now();

    //bubble\_para(arr4, n);

    auto end3 = chrono::steady\_clock::now();

    chrono::duration<double, micro> fp3 = end3 - start3;

    cout << "Parallel Bubble Sort Time: " << fp3.count() << " microseconds" << endl;

    // Calculate speedup for Merge Sort

    double SortSpeedup = fp.count() / fp1.count();

    cout << "Speedup Merge Sort: " << SortSpeedup << endl;

        // Calculate speedup for Bubble Sort

    double BSortSpeedup = fp2.count() / fp3.count();

    cout << "Speedup Bubble Sort: " << BSortSpeedup << endl;

**Assignment – min/max**

#include <iostream>

#include <chrono>

#include <random>

#include <algorithm>

#include <iomanip>

using namespace std;

const int dataSize = 1000000;

int data[dataSize];

// Function to initialize data with random values

void initializeData()

{

    random\_device rd;

    mt19937 gen(rd());

    uniform\_int\_distribution<> dis(0, 99);

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

    {

        ::data[i] = dis(gen);

    }

}

/\*// Function to initialize data with random values

void initializeData() {

    #pragma omp parallel for

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

        data[i] = rand() % 100;

    }

}

\*/

// Function to perform parallel reduction for min, max, sum, and average

void parallelReduction(int& minVal, int& maxVal, int& sum, double& avg) {

    minVal = numeric\_limits<int>::max();

    maxVal = numeric\_limits<int>::min();

    sum = 0;

    // Parallel reduction for min, max, and sum

    #pragma omp parallel for reduction(min:minVal) reduction(max:maxVal) reduction(+:sum)

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

    {

        minVal = min(minVal, ::data[i]);

        maxVal = max(maxVal, ::data[i]);

        sum += ::data[i];

    }

    // Calculate average

    avg = static\_cast<double>(sum) / dataSize;

}

// Function to perform sequential reduction for min, max, sum, and average

void sequentialReduction(int& minVal, int& maxVal, int& sum, double& avg) {

    minVal = numeric\_limits<int>::max();

    maxVal = numeric\_limits<int>::min();

    sum = 0;

    // Sequential reduction for min, max, and sum

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

        minVal = min(minVal, ::data[i]);

        maxVal = max(maxVal, ::data[i]);

        sum += ::data[i];

    }

    // Calculate average

    avg = static\_cast<double>(sum) / dataSize;

}

int main() {

    // Initialize data with random values

    initializeData();

    // Perform parallel reduction and measure time

    auto startParallel = chrono::steady\_clock::now();

    int minValP, maxValP, sumP;

    double avgP;

    parallelReduction(minValP, maxValP, sumP, avgP);

    auto endParallel = chrono::steady\_clock::now();

    auto durationParallel = chrono::duration\_cast<chrono::milliseconds>(endParallel - startParallel).count();

    // Perform sequential reduction and measure time

    auto startSequential = chrono::steady\_clock::now();

    int minValS, maxValS, sumS;

    double avgS;

    sequentialReduction(minValS, maxValS, sumS, avgS);

    auto endSequential = chrono::steady\_clock::now();

    auto durationSequential = chrono::duration\_cast<chrono::milliseconds>(endSequential - startSequential).count();

    // Display results and time taken

    cout << "Parallel Min: " << minValP << ", Max: " << maxValP << ", Sum: " << sumP << ", Avg: " << avgP << endl;

    cout << "Sequential Min: " << minValS << ", Max: " << maxValS << ", Sum: " << sumS << ", Avg: " << avgS << endl;

    cout << "Parallel Time: " << durationParallel << " milliseconds" << endl;

    cout << "Sequential Time: " << durationSequential << " milliseconds" << endl;

    // Add speedup calculation

    double speedup = durationSequential / durationParallel;

    cout << "Speedup is: "  << speedup << std::endl;

    return 0;

}