ASSIGNMENT 1:  
  
#include<iostream>

#include<stdlib.h>

#include<queue>

using namespace std;

class node

{

public:

node \*left, \*right;

int data;

};

class Breadthfs

{

public:

node \*insert(node \*, int);

void bfs(node \*);

void dfs(node \*);

};

node \*Breadthfs::insert(node \*root, int data)

{

if(!root)

{

root = new node;

root->left = NULL;

root->right = NULL;

root->data = data;

return root;

}

queue<node \*> q;

q.push(root);

while(!q.empty())

{

node \*temp = q.front();

q.pop();

if(temp->left == NULL)

{

temp->left = new node;

temp->left->left = NULL;

temp->left->right = NULL;

temp->left->data = data;

return root;

}

else

{

q.push(temp->left);

}

if(temp->right == NULL)

{

temp->right = new node;

temp->right->left = NULL;

temp->right->right = NULL;

temp->right->data = data;

return root;

}

else

{

q.push(temp->right);

}

}

return root;

}

void Breadthfs::bfs(node \*head)

{

queue<node\*> q;

q.push(head);

int qSize;

while (!q.empty())

{

qSize = q.size();

#pragma omp parallel for

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

{

node\* currNode;

#pragma omp critical

{

currNode = q.front();

q.pop();

cout << "\t" << currNode->data;

}

#pragma omp critical

{

if(currNode->left)

q.push(currNode->left);

if(currNode->right)

q.push(currNode->right);

}

}

}

}

void Breadthfs::dfs(node \*head)

{

if (head == NULL)

return;

// Visit the current node (pre-order)

cout << "\t" << head->data;

// Recursively visit the left subtree

dfs(head->left);

// Recursively visit the right subtree

dfs(head->right);

}

int main()

{

Breadthfs b;

node \*root = NULL;

int data;

char ans;

do

{

cout << "\nEnter data => ";

cin >> data;

root = b.insert(root, data);

cout << "Do you want to insert one more node? (y/n): ";

cin >> ans;

} while(ans == 'y' || ans == 'Y');

cout << "\nBreadth-First Search (BFS) Traversal: ";

b.bfs(root);

cout << "\nDepth-First Search (DFS) Traversal: ";

b.dfs(root);

return 0;

}

Assignment \_1

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

class Graph {

int V;

vector<vector<int>> adj;

public:

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int u, int v) {

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

adj[v].push\_back(u); // For undirected graph

}

void parallelBFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

cout << "\nParallel BFS starting from node " << start << ":\n";

while (!q.empty()) {

int size = q.size();

vector<int> levelNodes;

#pragma omp parallel

{

vector<int> localNodes;

#pragma omp for

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

int node = -1;

bool valid = false;

#pragma omp critical

{

if (!q.empty()) {

node = q.front();

q.pop();

valid = true;

}

}

if (!valid) continue;

localNodes.push\_back(node);

for (int neighbor : adj[node]) {

bool needVisit = false;

#pragma omp critical

{

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

needVisit = true;

}

}

}

}

#pragma omp critical

levelNodes.insert(levelNodes.end(), localNodes.begin(), localNodes.end());

}

for (int node : levelNodes)

cout << node << " ";

}

cout << endl;

}

void parallelDFSUtil(int node, vector<bool>& visited) {

bool alreadyVisited;

#pragma omp critical

{

alreadyVisited = visited[node];

if (!alreadyVisited) {

visited[node] = true;

cout << node << " ";

}

}

if (alreadyVisited) return;

#pragma omp parallel for

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

int neighbor = adj[node][i];

#pragma omp task

parallelDFSUtil(neighbor, visited);

}

}

void parallelDFS(int start) {

vector<bool> visited(V, false);

cout << "\nParallel DFS starting from node " << start << ":\n";

#pragma omp parallel

{

#pragma omp single

parallelDFSUtil(start, visited);

}

cout << endl;

}

};

int main() {

int V, E;

cout << "Enter number of vertices: ";

cin >> V;

Graph g(V);

cout << "Enter number of edges: ";

cin >> E;

cout << "Enter each edge as two space-separated vertices (u v):\n";

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

int u, v;

cin >> u >> v;

g.addEdge(u, v);

}

int start;

cout << "Enter starting node for traversal: ";

cin >> start;

g.parallelBFS(start);

g.parallelDFS(start);

return 0;

}

**Assignment\_2**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <algorithm>

#include <chrono> // Include chrono for timing

using namespace std;

using namespace chrono; // Use the chrono library for time measurements

// Sequential Bubble Sort

void bubbleSortSequential(vector<int>& arr) {

int n = arr.size();

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

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

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

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

}

// Parallel Bubble Sort

void bubbleSortParallel(vector<int>& arr) {

int n = arr.size();

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

#pragma omp parallel for

for (int j = i % 2; j < n - 1; j += 2) {

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

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

}

}

}

}

void merge(vector<int>& arr, int left, int mid, int right) {

vector<int> temp(right - left + 1);

int i = left, j = mid + 1, k = 0;

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

if (arr[i] <= arr[j]) temp[k++] = arr[i++];

else temp[k++] = arr[j++];

}

while (i <= mid) temp[k++] = arr[i++];

while (j <= right) temp[k++] = arr[j++];

for (i = left; i <= right; ++i) arr[i] = temp[i - left];

}

// Sequential Merge Sort

void mergeSortSequential(vector<int>& arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSortSequential(arr, left, mid);

mergeSortSequential(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

// Parallel Merge Sort with OpenMP

void mergeSortParallel(vector<int>& arr, int left, int right, int depth = 0) {

if (left < right) {

int mid = (left + right) / 2;

if (depth < 4) {

#pragma omp parallel sections

{

#pragma omp section

mergeSortParallel(arr, left, mid, depth + 1);

#pragma omp section

mergeSortParallel(arr, mid + 1, right, depth + 1);

}

} else {

mergeSortSequential(arr, left, mid);

mergeSortSequential(arr, mid + 1, right);

}

merge(arr, left, mid, right);

}

}

// Utility to print array

void printArray(const vector<int>& arr) {

for (int val : arr) cout << val << " ";

cout << "\n";

}

int main() {

int size;

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

cin >> size;

vector<int> original(size);

cout << "Enter " << size << " integers:\n";

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

cin >> original[i];

// Sequential Bubble Sort

vector<int> arr1 = original;

auto start = high\_resolution\_clock::now(); // Start timing using chrono

bubbleSortSequential(arr1);

auto end = high\_resolution\_clock::now(); // End timing

cout << "\nSorted (Sequential Bubble Sort): ";

printArray(arr1);

auto duration = duration\_cast<milliseconds>(end - start);

cout << "Time: " << duration.count() << " milliseconds\n";

// Parallel Bubble Sort

vector<int> arr2 = original;

start = high\_resolution\_clock::now(); // Start timing

bubbleSortParallel(arr2);

end = high\_resolution\_clock::now(); // End timing

cout << "\nSorted (Parallel Bubble Sort): ";

printArray(arr2);

duration = duration\_cast<milliseconds>(end - start);

cout << "Time: " << duration.count() << " milliseconds\n";

// Sequential Merge Sort

vector<int> arr3 = original;

start = high\_resolution\_clock::now(); // Start timing

mergeSortSequential(arr3, 0, arr3.size() - 1);

end = high\_resolution\_clock::now(); // End timing

cout << "\nSorted (Sequential Merge Sort): ";

printArray(arr3);

duration = duration\_cast<milliseconds>(end - start);

cout << "Time: " << duration.count() << " milliseconds\n";

// Parallel Merge Sort

vector<int> arr4 = original;

start = high\_resolution\_clock::now(); // Start timing

mergeSortParallel(arr4, 0, arr4.size() - 1);

end = high\_resolution\_clock::now(); // End timing

cout << "\nSorted (Parallel Merge Sort): ";

printArray(arr4);

duration = duration\_cast<milliseconds>(end - start);

cout << "Time: " << duration.count() << " milliseconds\n";

return 0;

}

**Assignmet\_3**

#include <iostream>

#include <vector>

#include <limits>

#include <climits> // For INT\_MAX and INT\_MIN

#include <omp.h>

using namespace std;

int main() {

int n, num\_threads;

cout << "Enter number of elements: ";

cin >> n;

vector<int> data(n);

cout << "Enter the elements separated by space:\n";

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

cin >> data[i];

}

cout << "Enter number of threads to use: ";

cin >> num\_threads;

int min\_val = INT\_MAX;

int max\_val = INT\_MIN;

long long sum = 0;

// Parallel reduction using OpenMP

#pragma omp parallel for reduction(min:min\_val) reduction(max:max\_val) reduction(+:sum) num\_threads(num\_threads)

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

min\_val = min(min\_val, data[i]);

max\_val = max(max\_val, data[i]);

sum += data[i];

}

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

cout << "\n--- Parallel Reduction Results ---\n";

cout << "Minimum: " << min\_val << "\n";

cout << "Maximum: " << max\_val << "\n";

cout << "Sum : " << sum << "\n";

cout << "Average: " << average << "\n";

return 0;

}

Assignment\_4

#include <iostream>

#include <omp.h>

using namespace std;

int main() {

const int N = 4;

int A[N][N] = {{1, 2, 3, 4},

{4, 3, 2, 1},

{1, 1, 1, 1},

{2, 2, 2, 2}};

int B[N][N] = {{1, 0, 0, 1},

{0, 1, 1, 0},

{1, 0, 1, 0},

{0, 1, 0, 1}};

int C[N][N] = {0};

// Parallel matrix multiplication using OpenMP

#pragma omp parallel for collapse(2)

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

for(int j = 0; j < N; j++) {

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

C[i][j] += A[i][k] \* B[k][j];

}

}

}

// Display result matrix

cout << "Result Matrix C (A x B):\n";

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

for(int j = 0; j < N; j++) {

cout << C[i][j] << " ";

}

cout << endl;

}

return 0;

}