**DFS**

#include <iostream>

#include <vector>

#include <stack>

#include <omp.h>

using namespace std;

const int MAX = 100000;

vector<int> graph[MAX];

bool visited[MAX];

void dfs(int node) {

stack<int> s;

s.push(node);

while (!s.empty()) {

int curr\_node = s.top();

s.pop();

if (!visited[curr\_node]) {

visited[curr\_node] = true;

if (visited[curr\_node]) {

cout << curr\_node << " ";

}

#pragma omp parallel for

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

int adj\_node = graph[curr\_node][i];

if (!visited[adj\_node]) {

s.push(adj\_node);

}

}

}

}

}

int main() {

int n, m, start\_node;

cout << "Enter No of Node,Edges,and start node:" ;

cin >> n >> m >> start\_node;

//n: node,m:edges

cout << "Enter Pair of edges:" ;

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

int u, v;

cin >> u >> v;

//u and v: Pair of edges

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

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

}

#pragma omp parallel for

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

visited[i] = false;

}

dfs(start\_node);

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

if (visited[i]) {

cout << i << " ";

}

}\*/

return 0;

}

**BFS**

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_set>

#include <omp.h>

using namespace std;

// Data structure to represent a graph

class Graph {

private:

int V; // Number of vertices

vector<vector<int>> adj; // Adjacency list

public:

Graph(int V) : V(V) {

adj.resize(V);

}

// Function to add an edge to the graph

void addEdge(int v, int w) {

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

}

// Breadth First Search

void BFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

unordered\_set<int> currentLevel;

visited[start] = true;

q.push(start);

currentLevel.insert(start);

cout << "BFS starting from vertex " << start << ": " << endl; // Moved this line here

while (!q.empty()) {

unordered\_set<int> nextLevel;

while (!q.empty()) {

int v = q.front();

q.pop();

cout << v << " ";

for (int u : adj[v]) {

if (!visited[u]) {

visited[u] = true;

nextLevel.insert(u);

}

}

}

for (int u : nextLevel) {

q.push(u);

}

}

}

};

int main() {

Graph g(7); // Create a graph with 7 vertices

// Add edges

g.addEdge(0, 1);

g.addEdge(0, 4);

g.addEdge(1, 5);

g.addEdge(1, 6);

g.addEdge(4, 5);

g.addEdge(4, 3);

cout << "BFS starting from vertex 0: " << endl; // Corrected vertex index here

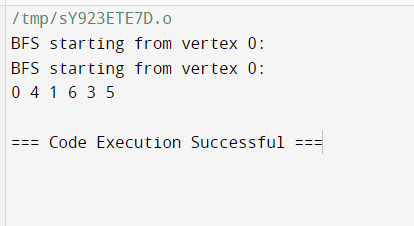
g.BFS(0); // Corrected the start vertex index

return 0;

}

Run Commands:

1. g++ -fopenmp bfs.cpp -o bfs
2. ./bfs



**Bubble Sort**

#include<iostream>

#include<stdlib.h>

#include<omp.h>

using namespace std;

void bubble(int \*, int);

void swap(int &, int &);

void bubble(int \*a, int n)

{

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

{

int first = i % 2;

#pragma omp parallel for shared(a,first)

for( int j = first; j < n-1; j += 2 )

{

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

{

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

}

}

}

}

void swap(int &a, int &b)

{

int test;

test=a;

a=b;

b=test;

}

int main()

{

int \*a,n;

cout<<"\n enter total no of elements=>";

cin>>n;

a=new int[n];

cout<<"\n enter elements=>";

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

{

cin>>a[i];

}

bubble(a,n);

cout<<"\n sorted array is=>";

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

{

cout<<a[i]<<endl;

}

return 0;

}

**Merge Sort**

#include<iostream>

#include<stdlib.h>

#include<omp.h>

using namespace std;

void mergesort(int a[],int i,int j);

void merge(int a[],int i1,int j1,int i2,int j2);

void mergesort(int a[],int i,int j)

{

int mid;

if(i<j)

{

mid=(i+j)/2;

#pragma omp parallel sections

{

#pragma omp section

{

mergesort(a,i,mid);

}

#pragma omp section

{

mergesort(a,mid+1,j);

}

}

merge(a,i,mid,mid+1,j);

}

}

void merge(int a[],int i1,int j1,int i2,int j2)

{

int temp[1000];

int i,j,k;

i=i1;

j=i2;

k=0;

while(i<=j1 && j<=j2)

{

if(a[i]<a[j])

{

temp[k++]=a[i++];

}

else

{

temp[k++]=a[j++];

}

}

while(i<=j1)

{

temp[k++]=a[i++];

}

while(j<=j2)

{

temp[k++]=a[j++];

}

for(i=i1,j=0;i<=j2;i++,j++)

{

a[i]=temp[j];

}

}

int main()

{

int \*a,n,i;

cout<<"\n enter total no of elements=>";

cin>>n;

a= new int[n];

cout<<"\n enter elements=>\n";

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

{

cin>>a[i];

}

mergesort(a, 0, n-1);

cout<<"\n sorted array is=>";

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

{

cout<<"\n"<<a[i];

}

return 0;

}

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

#include <iostream>

//#include <vector>

#include <omp.h>

#include <climits>

using namespace std;

void min\_reduction(int arr[], int n) {

int min\_value = INT\_MAX;

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

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

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

min\_value = arr[i];

}

}

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

}

void max\_reduction(int arr[], int n) {

int max\_value = INT\_MIN;

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

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

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

max\_value = arr[i];

}

}

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

}

void sum\_reduction(int arr[], int n) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += arr[i];

}

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

}

void average\_reduction(int arr[], int n) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += arr[i];

}

cout << "Average: " << (double)sum / (n-1) << endl;

}

int main() {

int \*arr,n;

cout<<"\n enter total no of elements=>";

cin>>n;

arr=new int[n];

cout<<"\n enter elements=>";

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

{

cin>>arr[i];

}

// int arr[] = {5, 2, 9, 1, 7, 6, 8, 3, 4};

// int n = size(arr);

min\_reduction(arr, n);

max\_reduction(arr, n);

sum\_reduction(arr, n);

average\_reduction(arr, n);

}

**Aim: Write a CUDA Program for : 1. Addition of two large vectors.**

#include<stdio.h>

#include<iostream>

#include<cstdlib>

//\*\*\*\*important to add following library to allow a programmer to use parallel paradigms\*\*\*\*\*

#include<omp.h>

using namespace std;

#define MAX 5

int main()

{

int a[MAX],b[MAX],c[MAX],i;

printf("\n First Vector:\t");

//Instruct a master thread to fork and generate more threads to process following loop structure

#pragma omp parallel for

for(i=0;i<MAX;i++)

{

a[i]=rand()%1000;

}

//Discuss issue of this for loop below-if we make it parallel, possibly values that get printed will not be in sequence as we dont have any control on order of threads execution

for(i=0;i<MAX;i++)

{

printf("%d\t",a[i]);

}

printf("\n Second Vector:\t");

#pragma omp parallel for

for(i=0;i<MAX;i++)

{

b[i]=rand()%1000;

}

for(i=0;i<MAX;i++)

{

printf("%d\t",b[i]);

}

printf("\n Parallel-Vector Addition:(a,b,c)\t");

#pragma omp parallel for

for(i=0;i<MAX;i++)

{

c[i]=a[i]+b[i];

}

for(i=0;i<MAX;i++)

{

printf("\n%d\t%d\t%d",a[i],b[i],c[i]);

}

}