Compile : g++ -fopenmp f\_nm.cpp -o ac

Run : ./ac

1. **BFS**

#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 \*);

};

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

// inserts a node in tree

{

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);

}

}

}

void bfs(node \*head)

{

queue<node\*> q;

q.push(head);

int qSize;

while (!q.empty())

{

qSize = q.size();

#pragma omp parallel for

//creates parallel threads

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

{

node\* currNode;

#pragma omp critical

{

currNode = q.front();

q.pop();

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

}// prints parent node

#pragma omp critical

{

if(currNode->left)// push parent's left node in queue

q.push(currNode->left);

if(currNode->right)

q.push(currNode->right);

}// push parent's right node in queue

}

}

}

int main(){

node \*root=NULL;

int data;

char ans;

do

{

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

cin>>data;

root=insert(root,data);

cout<<"do you want insert one more node?";

cin>>ans;

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

bfs(root);

return 0; }

1. **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);

return 0;

}

1. **Bubble**

**#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;**

**}**

1. **Merge**

**#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=>";**

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

**{**

**cin>>a[i];**

**}**

**mergesort(a, 0, n-1);**

**// stop…….**

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

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

**{**

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

**}**

**// Cout<<Stop-Start**

**return 0;**

**}**

1. **Min, Max, Sum and Average**

#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);

}

1. **CUDA (ADDITION)**

#include <iostream>

#include <cuda\_runtime.h>

using namespace std;

\_\_global\_\_ void addVectors(int\* A, int\* B, int\* C, int n)

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if (i < n)

{

C[i] = A[i] + B[i];

}

}

int main()

{

int n = 1000000;

int\* A, \* B, \* C;

int size = n \* sizeof(int);

// Allocate memory on the host

cudaMallocHost(&A, size);

cudaMallocHost(&B, size);

cudaMallocHost(&C, size);

// Initialize the vectors

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

{

A[i] = i;

B[i] = i \* 2;

}

// Allocate memory on the device

int\* dev\_A, \* dev\_B, \* dev\_C;

cudaMalloc(&dev\_A, size);

cudaMalloc(&dev\_B, size);

cudaMalloc(&dev\_C, size);

// Copy data from host to device

cudaMemcpy(dev\_A, A, size, cudaMemcpyHostToDevice);

cudaMemcpy(dev\_B, B, size, cudaMemcpyHostToDevice);

// Launch the kernel

int blockSize = 256;

int numBlocks = (n + blockSize - 1) / blockSize;

addVectors<<<numBlocks, blockSize>>>(dev\_A, dev\_B, dev\_C, n);

// Copy data from device to host

cudaMemcpy(C, dev\_C, size, cudaMemcpyDeviceToHost);

// Print the results

for (int i = 0; i < 10; i++)

{

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

}

cout << endl;

// Free memory

cudaFree(dev\_A);

cudaFree(dev\_B);

cudaFree(dev\_C);

cudaFreeHost(A);

cudaFreeHost(B);

cudaFreeHost(C);

return 0;

}