**1(A)-: BFS**

#include <iostream>

#include <stdlib.h>

#include <queue>

#include <vector>

#include <omp.h>

using namespace std;

class node {

public:

node \*left, \*right;

int data;

};

class Breadthfs {

public:

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

void bfs(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) {

if (!head) return;

queue<node \*> q;

q.push(head);

while (!q.empty()) {

int qSize = q.size();

vector<node \*> levelNodes(qSize);

// Sequentially store nodes to avoid race conditions

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

levelNodes[i] = q.front();

q.pop();

}

// Process nodes in parallel

#pragma omp parallel for

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

cout << "\t" << levelNodes[i]->data;

}

// Add child nodes to queue sequentially

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

if (levelNodes[i]->left) q.push(levelNodes[i]->left);

if (levelNodes[i]->right) q.push(levelNodes[i]->right);

}

}

}

int main() {

node \*root = NULL;

Breadthfs tree;

int data;

char ans;

do {

cout << "\nEnter data => ";

cin >> data;

root = tree.insert(root, data);

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

cin >> ans;

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

cout << "\nBFS Traversal: ";

tree.bfs(root);

cout << endl;

return 0;

}

**1 (B)-: 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;

}

**2(A)-: 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;

}

**2 (B)-: 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=>";

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

{

cin>>a[i];

}

// start=.......

//#pragma ompâ€¦..

mergesort(a, 0, n-1);

// stopâ€¦â€¦.

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

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

{

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

}

// Cout<<Stop-Start

return 0;

}

**3-: Min Max**

#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);

}

**4 (A) Addition**

#include <iostream>

#include <cuda\_runtime.h>

#include /usr/local/cuda/include/cuda\_runtime.h

\_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;

// 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;

}

**4 (B)-: Multiplication**

#include <cuda\_runtime.h>

#include <iostream>

\_\_global\_\_ void matmul(int\* A, int\* B, int\* C, int N) {

int Row = blockIdx.y\*blockDim.y+threadIdx.y;

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

if (Row < N && Col < N) {

int Pvalue = 0;

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

Pvalue += A[Row\*N+k] \* B[k\*N+Col];

}

C[Row\*N+Col] = Pvalue;

}

}

int main() {

int N = 512;

int size = N \* N \* sizeof(int);

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

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

cudaMallocHost(&A, size);

cudaMallocHost(&B, size);

cudaMallocHost(&C, size);

cudaMalloc(&dev\_A, size);

cudaMalloc(&dev\_B, size);

cudaMalloc(&dev\_C, size);

// Initialize matrices A and B

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

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

A[i\*N+j] = i\*N+j;

B[i\*N+j] = j\*N+i;

}

}

cudaMemcpy(dev\_A, A, size,

cudaMemcpyHostToDevice);

cudaMemcpy(dev\_B, B, size,

cudaMemcpyHostToDevice);

dim3 dimBlock(16, 16);

dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);

matmul<<<dimGrid, dimBlock>>>(dev\_A, dev\_B,

dev\_C, N);

cudaMemcpy(C, dev\_C

// Print the result

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

for (int j = 0; j < 10; j++) {

std::cout << C[i\*N+j] << " ";

}

std::cout << std::endl;

}

// Free memory

cudaFree(dev\_A);

cudaFree(dev\_B);

cudaFree(dev\_C);

cudaFreeHost(A);

cudaFreeHost(B);

cudaFreeHost(C);

return 0;