// C++ code to print\*\*\*\*\*\* BFS\*\*\*\*\*\* traversal from a given source vertex

#include <bits/stdc++.h>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph {

// No. of vertices

int V;

// Pointer to an array containing adjacency lists

vector<list<int> > adj;

public:

// Constructor

Graph(int V);

// Function to add an edge to graph

void addEdge(int v, int w);

// Prints BFS traversal from a given source s

void BFS(int s); };

Graph::Graph(int V){

this->V = V;

adj.resize(V);}

void Graph::addEdge(int v, int w){

// Add w to v’s list.

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

void Graph::BFS(int s)

{// Mark all the vertices as not visited

vector<bool> visited;

visited.resize(V, false);

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

while (!queue.empty()) {

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (auto adjacent : adj[s]) {

if (!visited[adjacent]) {

visited[adjacent] = true;

queue.push\_back(adjacent);

} } } }

// Driver code

int main()

{ // Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;}

// C++ program to print DFS

// traversal for a given

// graph

#include <bits/stdc++.h>

using namespace std;

class Graph {

// A function used by DFS

void DFSUtil(int v);

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// prints DFS traversal of the complete graph

void DFS();

};

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to vâ€™s list.

}

void Graph::DFSUtil(int v)

{

// Mark the current node as visited and print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i);

}

// The function to do DFS traversal. It uses recursive

// DFSUtil()

void Graph::DFS()

{

// Call the recursive helper function to print DFS

// traversal starting from all vertices one by one

for (auto i : adj)

if (visited[i.first] == false)

DFSUtil(i.first);

}

// Driver's Code

int main()

{

// Create a graph given in the above diagram

Graph g;

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal \n";

// Function call

g.DFS();

return 0;

}

\*\*\*\*\*\*\*CODE FOR ONLY MIN MAX\*\*\*\*\*\*\*

#include<bits/stdc++.h>

using namespace std;

// Python Implementation of the above approach

void minMax(vector<int>&arr){

// Initialize the min\_value

// and max\_value to 0

int min\_value = 0;

int max\_value = 0;

int n = arr.size();

// Sort array before calculating

// min and max value

sort(arr.begin(),arr.end());

int j = n - 1;

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

{

// All elements except

// rightmost will be added

min\_value += arr[i];

// All elements except

// leftmost will be added

max\_value += arr[j];

j -= 1;

}

// Output: min\_value and max\_value

cout<<min\_value<<" "<<max\_value<<endl;

}

// Driver Code

int main(){

vector<int>arr = {10, 9, 8, 7, 6, 5};

vector<int>arr1 = {100, 200, 300, 400, 500};

minMax(arr);

minMax(arr1);

}

\*\*\*\*\*\*\*MIN MAX WITH AVERAGE SUM\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <numeric>

#include <algorithm>

#include <thread>

template <typename T>

T parallel\_reduction(const std::vector<T>& data, T operation(const std::vector<T>&))

{

// Determine the number of available CPU cores

const unsigned int num\_cores = std::thread::hardware\_concurrency();

// Split the data into smaller subsets

const std::size\_t chunk\_size = data.size() / num\_cores;

std::vector<std::vector<T>> chunks(num\_cores);

for (unsigned int i = 0; i < num\_cores; ++i)

{

const auto first = data.begin() + (i \* chunk\_size);

const auto last = (i == num\_cores - 1) ? data.end() : first + chunk\_size;

chunks[i] = std::vector<T>(first, last);

}

// Perform the operation on each chunk in parallel

std::vector<T> results(num\_cores);

std::vector<std::thread> threads(num\_cores);

for (unsigned int i = 0; i < num\_cores; ++i)

{

threads[i] = std::thread([&results, &chunks, operation, i]() {

results[i] = operation(chunks[i]);

});

}

// Wait for all threads to finish

for (auto& thread : threads)

{

thread.join();

}

// Combine the results iteratively until the final result is obtained

while (results.size() > 1)

{

std::vector<T> new\_results;

const std::size\_t size = results.size();

for (std::size\_t i = 0; i < size; i += 2)

{

if (i + 1 < size)

{

new\_results.push\_back(operation({results[i], results[i + 1]}));

}

else

{

new\_results.push\_back(results[i]);

}

}

results = new\_results;

}

// Return the final result

return results[0];

}

template <typename T>

T find\_min(const std::vector<T>& data)

{

return \*std::min\_element(data.begin(), data.end());

}

template <typename T>

T find\_max(const std::vector<T>& data)

{

return \*std::max\_element(data.begin(), data.end());

}

template <typename T>

T find\_sum(const std::vector<T>& data)

{

return std::accumulate(data.begin(), data.end(), T(0));

}

template <typename T>

T find\_average(const std::vector<T>& data)

{

return find\_sum(data) / static\_cast<T>(data.size());

}

int main()

{

std::vector<int> data = {1, 5, 3, 9, 2, 7, 4, 6, 8};

std::cout << "Min: " << parallel\_reduction(data, find\_min<int>) << std::endl;

std::cout << "Max: " << parallel\_reduction(data, find\_max<int>) << std::endl;

std::cout << "Sum: " << parallel\_reduction(data, find\_sum<int>) << std::endl;

std::cout << "Average: " << parallel\_reduction(data, find\_average<int>) << std::endl;

return 0;

}

\*\*\*\*\*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=>\n";

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

{

cout<<a[i]<<endl;

}

return 0;

}

\*\*\*\*\*\*BUBBLE SORT ODD EVEN TRANSPOSITION\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <algorithm>

void odd\_even\_transposition\_sort(std::vector<int>& data)

{

const int n = data.size();

bool sorted = false;

while (!sorted)

{

sorted = true;

// Odd phase

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

{

if (data[i] > data[i + 1])

{

std::swap(data[i], data[i + 1]);

sorted = false;

}

}

// Even phase

for (int i = 0; i < n - 1; i += 2)

{

if (data[i] > data[i + 1])

{

std::swap(data[i], data[i + 1]);

sorted = false;

}

}

}

}

int main()

{

std::vector<int> data = {9, 2, 7, 4, 6, 8, 1, 5, 3};

std::cout << "Before sorting: ";

for (const auto& num : data)

{

std::cout << num << " ";

}

std::cout << std::endl;

odd\_even\_transposition\_sort(data);

std::cout << "After sorting: ";

for (const auto& num : data)

{

std::cout << num << " ";

}

std::cout << std::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;

}

\*\*\*\*\*\*\*\*cuda ADDITION OF 2 VECTORS\*\*\*\*\*\*\*

#include <iostream>

#include <vector>

#include <cuda\_runtime.h>

\_global\_ void vectorAddition(const int\* a, const int\* b, int\* result, int size)

{

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

if (tid < size)

{

result[tid] = a[tid] + b[tid];

}

}

void performVectorAddition(const std::vector<int>& a, const std::vector<int>& b, std::vector<int>& result)

{

// Size of the vectors

const int size = a.size();

// Allocate device memory

int\* dev\_a;

int\* dev\_b;

int\* dev\_result;

cudaMalloc((void\*\*)&dev\_a, size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_b, size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_result, size \* sizeof(int));

// Copy data from host to device

cudaMemcpy(dev\_a, a.data(), size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, b.data(), size \* sizeof(int), cudaMemcpyHostToDevice);

// Set up grid and block dimensions

const int threadsPerBlock = 256;

const int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

// Launch the CUDA kernel

vectorAddition<<<blocksPerGrid, threadsPerBlock>>>(dev\_a, dev\_b, dev\_result, size);

// Copy the result back to the host

cudaMemcpy(result.data(), dev\_result, size \* sizeof(int), cudaMemcpyDeviceToHost);

// Free device memory

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_result);

}

int main()

{

// Define the input vectors

std::vector<int> a = {1, 2, 3, 4, 5};

std::vector<int> b = {6, 7, 8, 9, 10};

const int size = a.size();

// Define the result vector

std::vector<int> result(size);

// Perform the vector addition

performVectorAddition(a, b, result);

// Print the result

std::cout << "Result: ";

for (const auto& value : result)

{

std::cout << value << " ";

}

std::cout << std::endl;

return 0;

\*\*\*\*\*\*\*\*\*\*cuda matrix multiplication\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

#include <cuda\_runtime.h>

// CUDA kernel for matrix multiplication

\_global\_ void matrixMultiplication(const int\* A, const 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 sum = 0;

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

sum += A[row \* N + i] \* B[i \* N + col];

}

C[row \* N + col] = sum;

}

}

// Function to initialize a matrix with random values

void initializeMatrix(int\* matrix, int size) {

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

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

matrix[i \* size + j] = rand() % 10;

}

}

}

// Function to print a matrix

void printMatrix(const int\* matrix, int size) {

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

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

std::cout << matrix[i \* size + j] << " ";

}

std::cout << std::endl;

}

}

int main() {

const int N = 4; // Matrix size

// Allocate memory for matrices on the host

int\* A = new int[N \* N];

int\* B = new int[N \* N];

int\* C = new int[N \* N];

// Initialize matrices with random values

initializeMatrix(A, N);

initializeMatrix(B, N);

// Allocate memory for matrices on the device

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

cudaMalloc((void\*\*)&d\_A, N \* N \* sizeof(int));

cudaMalloc((void\*\*)&d\_B, N \* N \* sizeof(int));

cudaMalloc((void\*\*)&d\_C, N \* N \* sizeof(int));

// Copy matrices from host to device

cudaMemcpy(d\_A, A, N \* N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, N \* N \* sizeof(int), cudaMemcpyHostToDevice);

// Define the grid and block dimensions

dim3 blockDims(16, 16);

dim3 gridDims((N + blockDims.x - 1) / blockDims.x, (N + blockDims.y - 1) / blockDims.y);

// Launch the matrix multiplication kernel

matrixMultiplication<<<gridDims, blockDims>>>(d\_A, d\_B, d\_C, N);

// Copy the result matrix from device to host

cudaMemcpy(C, d\_C, N \* N \* sizeof(int), cudaMemcpyDeviceToHost);

// Print the matrices and the result

std::cout << "Matrix A:" << std::endl;

printMatrix(A, N);

std::cout << "Matrix B:" << std::endl;

printMatrix(B, N);

std::cout << "Matrix C (Result):" << std::endl;

printMatrix(C, N);

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

delete[] A;

delete[] B;

delete[] C;

return 0;

}

}