**HPC**

**1A BFS**

#include <iostream>

#include <queue>

#include <omp.h>

using namespace std;

// Define the structure for a binary tree node

struct Node {

int data;

Node\* left;

Node\* right;

};

// Function to create a new node

Node\* createNode(int data) {

Node\* newNode = new Node;

newNode->data = data;

newNode->left = NULL; // Replace nullptr with NULL

newNode->right = NULL; // Replace nullptr with NULL

return newNode;

}

// Function to perform breadth-first search traversal on a binary tree

void BFS(Node\* root) {

if (root == NULL) return; // Replace nullptr with NULL

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

// Get the current size of the queue

int size = q.size();

// Parallelize the traversal of nodes at the current level

#pragma omp parallel for

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

// Dequeue a node from the queue

Node\* current;

#pragma omp critical

{

current = q.front();

q.pop();

}

// Process the current node (print its data, etc.)

cout << current->data << " ";

// Enqueue the left child if it exists

if (current->left != NULL) { // Replace nullptr with NULL

#pragma omp critical

{

q.push(current->left);

}

}

// Enqueue the right child if it exists

if (current->right != NULL) { // Replace nullptr with NULL

#pragma omp critical

{

q.push(current->right);

}

}

}

}

}

int main() {

// Construct a binary tree

Node\* root = createNode(5);

root->left = createNode(3);

root->left->left = createNode(2);

root->left->left->right = createNode(1);

root->right = createNode(7);

root->right->right = createNode(8);

// Perform BFS traversal in parallel using OpenMP

BFS(root);

return 0;

}

**1B DFS**

#include<iostream>

#include<vector>

#include<omp.h>

using namespace std;

const int N = 1e5+2;

bool vis[N];

vector<int> adj[N];

void dfs(int node)

{

vis[node]=1;

cout<<node<<" ";

vector<int> :: iterator it;

for(it = adj[node].begin(); it != adj[node].end(); it++)

{

if(vis[\*it]);

else

{

dfs(\*it);

}

}

}

int main()

{

int n,m;

cout<<"Enter the pair of nodes : \n";

cin>>n>>m;

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

{

vis[i]=false;

}

int x,y;

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

{

cin>>x>>y;

adj[x].push\_back(y);

adj[y].push\_back(x);

}

dfs(1);

return 0;

}

**2A 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]<<" ";

}

return 0;

}

**2B 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;

{

{

mergesort(a,i,mid);

}

{

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);

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

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

{

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

}

return 0;

}

**3 MIN, MAX, SUM, AVERAGE**

#include <iostream>

#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 << "\n 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 << "\n 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 << "\n 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 << "\n 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];

}

min\_reduction(arr, n);

max\_reduction(arr, n);

sum\_reduction(arr, n);

average\_reduction(arr, n);

}

**4A VECTOR 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;

}

**4B MATRIX 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;