ASSIGNMENT 1A

#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\* 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);

}

}

// add a default return statement

return root;

}

void printTreeHelper(node\* root, int space)

{ if (root == NULL)

return;

space += 10;

printTreeHelper(root->right, space);

cout<<endl;

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

cout<<" ";

cout<<root->data<<"\n";

printTreeHelper(root->left, space);

}

void printTree(node\* root)

{

printTreeHelper(root, 0);

}

Void Breadthfs::bfs(node \*head)

{queue<node\*> q;

q.push(head);

int qSize;

while (!q.empty())

{

qSize = q.size();

#pragma omp parallel for

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

{

node\* currNode;

#pragma omp critical

{

currNode = q.front();

q.pop();

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

}

#pragma omp critical

{

if(currNode->left)

q.push(currNode->left);

if(currNode->right)

q.push(currNode->right);

}

}

}

}

int main()

{

node \*root=NULL;

int data;

char ans;

do

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

cin>>data;

root=Breadthfs().insert(root,data);

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

cin>>ans;

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

cout<<"Tree:\n";

printTree(root);

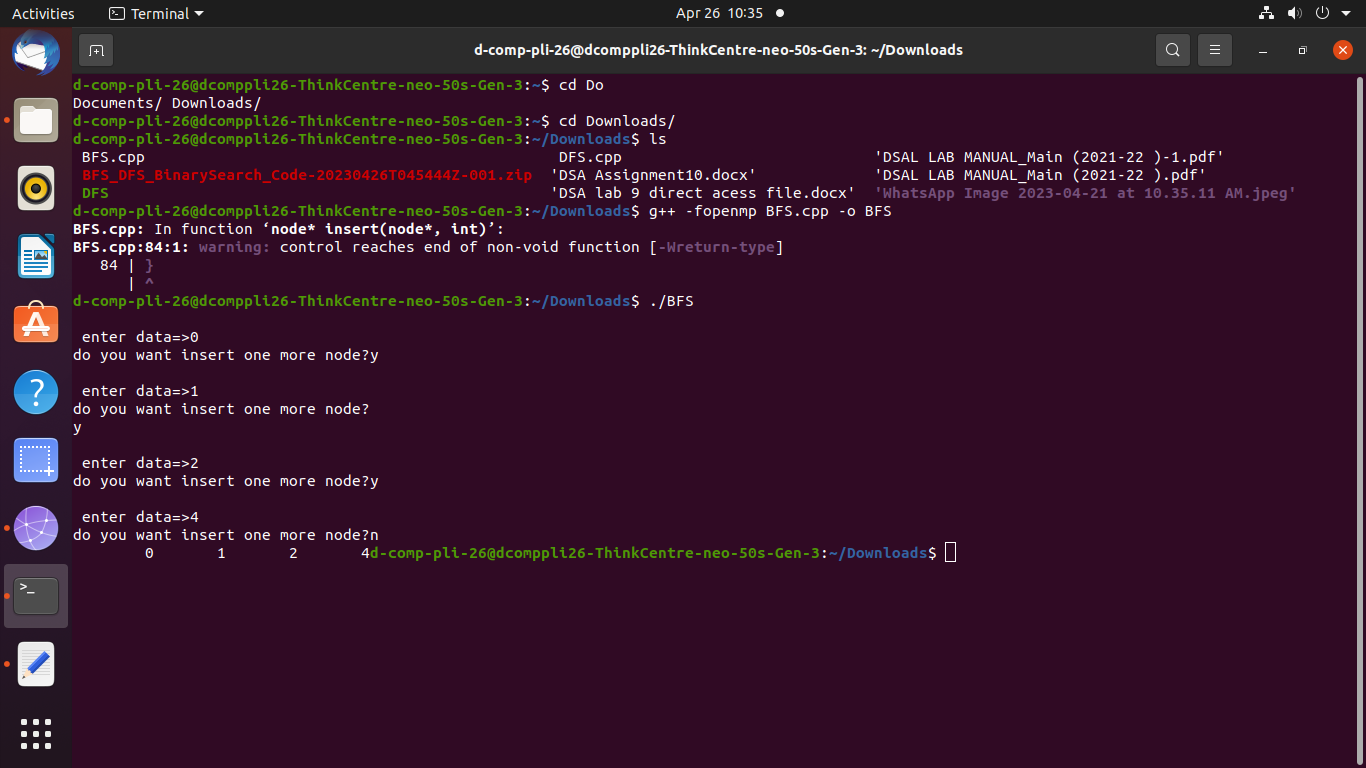
cout<<"\nBreadth First Search:";

Breadthfs().bfs(root);

cout<<"\n";

return 0;

}



ASSIGNMENT 1B

#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();

if (!visited[curr\_node]) {

visited[curr\_node] = true;

s.pop();

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 nodes, no. of edges, and starting node of graph:\n";

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

cout << "Enter pairs of nodes and edges:\n";

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

int u, v;

cin >> u >> v;

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

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

}

#pragma omp parallel for

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

visited[i] = false;

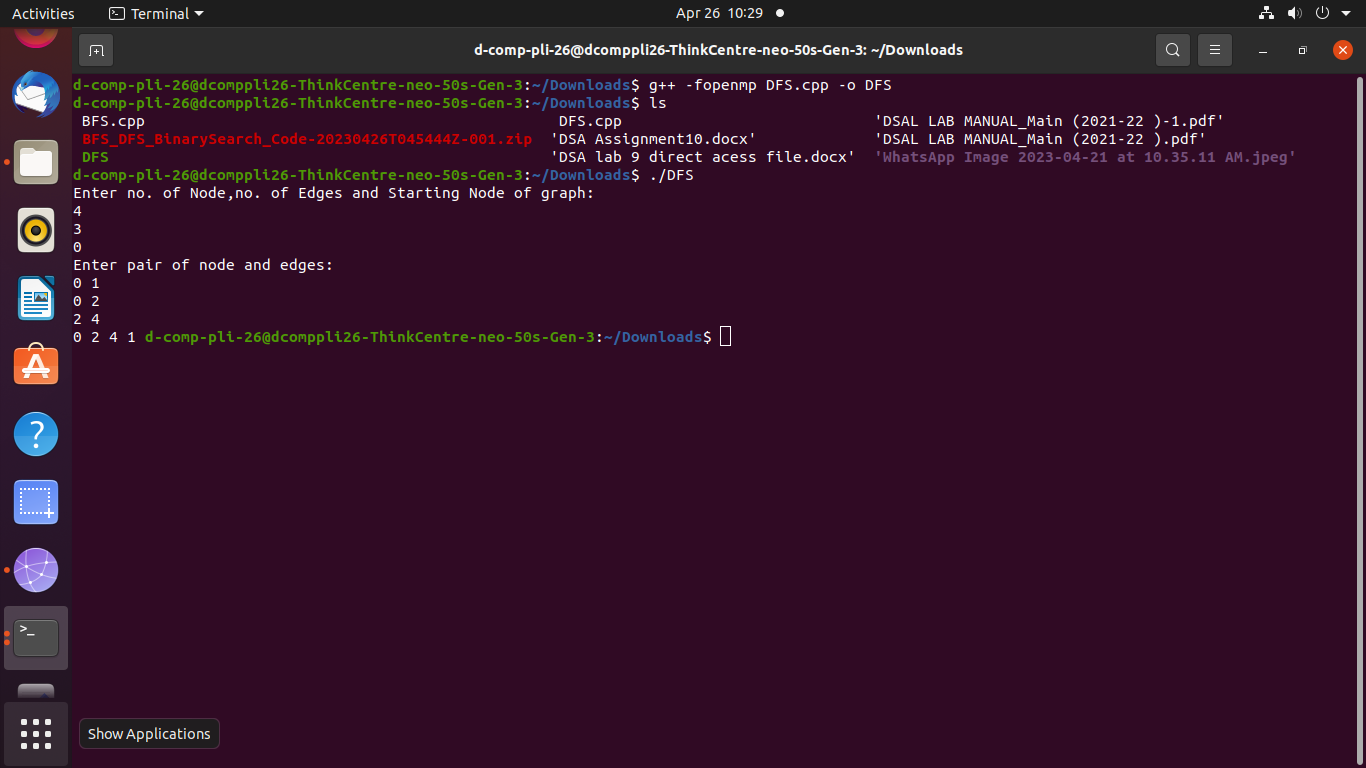
}

dfs(start\_node);

cout << endl; // add new line after output

return 0;

}



ASSIGNMENT 2A

#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\* 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);

}

}

// add a default return statement

return root;

}

void printTreeHelper(node\* root, int space)

{

if (root == NULL)

return;

space += 10;

printTreeHelper(root->right, space);

cout<<endl;

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

cout<<" ";

cout<<root->data<<"\n";

printTreeHelper(root->left, space);

}

void printTree(node\* root)

{

printTreeHelper(root, 0);

}

void Breadthfs::bfs(node \*head)

{

queue<node\*> q;

q.push(head);

int qSize;

while (!q.empty())

{

qSize = q.size();

#pragma omp parallel for

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

{

node\* currNode;

#pragma omp critical

{

currNode = q.front();

q.pop();

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

}

#pragma omp critical

{

if(currNode->left)

q.push(currNode->left);

if(currNode->right)

q.push(currNode->right);

}

}

}

}

int main()

{

node \*root=NULL;

int data;

char ans;

do

{

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

cin>>data;

root=Breadthfs().insert(root,data);

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

cin>>ans;

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

cout<<"Tree:\n";

printTree(root);

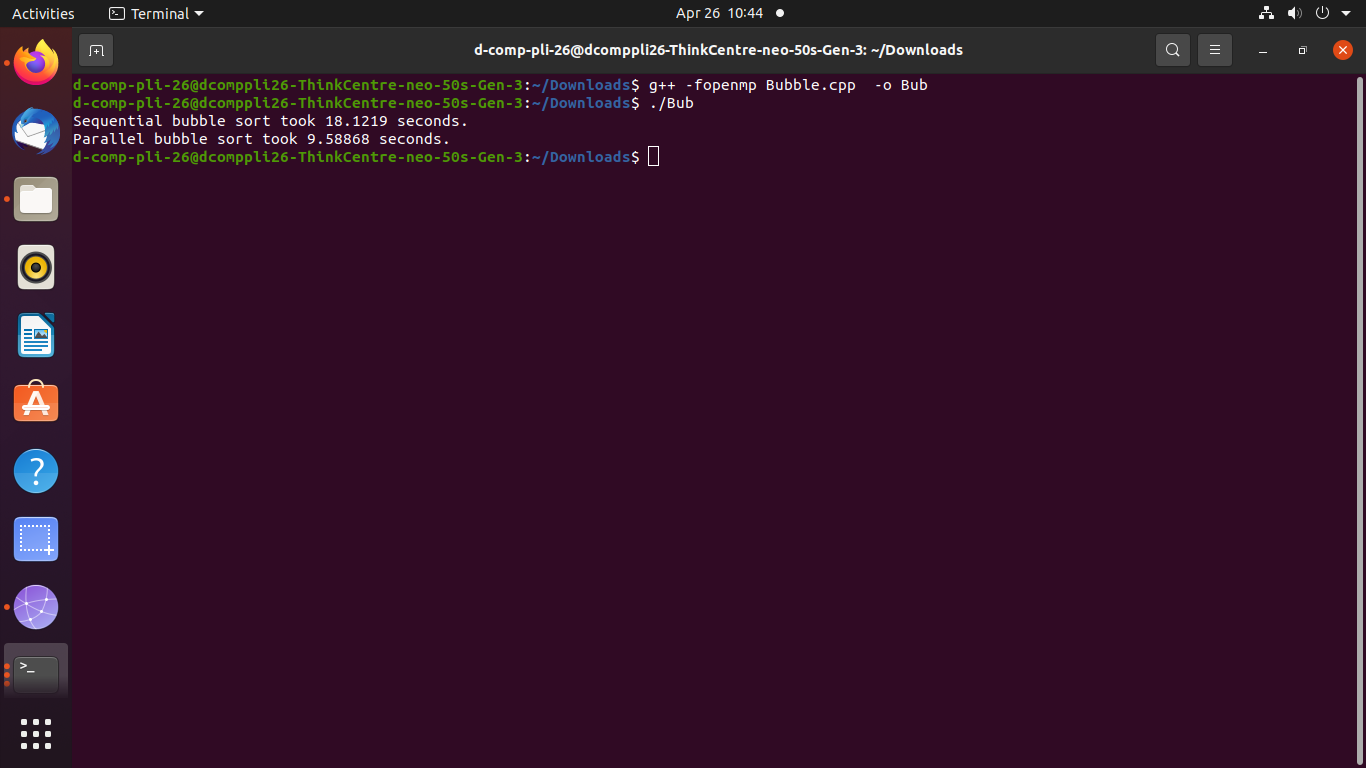
cout<<"\nBreadth First Search:";

Breadthfs().bfs(root);

cout<<"\n";

return 0;

}



ASSIGNMENT 2B

#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();

if (!visited[curr\_node]) {

visited[curr\_node] = true;

s.pop();

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 nodes, no. of edges, and starting node of graph:\n";

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

cout << "Enter pairs of nodes and edges:\n";

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

int u, v;

cin >> u >> v;

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

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

}

#pragma omp parallel for

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

visited[i] = false;

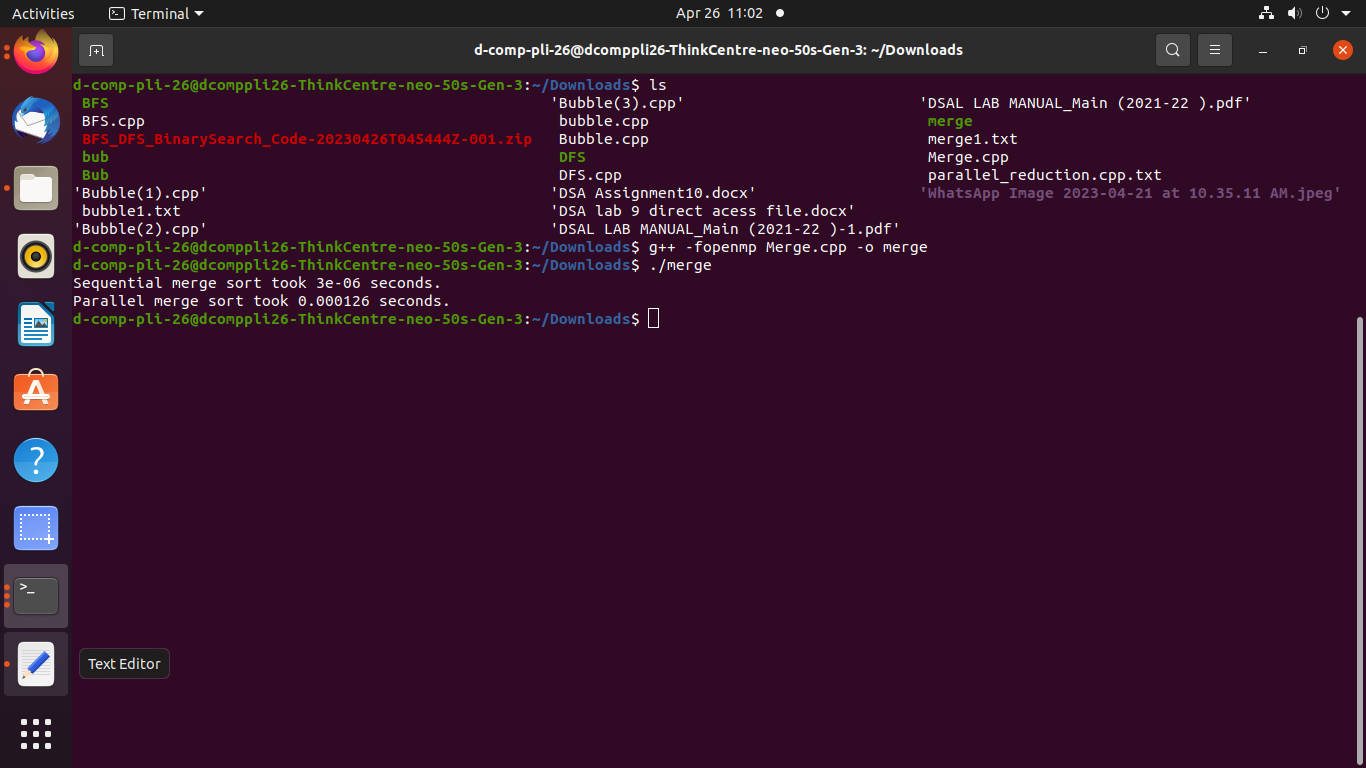
}

dfs(start\_node);

cout << endl; // add new line after output

return 0;

}



ASSIGNMENT 3

#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

using namespace std;

int min\_reduction(vector<int>& arr) {

int min\_value = INT\_MAX;

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

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

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

min\_value = arr[i];

}

}

return min\_value;

}

int max\_reduction(vector<int>& arr) {

int max\_value = INT\_MIN;

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

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

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

max\_value = arr[i];

}

}

return max\_value;

}

int sum\_reduction(vector<int>& arr) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += arr[i];}

return sum;

}

double average\_reduction(vector<int>& arr) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

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

sum += arr[i];

}

return (double)sum / arr.size();

}

int main() {

vector<int> arr;

arr.push\_back(5);

arr.push\_back(2);

arr.push\_back(9);

arr.push\_back(1);

arr.push\_back(7);

arr.push\_back(6);

arr.push\_back(8);

arr.push\_back(3);

arr.push\_back(4);

int min\_value = min\_reduction(arr);

int max\_value = max\_reduction(arr);

int sum = sum\_reduction(arr);

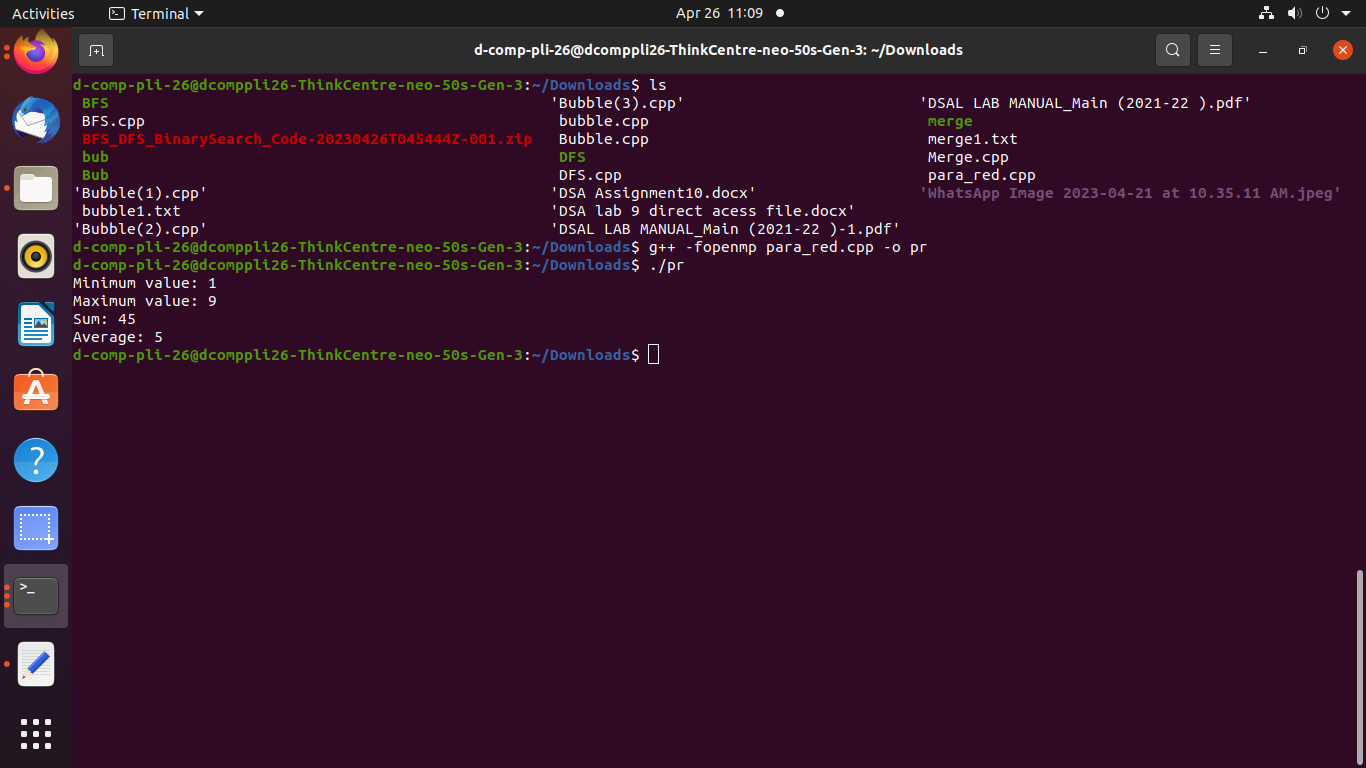
double average = average\_reduction(arr);

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

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

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

cout << "Average: " << average << endl;

}

ASSIGNMENT 4A

#include <iostream>

#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;

ASSIGNMENT 4B

#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);

63

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

64

// 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

MINI PROJECT HPC

#include <iostream>

#include <algorithm>

#include <mpi.h>

using namespace std;

// Function to partition the array

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high- 1; j++) {

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return (i + 1);

}

// Function to perform quicksort on the partition

void quicksort(int arr[], int low, int high) {

if (low < high) {

int pivot = partition(arr, low, high);

quicksort(arr, low, pivot - 1);

quicksort(arr, pivot + 1, high);

}

}

int main(int argc, char \*argv[]) {

int rank, size;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int n = 100;

int\* arr = new int[n];

int\* recvbuf = new int[n];

int\* sendbuf = new int[n];

// Fill the array with random values

if (rank == 0) {

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

arr[i] = rand() % 100;

}

}

// Divide the array into equal-sized partitions for each process

int sub\_arr\_size = n / size;

int\* sub\_arr = new int[sub\_arr\_size];

MPI\_Scatter(arr, sub\_arr\_size, MPI\_INT, sub\_arr, sub\_arr\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Sort the partition using quicksort

quicksort(sub\_arr, 0, sub\_arr\_size - 1);

// Gather the sorted partitions from each process

MPI\_Gather(sub\_arr, sub\_arr\_size, MPI\_INT, recvbuf, sub\_arr\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

// Print the sorted array

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

cout << recvbuf[i] << " ";

}

cout << endl;

// Measure the execution time of the parallel quicksort algorithm

double start\_time = MPI\_Wtime();

// Perform the above steps again

double end\_time = MPI\_Wtime();

double parallel\_execution\_time = end\_time - start\_time;

// Measure the execution time of the sequential quicksort algorithm

start\_time = MPI\_Wtime();

quicksort(arr, 0, n - 1);

end\_time = MPI\_Wtime();

double sequential\_execution\_time = end\_time - start\_time;

// Calculate speedup and efficiency

double speedup = sequential\_execution\_time / parallel\_execution\_time;

double efficiency = speedup / size;

cout << "Sequential execution time: " << sequential\_execution\_time << endl;

cout << "Parallel execution time: " << parallel\_execution\_time << endl;

cout << "Speedup: " << speedup << endl;

cout << "Efficiency: " << efficiency << endl;

}

MPI\_Finalize();

return 0;

}

