HPC PRACTICALS

Installation commands of g++ and openmp-

sudo apt-get install g++

sudo apt-get install libomp.dev

1) DFS and BFS using parallel programming and OpenMP

Steps-

* Create file using command-

Cat > filename.cpp

* Write code in terminal and press ctrl D to save the code
* To compile file use command-

g++ -o filename –fopenmp filename.cpp

* Run file using command-

./filename

DFS Code-

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

const int MAXN = 1e5;

vector<int> adj[MAXN+5]; // adjacency list

bool visited[MAXN+5]; // mark visited nodes

void dfs(int node) {

visited[node] = true;

#pragma omp parallel for

for (int i = 0; i < adj[node].size(); i++) {

int next\_node = adj[node][i];

if (!visited[next\_node]) {

dfs(next\_node);

}

}

}

int main() {

cout << "Please enter nodes and edges";

int n, m; // number of nodes and edges

cin >> n >> m;

for (int i = 1; i <= m; i++) {

int u, v; // edge between u and v

cin >> u >> v;

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

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

}

int start\_node; // start node of DFS

cin >> start\_node;

dfs(start\_node);

// Print visited nodes

for (int i = 1; i <= n; i++) {

if (visited[i]) {

cout << i << " ";

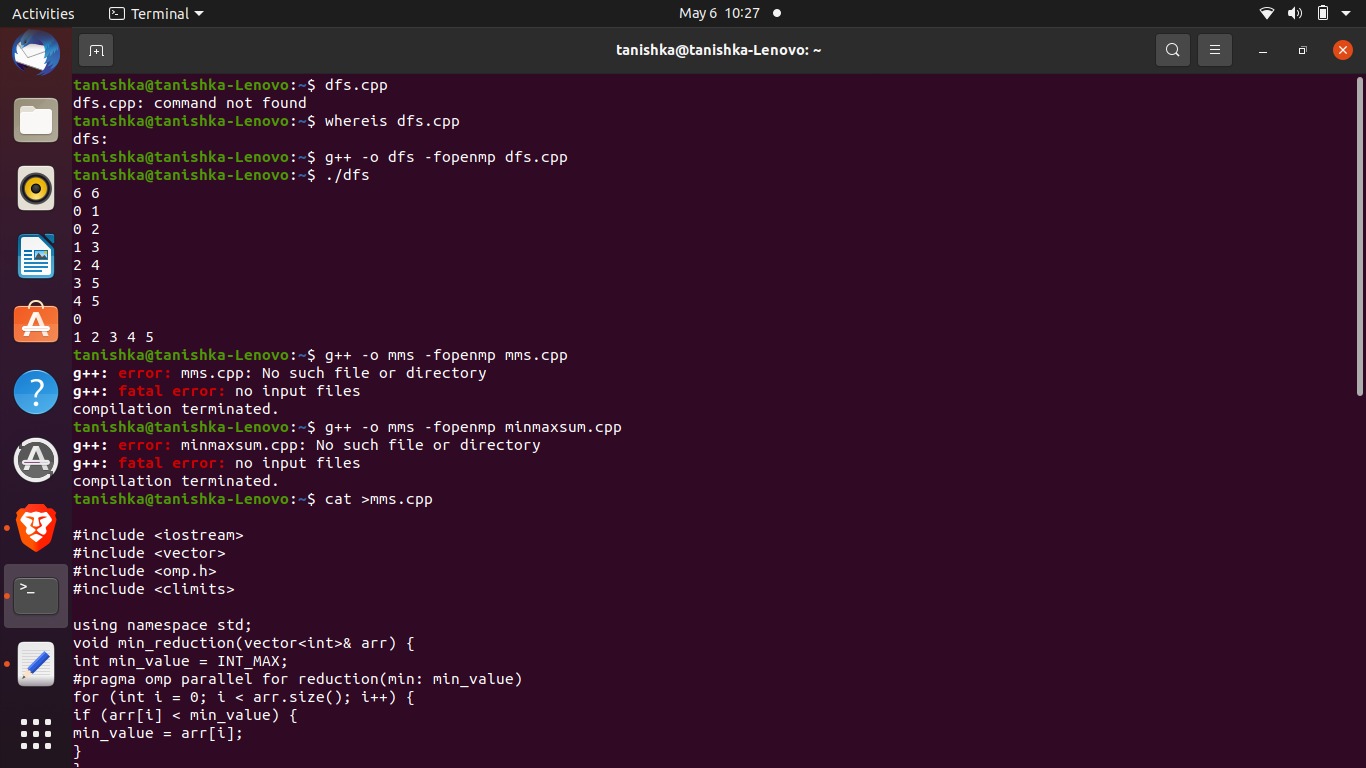
}

}

cout << endl;

return 0;

}



BFS Code

#include <iostream>

#include <queue>

#include <vector>

#include <omp.h>

using namespace std;

int main() {

int num\_vertices, num\_edges, source;

cin >> num\_vertices >> num\_edges >> source;

vector<vector<int>> adj\_list(num\_vertices + 1);

for (int i = 0; i < num\_edges; i++) {

int u, v;

cin >> u >> v;

adj\_list[u].push\_back(v);

adj\_list[v].push\_back(u);

}

queue<int> q;

vector<bool> visited(num\_vertices + 1, false);

q.push(source);

visited[source] = true;

while (!q.empty()) {

int curr\_vertex = q.front();

q.pop();

cout << curr\_vertex << " ";

#pragma omp parallel for shared(adj\_list, visited, q) schedule(dynamic)

for (int i = 0; i < adj\_list[curr\_vertex].size(); i++) {

int neighbour = adj\_list[curr\_vertex][i];

if (!visited[neighbour]) {

visited[neighbour] = true;

q.push(neighbour);

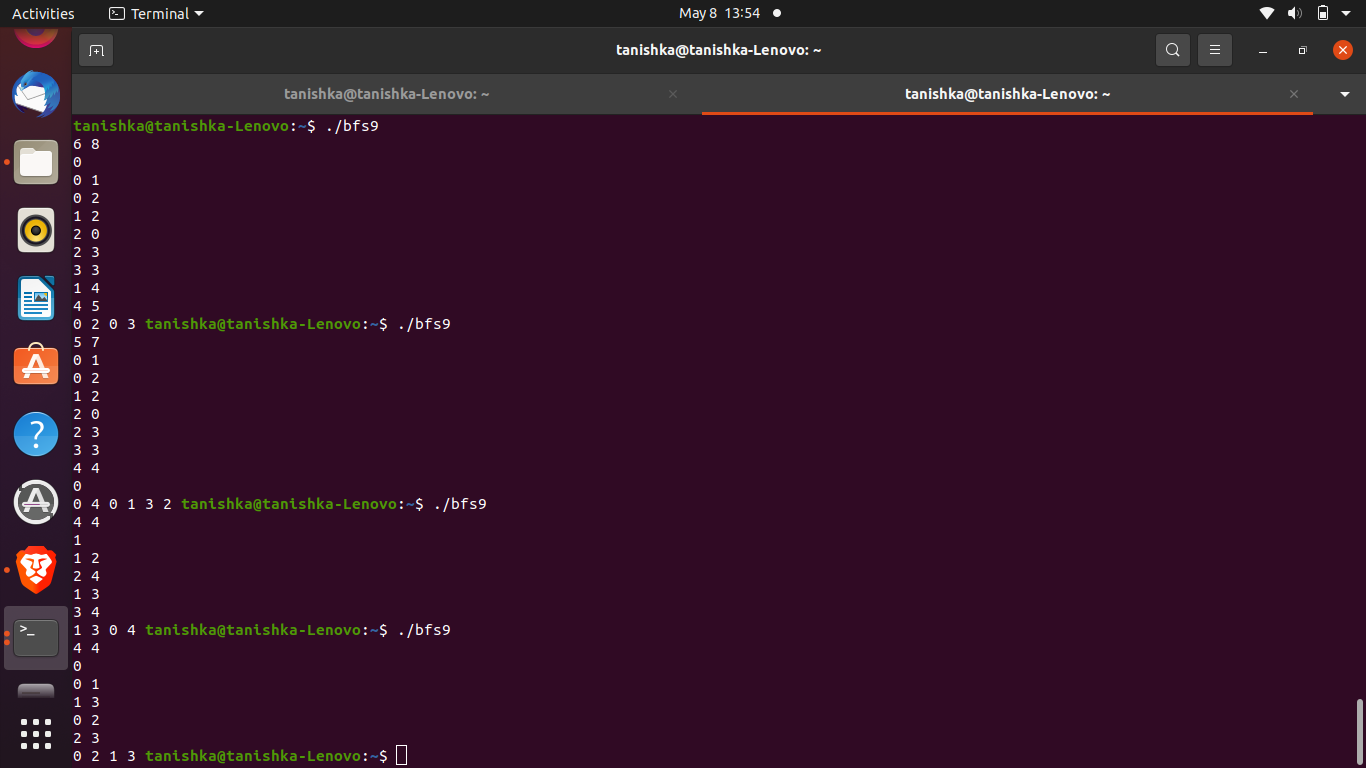
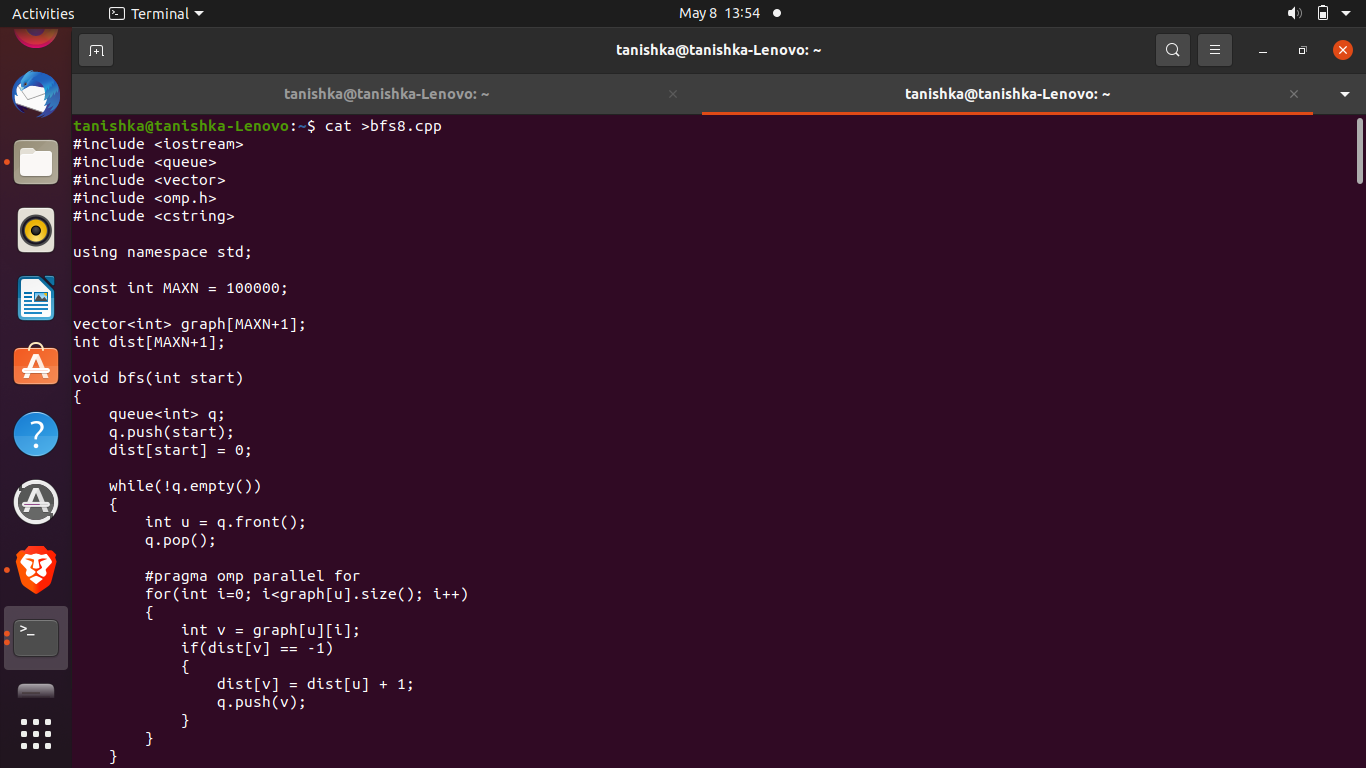
}

}

}

return 0;

}



2) Merge sort and bubble sort using parallel programming and OpenMP

(use steps given in practical 1)

Merge sort: recursive algorithm, uses divide and conquer and merge, time complexity: O(nlog n)

Merge sort is a sorting algorithm that works by dividing the unsorted list into smaller sublists, sorting those sublists, and then merging them back together into a sorted list.

MergeSort Code-

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

void merge(vector<int>& arr, int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

vector<int> L(n1), R(n2);

for (i = 0; i < n1; i++) {

L[i] = arr[l + i];

}

for (j = 0; j < n2; j++) {

R[j] = arr[m + 1 + j];

}

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k++] = L[i++];

} else {

arr[k++] = R[j++];

}

}

}

void merge\_sort(vector<int>& arr, int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

#pragma omp task

merge\_sort(arr, l, m);

#pragma omp task

merge\_sort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

void parallel\_merge\_sort(vector<int>& arr) {

#pragma omp parallel

{

#pragma omp single

merge\_sort(arr, 0, arr.size() - 1);

}

}

int main() {

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

double start, end;

// Measure performance of sequential merge sort

start = omp\_get\_wtime();

merge\_sort(arr, 0, arr.size() - 1);

end = omp\_get\_wtime();

cout << "Sequential merge sort time: " << end - start <<endl;

// Measure performance of parallel merge sort

arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

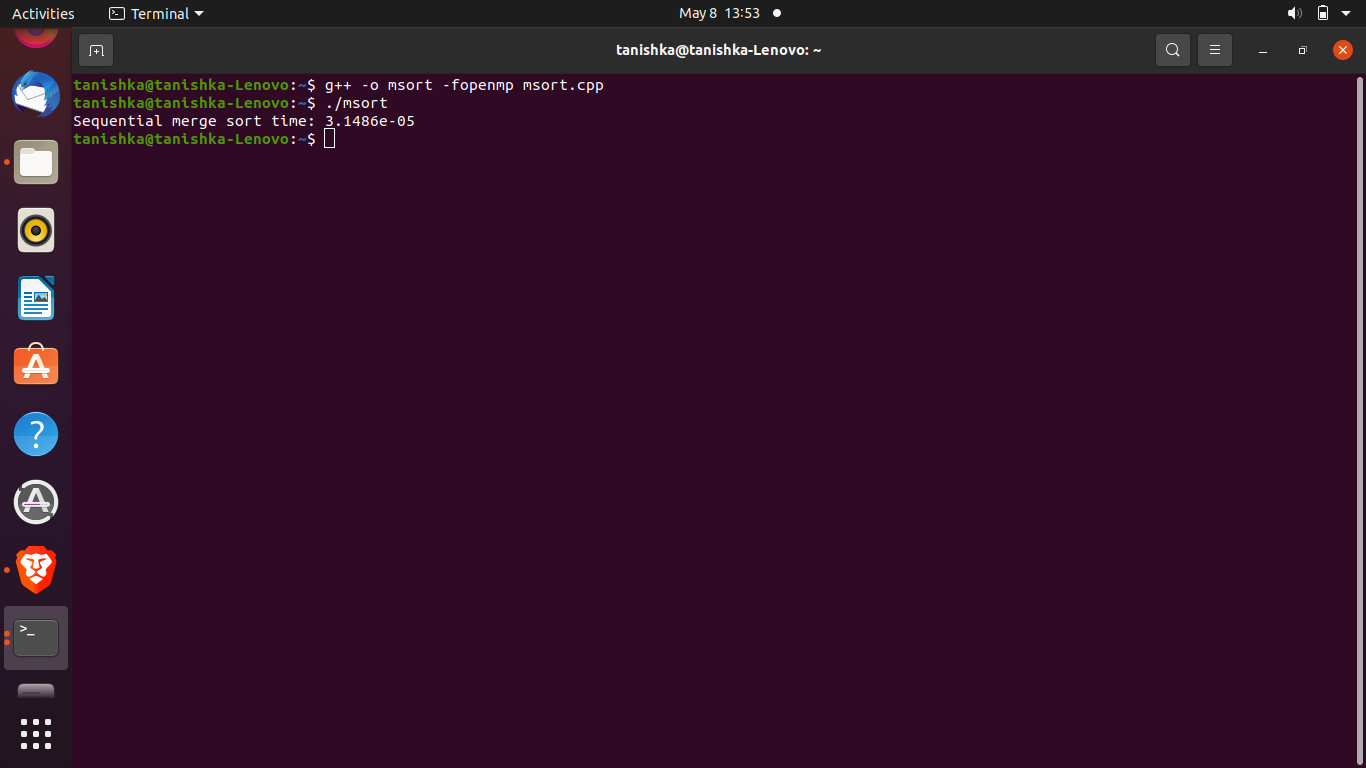
start = omp\_get\_wtime();

parallel\_merge\_sort(arr);

end = omp\_get\_wtime();

return 0;

}



Bubble Sort Code-

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

void bubble\_sort\_odd\_even(vector<int>& arr) {

bool isSorted = false;

while (!isSorted) {

isSorted = true;

#pragma omp parallel for

for (int i = 0; i < arr.size() - 1; i += 2) {

if (arr[i] > arr[i + 1]) {

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

isSorted = false;

}

}

#pragma omp parallel for

for (int i = 1; i < arr.size() - 1; i += 2) {

if (arr[i] > arr[i + 1]) {

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

isSorted = false;

}

}

}

}

int main() {

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

double start, end;

// Measure performance of parallel bubble sort using odd-

//even transposition

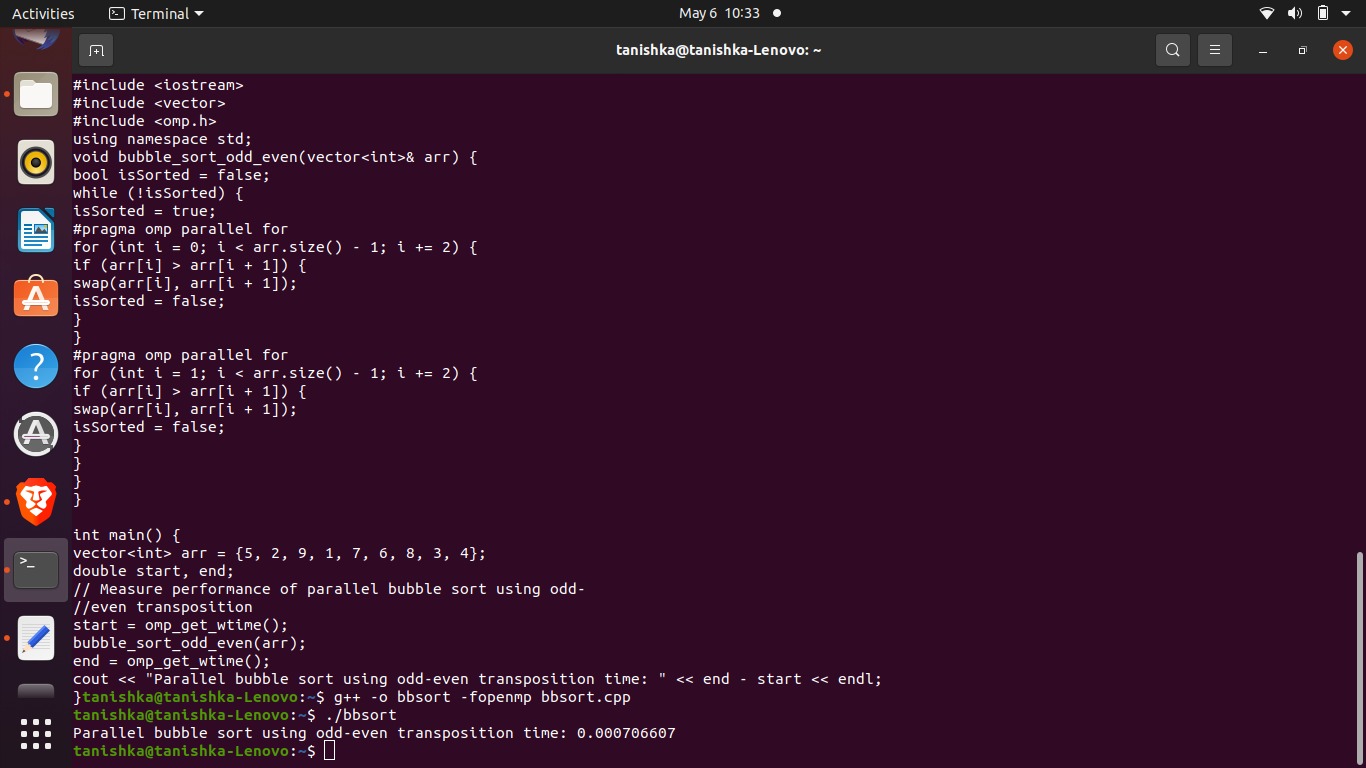
start = omp\_get\_wtime();

bubble\_sort\_odd\_even(arr);

end = omp\_get\_wtime();

cout << "Parallel bubble sort using odd-even transposition time: " << end - start << endl;

}



3) Min, Max, Sum and Avg using parallel reduction

Code-

#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

using namespace std;

void 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];

}

}

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

}

void 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];

}

}

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

}

void 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];

}

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

}

void 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];

}

cout << "Average: " << (double)sum / arr.size() << endl;

}

int main() {

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

min\_reduction(arr);

max\_reduction(arr);

sum\_reduction(arr);

average\_reduction(arr);

}

