**Problem Practical 1 Problem Statement**: Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

class Graph

{

public:

    int V;

    vector<vector<int>> adj;

    Graph(int vertices)

    {

        V = vertices;

        adj.resize(V);

    }

    void addEdge(int u, int v)

    {

        // Undirected Graph

        adj[u].push\_back(v);

        adj[v].push\_back(u);

    }

    void parallelBFS(int start)

    {

        vector<bool> visited(V, false);

        queue<int> q;

        visited[start] = true;

        q.push(start);

        cout << "Parallel BFS starting from node " << start << ":\n";

        while (!q.empty())

        {

            int levelSize = q.size();

#pragma omp parallel for

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

            {

                int current;

#pragma omp critical

                {

                    if (!q.empty())

                    {

                        current = q.front();

                        q.pop();

                        cout << "Thread " << omp\_get\_thread\_num() << " visiting " << current << endl;

                    }

                }

#pragma omp parallel for

                for (int j = 0; j < adj[current].size(); ++j)

                {

                    int neighbor = adj[current][j];

#pragma omp critical

                    {

                        if (!visited[neighbor])

                        {

                            visited[neighbor] = true;

                            q.push(neighbor);

                        }

                    }

                }

            }

        }

    }

    void parallelDFSUtil(int v, vector<bool> &visited)

    {

#pragma omp critical

        {

            visited[v] = true;

            cout << "Thread " << omp\_get\_thread\_num() << " visiting " << v << endl;

        }

        for (int u : adj[v])

        {

            if (!visited[u])

            {

#pragma omp task firstprivate(u)

                {

                    parallelDFSUtil(u, visited);

                }

            }

        }

    }

    void parallelDFS(int start)

    {

        vector<bool> visited(V, false);

        cout << "Parallel DFS starting from node " << start << ":\n";

#pragma omp parallel

        {

#pragma omp single

            {

                parallelDFSUtil(start, visited);

            }

        }

    }

};

int main()

{

    int V;

    cout << "Enter number of nodes: ";

    cin >> V;

    Graph g(V);

    int E;

    cout << "Enter number of edges: ";

    cin >> E;

    cout << "Enter " << E << " edges (u v):\n";

    for (int i = 0; i < E; ++i)

    {

        int u, v;

        cin >> u >> v;

        g.addEdge(u, v);

    }

    int startNode;

    cout << "Enter start node for BFS/DFS: ";

    cin >> startNode;

    double start\_time = omp\_get\_wtime();

    g.parallelBFS(startNode);

    double end\_time = omp\_get\_wtime();

    cout << "BFS Execution time: " << (end\_time - start\_time) << " seconds\n\n";

    start\_time = omp\_get\_wtime();

    g.parallelDFS(startNode);

    end\_time = omp\_get\_wtime();

    cout << "DFS Execution time: " << (end\_time - start\_time) << " seconds\n";

    return 0;

}  
  
Steps to execute the code:

1. Open the cmd in the folder where the code is present
2. Run the command g++ -fopenmp file\_name.cpp -o file\_name
3. Run the command .\file\_name

Output:

C:\Users\Shravan\OneDrive\Desktop\Engineering Degree Stuff\4th Year Stuff\8th Sem Stuff\LP-5 Problem Statement & Programs\Programs\HPC Practical 1>**g++ -fopenmp bfs\_dfs.cpp -o bfs\_dfs**

C:\Users\Shravan\OneDrive\Desktop\Engineering Degree Stuff\4th Year Stuff\8th Sem Stuff\LP-5 Problem Statement & Programs\Programs\HPC Practical 1>**.\bfs\_dfs**

Enter number of nodes: 6

Enter number of edges: 7

Enter 7 edges (u v):

0 1

0 2

0 3

1 4

2 5

3 5

4 5

Enter start node for BFS/DFS: 0

Parallel BFS starting from node 0:

Thread 0 visiting 0

Thread 2 visiting 1

Thread 1 visiting 2

Thread 0 visiting 3

Thread 1 visiting 4

Thread 0 visiting 5

BFS Execution time: 0.0139999 seconds

Parallel DFS starting from node 0:

Thread 4 visiting 0

Thread 1 visiting 1

Thread 4 visiting 2

Thread 3 visiting 3

Thread 1 visiting 4

Thread 4 visiting 5

Thread 3 visiting 5

Thread 1 visiting 5

Thread 4 visiting 3

Thread 2 visiting 4

Thread 2 visiting 1

Thread 5 visiting 4

Thread 5 visiting 1

Thread 0 visiting 3

Thread 3 visiting 2

Thread 1 visiting 2

DFS Execution time: 0.0150001 seconds

The code begins with #include <iostream>, which is a preprocessor directive that includes the iostream library, providing input and output functionalities like printing to the console and reading user input. #include <vector> includes the vector library, which allows the use of dynamic arrays, a fundamental data structure for representing the adjacency list of the graph. #include <queue> includes the queue library, which provides the queue data structure, essential for the BFS algorithm to maintain the order of nodes to be visited. #include <omp.h> includes the OpenMP header file, which enables parallel programming constructs for utilizing multi-core processors to speed up the graph traversal algorithms. using namespace std; brings all identifiers (like cout, cin, vector, queue) from the standard namespace into the current scope, making them directly usable without the std:: prefix.

Next, a class named Graph is defined. This class encapsulates the data and methods related to a graph. Inside the class, public: declares that the following members are accessible from outside the class. int V; declares an integer variable V which will store the number of vertices (nodes) in the graph. vector<vector<int>> adj; declares a 2D vector named adj. This vector represents the adjacency list of the graph, where adj[i] will store a list of all the vertices that are directly connected to vertex i.

The Graph(int vertices) is the constructor of the Graph class. It takes an integer vertices as input, which represents the initial number of vertices in the graph. Inside the constructor, V = vertices; assigns the input vertices to the member variable V. adj.resize(V); resizes the adj vector to have V elements. Each element of adj is itself a vector of integers, ready to store the neighbors of each vertex.

The void addEdge(int u, int v) is a member function that adds an edge between two vertices u and v. // Undirected Graph is a comment indicating that this implementation is for an undirected graph. adj[u].push\_back(v); adds vertex v to the adjacency list of vertex u, indicating that v is a neighbor of u. adj[v].push\_back(u); adds vertex u to the adjacency list of vertex v, because the graph is undirected, so if v is a neighbor of u, then u is also a neighbor of v.

The void parallelBFS(int start) is a member function that performs a parallel Breadth-First Search starting from the given start vertex. vector<bool> visited(V, false); creates a vector of booleans named visited of size V, initialized to false. This vector keeps track of whether each vertex has been visited during the traversal. queue<int> q; creates a queue of integers named q, which will store the vertices to be visited in the BFS process. visited[start] = true; marks the starting vertex as visited. q.push(start); enqueues the starting vertex into the queue. cout << "Parallel BFS starting from node " << start << ":\n"; prints a message to the console indicating the start of the parallel BFS.

The while (!q.empty()) loop continues as long as the queue is not empty, meaning there are still vertices to explore. int levelSize = q.size(); gets the current size of the queue, which represents the number of nodes at the current level of the BFS tree. #pragma omp parallel for is an OpenMP directive that instructs the compiler to parallelize the following for loop across multiple threads. for (int i = 0; i < levelSize; ++i) iterates through all the nodes at the current level. Inside this parallel loop, int current; declares an integer variable current to store the vertex being processed by the current thread. #pragma omp critical is an OpenMP directive that ensures that the following block of code is executed by only one thread at a time, preventing race conditions when accessing shared resources like the queue and cout. Inside the critical section, if (!q.empty()) checks if the queue is not empty before attempting to dequeue. If it's not empty, current = q.front(); gets the front element of the queue (the next node to visit). q.pop(); removes the front element from the queue. cout << "Thread " << omp\_get\_thread\_num() << " visiting " << current << endl; prints a message indicating which thread is visiting which vertex.

After processing a vertex, another #pragma omp parallel for directive is used to parallelize the iteration through the neighbors of the current vertex. for (int j = 0; j < adj[current].size(); ++j) iterates through all the neighbors in the adjacency list of current. int neighbor = adj[current][j]; gets the current neighbor of current. Another #pragma omp critical section ensures that the following code for checking and marking visited neighbors and enqueuing them is done atomically. Inside this critical section, if (!visited[neighbor]) checks if the neighbor has not been visited yet. If it hasn't been visited, visited[neighbor] = true; marks the neighbor as visited. q.push(neighbor); enqueues the neighbor into the queue for further exploration in the next levels of BFS.

The void parallelDFSUtil(int v, vector<bool> &visited) is a recursive helper function for the parallel Depth-First Search. It takes the current vertex v and the visited vector as input. #pragma omp critical ensures that marking the current vertex as visited and printing the visiting message is done by only one thread at a time. Inside the critical section, visited[v] = true; marks the current vertex v as visited. cout << "Thread " << omp\_get\_thread\_num() << " visiting " << v << endl; prints a message indicating which thread is visiting the current vertex. The for (int u : adj[v]) loop iterates through all the neighbors u of the current vertex v. if (!visited[u]) checks if the neighbor u has not been visited yet. If it hasn't, #pragma omp task firstprivate(u) is an OpenMP directive that creates a new task to recursively call parallelDFSUtil on the unvisited neighbor u. firstprivate(u) ensures that each task gets its own private copy of the u variable. The task will be executed by one of the available threads. parallelDFSUtil(u, visited); recursively calls the DFS utility function on the neighbor u.

The void parallelDFS(int start) is the main function for performing parallel Depth-First Search starting from the given start vertex. vector<bool> visited(V, false); creates a visited vector initialized to false. cout << "Parallel DFS starting from node " << start << ":\n"; prints a starting message. #pragma omp parallel is an OpenMP directive that creates a team of threads to execute the following code in parallel. #pragma omp single is an OpenMP directive that ensures that the following block of code is executed by only one thread in the team. Inside this single region, parallelDFSUtil(start, visited); initiates the recursive parallel DFS starting from the start node.

The int main() is the entry point of the program. int V; declares an integer variable V for the number of nodes. cout << "Enter number of nodes: "; prompts the user to enter the number of nodes. cin >> V; reads the number of nodes from the user's input. Graph g(V); creates an object g of the Graph class with the specified number of vertices. int E; declares an integer variable E for the number of edges. cout << "Enter number of edges: "; prompts the user to enter the number of edges. cin >> E; reads the number of edges. cout << "Enter " << E << " edges (u v):\n"; prompts the user to enter the edges. The for (int i = 0; i < E; ++i) loop iterates E times to read each edge. int u, v; declares two integer variables u and v to store the vertices of an edge. cin >> u >> v; reads the two vertices of the edge from the user. g.addEdge(u, v); adds an edge between vertices u and v in the graph g. int startNode; declares an integer variable startNode for the starting node of BFS/DFS. cout << "Enter start node for BFS/DFS: "; prompts the user to enter the starting node. cin >> startNode; reads the starting node. double start\_time = omp\_get\_wtime(); records the starting time using OpenMP's wall-clock time function. g.parallelBFS(startNode); calls the parallelBFS function starting from the entered startNode. double end\_time = omp\_get\_wtime(); records the ending time. cout << "BFS Execution time: " << (end\_time - start\_time) << " seconds\n\n"; calculates and prints the execution time of the BFS. The same process is repeated for the parallel DFS: the start time is recorded, g.parallelDFS(startNode) is called, the end time is recorded, and the execution time is printed. Finally, return 0; indicates that the program executed successfully.