Title:
Design and Implement Parallel Breadth-First Search (BFS) and Depth-First Search (DFS) using OpenMP on a Tree or Undirected Graph.

Objective:
To improve the efficiency and reduce the execution time of traditional graph traversal algorithms by leveraging parallelism using OpenMP. The goal is to implement both BFS and DFS traversals in parallel on a graph or tree structure and compare them to their sequential counterparts.

Description:

Graph traversal is a fundamental operation in computer science and is widely used in various applications such as network routing, social network analysis, and artificial intelligence. Traditionally, graph traversal algorithms like BFS and DFS are implemented sequentially. However, with the advent of multi-core processors, it is possible to parallelize parts of these algorithms to achieve better performance.

This project aims to:

1. **Implement Sequential BFS and DFS** on an undirected graph or tree.
2. **Design and implement Parallel BFS and DFS** using OpenMP.
3. Analyze the **speedup** and **performance gain** of the parallel versions compared to the sequential versions.

Requirements:

- **Programming Language:** C/C++
- **Parallelism Library:** OpenMP
- **Input:** A tree or undirected graph represented using adjacency list or adjacency matrix.
- **Output:** The traversal order of nodes using both BFS and DFS, and timing comparisons between sequential and parallel versions.

OpenMP (Open Multi-Processing) is an **API (Application Programming Interface)** that supports **multi-platform shared-memory parallel programming in C, C++, and Fortran**. It allows developers to write programs that can run efficiently on multi-core systems by using **multiple threads** to execute parts of the code simultaneously.

🔧 **Key Features of OpenMP:**

1. **Easy to Use:**

- Uses **compiler directives (pragmas)** to parallelize code (e.g., #pragma omp parallel).
- Minimal changes needed in existing serial code.

2. **Shared Memory Model:**
   - All threads share the same memory space, which makes communication between them fast and straightforward.
3. **Scalable:**
   - Can run on systems with 2, 4, 8, or more cores/threads.
4. **Supports:**
   - **Fork-join parallelism:** A master thread spawns child threads to perform tasks and then joins back.
   - **Loop parallelism:** Distributes loop iterations across threads using #pragma omp for.

Problem Statement (In Simple Words)

You are required to:

- Implement **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** on a **graph** (can be a tree or undirected graph).
- **Parallelize** these algorithms using **OpenMP**, a popular library in C/C++ for parallel processing.

🔶 Objective

1. Understand BFS and DFS traversal on graphs.
2. Use **OpenMP** to make BFS and DFS **faster** by executing parts of them **in parallel**.
3. Input: a number of vertices and edges.
4. Output: the order in which nodes are visited in BFS and DFS (starting from node 0).
5. Compare how the traversal performs (in terms of speed) when done in parallel vs. sequentially (though this code does not measure time—this could be added).

🧠 Approach

- Graph is represented using an **adjacency list** (vector<vector<int>>).
- Both BFS and DFS are implemented as member functions of a Graph class.
- OpenMP is used to parallelize:

- BFS: parallel exploration of neighbors.
- DFS: parallel recursive calls to unvisited neighbors.

📜 Code Breakdown

📌 1. Header and Graph Definition

cpp

CopyEdit

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>
```

- omp.h: header for OpenMP.
- vector: for adjacency list.
- queue: used in BFS.

cpp

CopyEdit

```
struct Graph {
    int V;
    vector<vector<int>> adj;
```

- V: number of vertices.
- adj: adjacency list.

📌 2. Constructor and Edge Addition

cpp

CopyEdit

```
Graph(int V) {
    this->V = V;
    adj.resize(V);
}
void addEdge(int u, int v) {
    adj[u].push_back(v);
```

```
    adj[v].push_back(u);  // since graph is undirected
}
```

- Initializes graph with V vertices.
- Adds edges to both directions (undirected).

📌 3. Parallel BFS Traversal

```
cpp
CopyEdit
void BFS(int start) {
    vector<bool> visited(V, false);
    queue<int> q;
    visited[start] = true;
    q.push(start);
```

- visited: keeps track of visited nodes.
- queue: holds current layer nodes in BFS.

```
cpp
CopyEdit
while (!q.empty()) {
    int u = q.front();
    q.pop();
    cout << u << " ";

    #pragma omp parallel for
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (!visited[v]) {
            #pragma omp critical
            {
                visited[v] = true;
                q.push(v);
```

```
        }
      }
    }
}
```

✅ **Parallelization**:

- The loop through neighbors is **parallelized**.
- A **critical section** is used to safely update shared data (visited and queue), preventing **race conditions**.

📌 4. Parallel DFS Traversal

cpp

CopyEdit

```
void DFS(int start) {
    vector<bool> visited(V, false);
    #pragma omp parallel
    {
        #pragma omp single nowait
        {
            DFSUtil(start, visited);
        }
    }
    cout << endl;
}
```

✅ OpenMP:

- #pragma omp single nowait: Only one thread starts DFS, but no waiting is enforced, enabling further parallelism.

cpp

CopyEdit

```
void DFSUtil(int u, vector<bool>& visited) {
    visited[u] = true;
    cout << u << " ";
```

```
    # pragma omp parallel for
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (!visited[v]) {
            DFSUtil(v, visited);  // recursive call
        }
    }
}
```

⚠️ **Issue here**:

- Although loop is parallelized, visited is shared and not thread-safe.
- Also, recursive calls are not managed with OpenMP tasks. For true parallel DFS, you'd need #pragma omp task with synchronization.

📌 5. Main Function

```
cpp
CopyEdit
int main() {
    int V;
    cout << "Enter the number of vertices: ";
    cin >> V;

    Graph g(V);

    int edgeCount;
    cout << "Enter the number of edges: ";
    cin >> edgeCount;

    cout << "Enter the edges (in format 'source destination'): \n";
    for (int i = 0; i < edgeCount; i++) {
        int u, v;
```

```
        cin >> u >> v;
        g.addEdge(u, v);
    }

    cout << "BFS traversal starting from node 0: ";
    g.BFS(0);

    cout << "DFS traversal starting from node 0: ";
    g.DFS(0);

    return 0;
}
```

- Takes user input for graph.
- Calls BFS and DFS from node 0.

```
      0
     / \
    1   2
   /|   |\
  3 4   5 6
```

Graph as edge list (undirected):

0 - 1

0 - 2

1 - 3

1 - 4

2 - 5

2 - 6

3 - 4

5 – 6

Input to Provide:

Enter the number of vertices:

7

Enter the number of edges:

8

Enter the edges (in format 'source destination'):

0 1

0 2

1 3

1 4

2 5

2 6

3 4

5 6

What You Implemented

- Created a Graph class with adjacency list representation.
- Implemented:
    - Sequential-style BFS and DFS
    - Used **OpenMP** directives to make them **parallel**
        - #pragma omp parallel for to parallelize neighbor traversal
        - #pragma omp critical to manage shared data safely in BFS
        - #pragma omp single for DFS entry point
- Collected input dynamically (nodes and edges).
- BFS and DFS both start from **node 0**.

How to Run the Code During Viva

1. Compile with OpenMP support:

bash

g++ -fopenmp 01_bfs_dfs.cpp -o bfs_dfs

./bfs_dfs

2. Input:

7

8

0 1

0 2

1 3

1 4

2 5

2 6

3 4

5 6

What to Say About Parallelism

For BFS:

“In BFS, neighbor traversal is done in parallel using #pragma omp parallel for. Since threads share the visited array and queue, I used #pragma omp critical to avoid race conditions.”

For DFS:

“DFS is more recursive and less naturally parallel. I used OpenMP to start DFS in a single thread, and looped through neighbors in parallel. However, DFS parallelization is limited without tasks.”

You can also acknowledge that:

- DFS parallelism **can be improved** with #pragma omp task, and you're aware of it.
- You focused on basic parallel constructs for this version.

Common Viva Questions & What to Answer

Question	Suggested Answer
Why OpenMP?	It's easy to use, supports shared-memory parallelism, and is built into C/C++.
What is a race condition?	When multiple threads access shared data at the same time without proper synchronization.

Question	Suggested Answer
Why is critical used in BFS?	To ensure only one thread updates the queue and visited array at a time.
Why not use task in DFS?	Tasks would improve parallelism, but I chose simple constructs for clarity.