Explanation of Parallel DFS and BFS Practical

1 Overview of the Practical

This practical involves implementing and comparing **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** graph traversal algorithms in both **sequential** (single-threaded) and **parallel** (multi-threaded) versions using **OpenMP**, a parallel programming tool for C/C++. The objectives are:

- Write programs for sequential and parallel DFS and BFS.
- Measure performance on a user-defined graph (nodes, edges, start node).
- Compute metrics like speedup (sequential time / parallel time) and efficiency (speedup / number of threads).
- Verify that parallel versions produce the same traversal order as sequential versions.
- Understand OpenMP directives for parallelization.

2 DFS Code Explanation

The DFS code implements iterative **Sequential DFS** and **Parallel DFS** using a stack to traverse a graph.

2.1 Sequential DFS

Purpose: Traverses the graph by exploring as far as possible along each branch before backtracking.

How it Works:

- Uses a stack to track nodes.
- Starts at the given node, marks it visited, and pushes it onto the stack.
- Pops a node, explores its unvisited neighbors, marks them visited, and pushes

them onto the stack.

- Stores the visitation order in `visitedOrder`.
- Runs on a single thread.

Code:

```
void sequentialDFS(int start, int n) {
    for (int i = 1; i <= n; i++) visited[i] = false;
    visitedOrder.clear();
    stack<int> s;
    s.push(start);
    visited[start] = true;
    visitedOrder.push_back(start);
    while (!s.empty()) {
        int node = s.top();
        s.pop();
        for (int next_node : adj[node]) {
            if (!visited[next_node]) {
                visited[next_node] = true;
                visitedOrder.push_back(next_node);
                s.push(next_node);
            }
        }
    }
}
```

2.2 Parallel DFS

Purpose: Parallelizes the exploration of a node's neighbors using OpenMP.

How it Works:

- Similar to sequential DFS but uses `#pragma omp parallel for` to process neighbors in parallel.
- Uses `atomic<bool>` for the `visited` array to prevent race conditions.
- Uses `#pragma omp critical` to safely update `visitedOrder` and the stack.

Code:

```
void parallelDFS(int start, int n, int numThreads) {
    omp_set_num_threads(numThreads);
    for (int i = 1; i <= n; i++) visited[i] = false;
    visitedOrder.clear();
    stack<int> s;
    s.push(start);
    visited[start] = true;
    visitedOrder.push_back(start);
    while (!s.empty()) {
        int node = s.top();
        s.pop();
        #pragma omp parallel for
        for (int i = 0; i < adj[node].size(); i++) {
            int next_node = adj[node][i];
            if (!visited[next_node]) {
                visited[next_node] = true;
                #pragma omp critical
                {
                    visitedOrder.push_back(next_node);
                    s.push(next_node);
                }
            }
        }
    }
}
```

2.3 DFS Main Function

Purpose: Sets up the graph, runs both DFS versions, and measures performance.

How it Works:

- Takes input for nodes, edges, threads, and start node.
- Validates input and builds an undirected graph using an adjacency list.
- Runs Sequential DFS, measures time, and stores the visitation order.
- Runs Parallel DFS, measures time, and compares the order to verify correctness.
- Prints times, speedup, efficiency, correctness, and visited nodes.

3 BFS Code Explanation

The BFS code implements **Sequential BFS** and **Parallel BFS** using a queue to traverse a graph level by level.

3.1 Sequential BFS

Purpose: Traverses the graph by exploring all nodes at the current distance before moving to the next level.

How it Works:

- Uses a queue to track nodes.

- Starts at the given node, marks it visited, and enqueues it.
- Dequeues a node, explores its unvisited neighbors, marks them visited, and enqueues them.
- Stores the visitation order in `visitedOrder`.
- Runs on a single thread.

Code:

```
void sequentialBFS(int start, int n) {
    for (int i = 1; i <= n; i++) visited[i] = false;
    visitedOrder.clear();
    queue<int> q;
    q.push(start);
    visited[start] = true;
    visitedOrder.push_back(start);
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        for (int next_node : adj[node]) {
            if (!visited[next_node]) {
                visited[next_node] = true;
                visitedOrder.push_back(next_node);
                q.push(next_node);
            }
        }
    }
}
```

3.2 Parallel BFS

Purpose: Parallelizes the exploration of nodes at each level.

How it Works:

- Processes nodes level by level.
- Uses `#pragma omp parallel` and `#pragma omp for` to distribute nodes in the current level across threads.
- Each thread pops a node, explores its neighbors, and stores unvisited neighbors in a thread-local list.
- Merges thread-local lists into the global queue and `visitedOrder` after each level.

Code:

```
void parallelBFS(int start, int n, int numThreads) {
    omp_set_num_threads(numThreads);
    for (int i = 1; i <= n; i++) visited[i] = false;
    visitedOrder.clear();
    queue<int> q;
    q.push(start);
    visited[start] = true;
    visitedOrder.push_back(start);
    while (!q.empty()) {
        int currentLevelSize = q.size();
        vector<vector<int>> localNextLevel(numThreads);
        #pragma omp parallel
        {
            int tid = omp_get_thread_num();
            #pragma omp for schedule(static)
            for (int i = 0; i < currentLevelSize; i++) {
                int node = q.front();
                q.pop();
                for (int next_node : adj[node]) {
                    if (!visited[next_node]) {
                        visited[next_node] = true;
                        localNextLevel[tid].push_back(next_node);
                    }
                }
            }
        }
        for (int t = 0; t < numThreads; t++) {
            for (int next_node : localNextLevel[t]) {
                q.push(next_node);
                visitedOrder.push_back(next_node);
            }
        }
    }
}
```

3.3 BFS Main Function

Purpose: Sets up the graph, runs both BFS versions, and measures performance.

How it Works:

- Similar to DFS: takes input, validates, builds the graph, runs Sequential and Parallel BFS, and compares results.

4 Output Explanation

4.1 DFS Output Examples

Input: 10 nodes, 9 edges, 8 threads, start node 1

```
Sequential DFS Time: 0.0281 ms
Parallel DFS Time: 7.4499 ms
Speedup: 0.00377186
Threads Used: 8
Efficiency: 0.000471483
Correctness: Pass
Visited nodes: 1 2 3 4 5 6 7 8 9 10
```

Input: 15 nodes, 14 edges, 12 threads, start node 1

```
Sequential DFS Time: 0.0162 ms
Parallel DFS Time: 3.9135 ms
Speedup: 0.00413952
Threads Used: 12
Efficiency: 0.00034496
Correctness: Pass
Visited nodes: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Input: 10 nodes, 9 edges, 4 threads, start node 1

```
Sequential DFS Time: 0.0154 ms
Parallel DFS Time: 4.059 ms
Speedup: 0.00379404
Threads Used: 4
Efficiency: 0.000948509
Correctness: Pass
Visited nodes: 1 2 3 4 5 6 7 8 9 10
```

4.2 BFS Output Example

Input: 10 nodes, 9 edges, 4 threads, start node 1

```
Sequential BFS Time: 0.0165 ms
Parallel BFS Time: 7.7299 ms
Speedup: 0.00213457
Threads Used: 4
Efficiency: 0.000533642
Correctness: Pass
```

Key Points:

- Sequential DFS/BFS (0.0154–0.0281 ms for DFS, 0.0165 ms for BFS) is faster than Parallel DFS/BFS (3.9135–7.4499 ms for DFS, 7.7299 ms for BFS).

- Speedup < 1 (e.g., 0.00377186 for DFS) shows parallel is slower due to thread overhead.

- Low efficiency (e.g., 0.000471483 for DFS) indicates poor thread utilization for small graphs.

- Correctness “Pass” confirms identical traversal orders.
- Visited nodes reflect the graph’s linear structure.

5 How to Explain to the Examiner

To present this practical confidently:

1. **Introduce the Practical**:
    - “This practical implements DFS and BFS in sequential and parallel versions using OpenMP. We test them on a graph, measure times, and compute speedup and efficiency to compare performance.”
2. **Explain DFS Code**:
    - **Sequential DFS**: “Uses a stack to explore deeply, visiting one branch fully before backtracking, on one thread.”
    - **Parallel DFS**: “Parallelizes neighbor exploration with OpenMP. Uses atomic operations and critical sections to avoid conflicts.”
    - **Main Function**: “Inputs the graph, runs both DFS versions, measures times, verifies order, and prints metrics.”
3. **Explain BFS Code**:
    - **Sequential BFS**: “Uses a queue to explore level by level, visiting nodes at the same distance first, on one thread.”
    - **Parallel BFS**: “Parallelizes each level’s nodes. Threads store neighbors in local lists, merged into the global queue.”
    - **Main Function**: “Similar to DFS, it sets up the graph, runs BFS versions, and compares results.”
4. **Explain Output**:
    - “For DFS with 10 nodes and 8 threads, Sequential DFS took 0.0281 ms, Parallel DFS took 7.4499 ms. Speedup was 0.00377186, efficiency was 0.000471483. For BFS with 10 nodes and 4 threads, Sequential BFS took 0.0165 ms, Parallel BFS took 7.7299 ms.”
    - “Parallel was slower due to thread overhead for small graphs (10–15 nodes). Correctness passed, showing the same node order.”
5. **Explain Why Sequential Was Faster**:
    - “Small graphs meant little work per thread, and OpenMP’s thread creation and synchronization overhead was significant. Larger graphs

might favor parallel versions.”

6 Tips for Explanation

- Simplify: DFS explores deeply; BFS explores level by level.
- Highlight OpenMP: `#pragma omp parallel for` splits work, `#pragma omp critical` protects shared data.
- Explain overhead: Small graphs make parallel slower due to thread management.
- Be ready for questions: Why parallel is slower, how correctness is checked, or how DFS/BFS are parallelized.