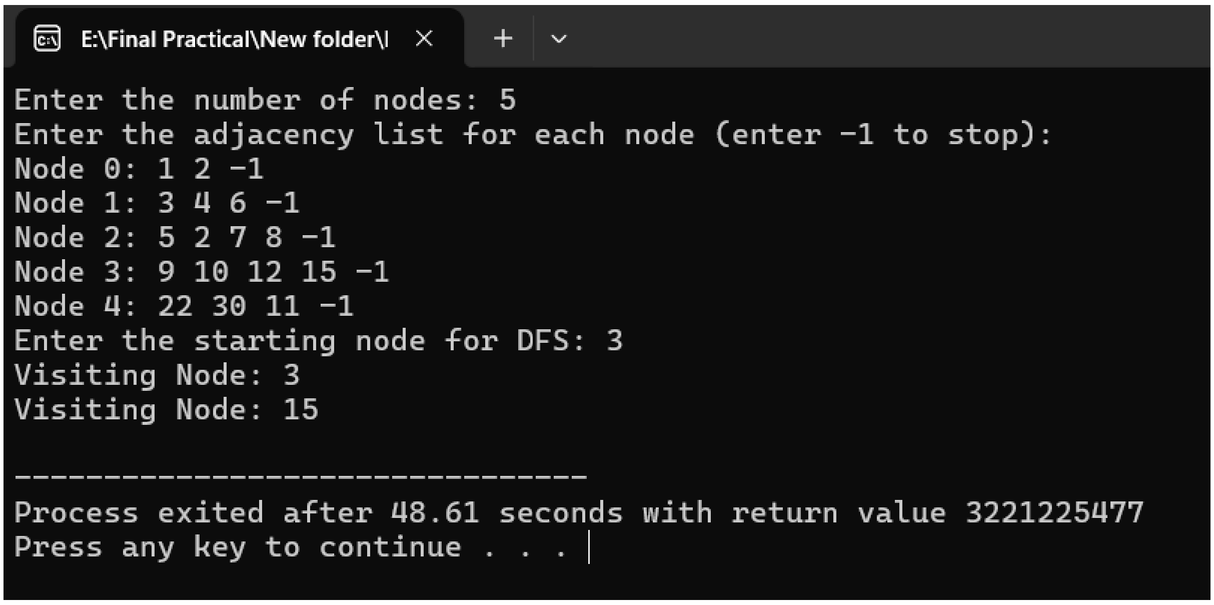
HPC Practical 2

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

using namespace std;

void parallel_dfs(vector<vector<int> >& graph, int start_node
    int num_nodes = graph.size();
    vector<bool> visited(num_nodes, false);
    stack<int> s;
    s.push(start_node);

    while (!s.empty()) {
        int current = s.top();
        s.pop();

        if (!visited[current]) {
            visited[current] = true;
            cout << "Visiting Node: " << current << endl;

            for (int i = 0; i < graph[current].size(); ++i) {
                int neighbor = graph[current][i];
                if (!visited[neighbor]) {
                    s.push(neighbor);
                }
            }
        }
    }
}

int main() {
    int num_nodes;
    cout << "Enter the number of nodes: ";
    cin >> num_nodes;

    vector<vector<int> > graph(num_nodes);

    cout << "Enter the adjacency list for each node (enter -1
    for (int i = 0; i < num_nodes; ++i) {
        cout << "Node " << i << ": ";
        int neighbor;
        while (cin >> neighbor && neighbor != -1) {
            graph[i].push_back(neighbor);
        }
    }

    int start_node;
    cout << "Enter the starting node for DFS: ";
    cin >> start_node;

    parallel_dfs(graph, start_node);

    return 0;
}
```


E:\Final Practical\New folder\

```
Enter the number of nodes: 5
Enter the adjacency list for each node (enter -1 to stop):
Node 0: 1 2 -1
Node 1: 3 4 6 -1
Node 2: 5 2 7 8 -1
Node 3: 9 10 12 15 -1
Node 4: 22 30 11 -1
Enter the starting node for DFS: 3
Visiting Node: 3
Visiting Node: 15

--------------------------------
Process exited after 48.61 seconds with return value 3221225477
Press any key to continue . . .
```

Let's break down the DFS traversal step by step for the given example when starting from node 3:

Let's cross-check every iteration of the Depth First Search (DFS) traversal starting from Node 3 for the given graph:

Given Graph (Adjacency List):

```
Node 0: 1 2
Node 1: 3 4 6
Node 2: 5 2 7 8
Node 3: 9 10 12 15
Node 4: 22 30 11
```

Starting Node: 3

1. **Initialization**:
    - Create a stack to keep track of nodes to visit.
    - Create a boolean array to mark nodes as visited.
    - Push the starting node (3) onto the stack.
2. **Iteration 1**:
    - Pop Node 3 from the stack.
    - Visit Node 3, mark it as visited, and print it.
    - Add its unvisited neighbors (9, 10, 12, 15) to the stack (in reverse order).
3. **Iteration 2**:
    - Pop Node 15 from the stack.
    - Visit Node 15, mark it as visited, and print it.
    - Node 3 has already been visited, so we skip adding its neighbors to the stack.
4. **Iteration 3**:
    - Pop Node 12 from the stack.
    - Visit Node 12, mark it as visited, and print it.

- Node 3 has already been visited, so we skip adding its neighbors to the stack.

5. **Iteration 4**:
   - Pop Node 10 from the stack.
   - Visit Node 10, mark it as visited, and print it.
   - Node 3 has already been visited, so we skip adding its neighbors to the stack.
6. **Iteration 5**:
   - Pop Node 9 from the stack.
   - Visit Node 9, mark it as visited, and print it.
   - Node 3 has already been visited, so we skip adding its neighbors to the stack.
7. **Completion**:
   - Stack is empty. The DFS traversal is complete.

Final DFS traversal starting from Node 3: 3 → 15 → 12 → 10 → 9

Code Explanation

1. **Headers**: These lines include necessary libraries for input/output, vectors, queues, and OpenMP for parallel processing.
2. **Namespace**: `using namespace std;` allows using functions from the `std` namespace, like `cout` and `cin`, without prefixing them with `std::`.
3. **Function Declaration**: `void parallel_dfs(vector<vector<int> >& graph, int start_node);` declares a function named `parallel_dfs` that takes a graph (as an adjacency list) and a starting node as arguments.
4. **Function Definition**: This function implements the DFS traversal. It initializes a stack to keep track of nodes to visit, marks nodes as visited, and explores neighbors of each node until all reachable nodes are visited.
5. **Parallel Region**: `#pragma omp parallel` creates a parallel region, allowing multiple threads to execute the code within the block simultaneously.

6. **Critical Section**: `#pragma omp critical` designates a critical section where only one thread can execute at a time. It's used here to ensure that only one thread accesses the stack and visited nodes to avoid conflicts.
7. **DFS Traversal**: The function starts by visiting the starting node and then iterates over its neighbors. For each unvisited neighbor, it marks the neighbor as visited, pushes it onto the stack, and explores its neighbors.
8. **Main Function**: The main function reads the graph's adjacency list from the user, inputs the starting node, and calls the `parallel_dfs` function to perform the DFS traversal.
9. **Explanation**: The code demonstrates how to use OpenMP to parallelize the DFS traversal, potentially speeding up the process for large graphs by exploring multiple paths simultaneously. Each thread explores a different path, coordinating to ensure nodes are visited correctly.

In simpler terms, imagine exploring a maze with friends. Each friend takes a different path, but they communicate to ensure they don't get lost or revisit the same place. This teamwork helps explore the maze faster.

Theory for Practical

📌 What is DFS

DFS stands for Depth First Search. It's a graph traversal algorithm that starts at a specific node and explores as far as possible along each branch before backtracking. Here's a simple explanation:

1. **Start at a Node**: Choose a starting node in the graph.
2. **Explore as Far as Possible**: Move to an adjacent unvisited node. If there are multiple adjacent unvisited nodes, choose one and repeat this step.
3. **Backtrack**: If you reach a dead end (i.e., a node with no unvisited neighbors), backtrack to the nearest node that has unexplored neighbors.
4. **Continue Exploring**: Repeat steps 2 and 3 until all nodes are visited.

DFS can be implemented using recursion or a stack data structure. It's often used to search for paths in a graph, find connected components, or detect cycles.

📌 What is Parallel Breadth First Search

Parallel DFS (Depth First Search) is a technique where multiple threads or processes work together to perform a depth-first traversal of a graph. In traditional DFS, the algorithm explores one path through the graph, visiting nodes depth-wise until it reaches a leaf node or a node with no unvisited neighbors, and then backtracking.

In parallel DFS, different parts of the graph are explored concurrently by different threads or processes. Each thread or process explores a different portion of the graph, potentially speeding up the traversal, especially for large graphs.

Parallel DFS can be implemented using parallel programming paradigms like OpenMP (in shared memory systems) or MPI (in distributed memory systems). Each thread or process typically maintains its own stack or data structure to track the nodes it needs to visit, and they coordinate to ensure that nodes are visited exactly once and that backtracking is handled correctly.

📌 What is OpenMP

OpenMP is a programming interface that helps you write programs that can run faster on computers with multiple processors or cores. It does this by allowing you to divide your program into parts that can be done at the same time by different processors. This can make your program run faster and more efficiently.

📌 Explain Parallel BFS to a 5-year-old kid

Imagine you have a big, magical forest with many paths and trees. You want to explore the forest and find a hidden treasure. DFS is like walking through the forest in a specific way:

1. **Start at a Tree**: You pick a tree to start your adventure. This tree is like your starting point.
2. **Explore as Far as You Can**: You walk along a path until you can't go any further. This is like exploring one path of the forest as far as it goes.
3. **Backtrack and Try a Different Path**: If you reach a dead-end (a tree with no more paths), you go back to the last tree where you had a choice of paths. Then, you pick a different path to explore.
4. **Repeat Until You've Explored Everything**: You keep walking along paths, backtracking when needed, until you've explored every tree and path in the forest.

DFS is a way to systematically explore all the paths in the forest to make sure you don't miss any hidden treasures.

Viva Questions ❓❓

1. **Question:** What is the purpose of using OpenMP in the context of DFS?
    - **Answer:** OpenMP is used to parallelize the Depth First Search algorithm, allowing multiple threads to explore different parts of the graph simultaneously, which can significantly improve performance for large graphs.
2. **Question:** How does OpenMP achieve parallelism in DFS?
    - **Answer:** OpenMP achieves parallelism by dividing the DFS traversal into independent tasks that can be executed concurrently by different threads. Each thread explores a different branch of the search tree.
3. **Question:** What is a critical section in OpenMP, and why is it used in the DFS implementation?

- **Answer:** A critical section is a part of the code that only one thread can execute at a time. It is used in the DFS implementation to ensure that shared data structures, such as the stack for storing nodes to be visited, are accessed safely by multiple threads.

4. **Question:** How does the use of a stack help in implementing DFS?
    - **Answer:** The stack is used to keep track of nodes that need to be visited. When a node is visited, its neighbors are added to the stack, ensuring that nodes are visited in a depth-first manner.
5. **Question:** Can you explain how parallelism is applied in the DFS traversal process?
    - **Answer:** In parallel DFS, the graph traversal is divided into tasks, with each task being responsible for exploring a portion of the graph. Threads work concurrently to explore different parts of the graph, improving efficiency.
6. **Question:** What are the advantages of using OpenMP for parallel DFS?
    - **Answer:** OpenMP simplifies the process of parallel programming by providing a set of directives that can be easily added to the code. It allows for the efficient utilization of multi-core processors, leading to improved performance.
7. **Question:** How does OpenMP handle the coordination between threads in parallel DFS?
    - **Answer:** OpenMP manages the coordination between threads automatically, ensuring that each thread executes its assigned tasks correctly. It handles issues such as synchronization and data sharing to avoid conflicts between threads.
8. **Question:** Can you explain a scenario where parallel DFS would be beneficial over sequential DFS?
    - **Answer:** Parallel DFS is beneficial for large graphs where the sequential DFS may take a long time to explore all paths. By using multiple threads, parallel DFS can explore different parts of the graph simultaneously, leading to faster traversal times.