```
// 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

#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <omp.h>
using namespace std;

class Graph {
    public:
    int nodes;
    vector<vector<int>> adjList;

    Graph(int n) {
        nodes = n;
        adjList.resize(n);
    }

    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    void parallelBFS(int start) {
        vector<bool> visited(nodes, false);
        queue<int> q;

        visited[start] = true;
        q.push(start);

        cout << "Parallel BFS: ";

        while (!q.empty()) {
            int size = q.size();

            #pragma omp parallel for
            for (int i = 0; i < size; i++) {
                int current;

                #pragma omp critical
                {
                    if (!q.empty()) {
                        current = q.front();
                        q.pop();
                        cout << current << " ";
                    }
                }

                for (int neighbor : adjList[current]) {
                    if (!visited[neighbor]) {
                        #pragma omp critical
                        {
                            if (!visited[neighbor]) {
                                visited[neighbor] = true;
                                q.push(neighbor);
                            }
                        }
                    }
                }
            }
        }

        cout << endl;
    }

    void parallelDFS(int start) {
        vector<bool> visited(nodes, false);
        stack<int> s;
        s.push(start);

        cout << "Parallel DFS: ";

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

            if (!visited[current]) {
                visited[current] = true;
                cout << current << " ";

                #pragma omp parallel for
                for (int i = 0; i < adjList[current].size(); i++) {
                    int neighbor = adjList[current][i];

                    if (!visited[neighbor]) {
                        #pragma omp critical
                        s.push(neighbor);
                    }
                }
            }
        }

        cout << endl;
    }
};

int main() {
    Graph g(8);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);
    g.addEdge(3, 7);

    g.parallelBFS(0);
    g.parallelDFS(0);

    return 0;
}
```