**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.**

**Name: Kaustubh Shrikant Kabra**

**Roll No: 37**

**BE COMPUTER SHIFT 1**

**Parallel Breadth First Search using Tree.**

**Code:**

#include <iostream>

#include <queue>

#include <vector>

#include <omp.h>

using namespace std;

struct TreeNode {

    int val;         // Value of the node

    TreeNode\* left;  // Pointer to the left child

    TreeNode\* right; // Pointer to the right child

    TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}

};

bool bfs(TreeNode\* root, int target\_val, int\* visited\_nodes) {

    queue<TreeNode\*> q;

    q.push(root);

    int num\_visited = 0;

    visited\_nodes[num\_visited++] = root->val;

    while (!q.empty()) {

        bool found = false;

        #pragma omp parallel for // Begin parallel section

        for (int i = 0; i < q.size(); i++) {

            TreeNode\* node = q.front();

            q.pop();

            if (node->val == target\_val) {

                // Node found!

                found = true;

            }

            if (node->left) {

                #pragma omp critical

                {

                    q.push(node->left);

                    visited\_nodes[num\_visited++] = node->left->val;

                }

            }

            if (node->right) {

                #pragma omp critical

                {

                    q.push(node->right);

                    visited\_nodes[num\_visited++] = node->right->val;

                }

            }

        }

        if (found) { // Node found!

            return true;

        }

    }

    // Node not found

    return false;

}

int main() {

    // Example binary tree

    TreeNode\* root = new TreeNode(1);

    root->left = new TreeNode(2);

    root->right = new TreeNode(3);

    root->left->left = new TreeNode(4);

    root->left->right = new TreeNode(5);

    root->right->left = new TreeNode(6);

    root->right->right = new TreeNode(7);

    int target\_val = 5;

    int visited\_nodes[7]; // Array to store visited nodes

    for (int i = 0; i < 7; i++) {

        visited\_nodes[i] = -1; // Initialize array with -1

    }

    bool found = bfs(root, target\_val, visited\_nodes);

    if (found) {

        cout << "Node with value " << target\_val << " found in the tree!" << endl;

    } else {

        cout << "Node with value " << target\_val << " not found in the tree." << endl;

    }

    cout << "Nodes visited in order: ";

    for (int i = 0; i < 7; i++) {

        if (visited\_nodes[i] != -1) {

            cout << visited\_nodes[i] << " ";

        }

    }

    cout << endl;

    return 0;

}

**Output:**

**Case 1:**

PS C:\Practical\_1> g++ -fopenmp BFS\_TREE.cpp -o tree

PS C:\Practical\_1> .\tree.exe

Node with value 5 found in the tree!

Nodes visited in order: 1 2 3 4 5 6 7

**Case 2:**

PS C:\Practical\_1> .\tree.exe

Node with value 10 not found in the tree.

Nodes visited in order: 1 2 3 4 5 6 7

**Parallel Depth First Search using Tree.**

**Code:**

#include <iostream>

#include <stack>

#include <vector>

#include <omp.h>

using namespace std;

struct TreeNode {

    int val;         // Value of the node

    TreeNode\* left;  // Pointer to the left child

    TreeNode\* right; // Pointer to the right child

    TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}

};

bool dfs(TreeNode\* root, int target\_val) {

    stack<TreeNode\*> s;

    s.push(root);

    bool found = false;

    while (!s.empty()) {

        vector<int> visited\_nodes; // to store the order of visited nodes in each iteration

        #pragma omp parallel shared(found) // Begin parallel section

        {

            vector<TreeNode\*> local\_stack; // to store the nodes in the local stack

            #pragma omp single // only one thread will execute this block

            {

                local\_stack.push\_back(s.top());

                s.pop();

            }

            while (!local\_stack.empty()) {

                TreeNode\* node = local\_stack.back();

                local\_stack.pop\_back();

                visited\_nodes.push\_back(node->val); // add the visited node to visited\_nodes vector

                if (node->val == target\_val) {

                    #pragma omp critical

                    {

                        found = true; // Node found!

                    }

                }

                if (node->right) {

                    local\_stack.push\_back(node->right);

                }

                if (node->left) {

                    local\_stack.push\_back(node->left);

                }

            }

        }

        // Print the order of visited nodes in this iteration

        cout << "Visited nodes: ";

        for (int i = 0; i < visited\_nodes.size(); i++) {

            cout << visited\_nodes[i] << " ";

        }

        cout << endl;

        if (found) {

            // Node found!

            return true;

        }

    }

    // Node not found

    return false;

}

int main() {

    // Example binary tree

    TreeNode\* root = new TreeNode(1);

    root->left = new TreeNode(2);

    root->right = new TreeNode(3);

    root->left->left = new TreeNode(4);

    root->left->right = new TreeNode(5);

    root->right->left = new TreeNode(6);

    root->right->right = new TreeNode(7);

    int target\_val = 5;

    bool found = dfs(root, target\_val);

    if (found) {

        cout << "Node with value " << target\_val << " found in the tree!" << endl;

    } else {

        cout << "Node with value " << target\_val << " not found in the tree." << endl;

    }

    return 0;

}

**Output:**

**Case 1:**

PS C:\ Practical\_1> g++ -fopenmp DFS\_TREE.cpp -o dfstree

PS C:\Practical\_1> .\dfstree.exe

Visited nodes order: 1 2 4 5 3 6 7

Node with value 5 found in the tree!

**Case 2:**

PS C:\Practical\_1> g++ -fopenmp .\DFS\_TREE.cpp -o dfstree

PS C:\Practical\_1> .\dfstree.exe

Visited nodes: 1 2 4 5 3 6 7

Node with value 8 not found in the tree.