Implementing Parallelism in DFS for Directed Acyclic Graphs

Languages Used:

• C++

Algorithm:

- **Input Parameters**:

Prompt the user to input the number of nodes (**n**) and the number of edges (**edges**) in the graph.
Take input for each edge, consisting of a parent and a child node.
Input the source (**src**) and destination (**dst**) nodes.

- **Check Source and Destination**:

If the source and destination nodes are the same, output the source node and exit.

- **Initialize Variables**:

Determine the number of threads (**num_threads**) based on the number of child nodes of the source node.
If there are no child nodes (i.e., **num_threads** is 0), output "No Solution" and exit.
Create a vector **branch_paths** to store the initial paths for each thread, where each path starts from the source node.

- **Parallel DFS**:

Start parallel execution using OpenMP.
Each thread runs a DFS traversal starting from one of the child nodes of the source node.
Each thread maintains its own stack (**dfs_parallel_stack**) to perform DFS traversal.
Inside the loop, each thread pops the top node from its stack and checks if it is the destination node. If yes, update the solution and set **found** flag to true.
If the destination node is not found, the thread continues the DFS traversal until the stack becomes empty or the destination node is found.
Use **#pragma omp critical** to ensure mutual exclusion while updating the solution variable.
Parallel execution ends when either the destination is found or all threads have finished.

- **Output Parallel DFS Results**:

Output the path found by parallel DFS, along with the number of threads used and the time taken for computation.

- **Normal DFS**:

Perform DFS traversal using a single thread, maintaining a stack (**dfs_normal_stack**) similar to parallel DFS.

Continue traversal until the destination node is found or the stack becomes empty.

- **Output Normal DFS Results**:

Output the path found by normal DFS and the time taken for computation.

Code:

```cpp
#include<bits/stdc++.h>
#include<omp.h>
using namespace std;

int main(){

    //Input Parameters
    int n;
    cout << "Enter number of nodes: ";
    cin >> n;
    vector<vector<int>> G(n + 1);
    int edges;
    cout << "Enter number of edges: ";
    cin >> edges;
    cout << "---Input for Acyclic Directed Graph---\n";
    while(edges > 0){
        int parent, child;
        cout << "Enter parent and child: ";
        cin >> parent >> child;
        G[parent].push_back(child);
        edges -= 1;
    }
    int src, dst;
    cout << "Enter source and destination: ";
    cin >> src >> dst;
    if(src == dst){
        cout << src << "\n";
        return 0;
    }

    //Defining threads for the branchinf factor from source node
    int num_threads = G[src].size();
    if(num_threads == 0){
        cout << "No Solution";
        return 0;
    }
    vector<pair<int,string>> branch_paths;
    for(auto x : G[src]) branch_paths.push_back({x, to_string(src) + "->" + to_string(x)});

    //Starting Parallel DFS
    string solution = "";
    bool found = false;
    double start_time = omp_get_wtime();
    omp_set_num_threads(num_threads);
    #pragma omp parallel
    {
        //Letting each branch take a stack
        int thread_id = omp_get_thread_num();
        stack<pair<int,pair<int,string>>> dfs_parallel_stack;
        dfs_parallel_stack.push({0,{branch_paths[thread_id].first,branch_paths[thread_id].second}});

        //Running till stack is empty
        while(!dfs_parallel_stack.empty()){
```

```
            if(found) break;
            pair<int,pair<int,string>> cur = dfs_parallel_stack.top();
            dfs_parallel_stack.pop();
            int node = cur.second.first, next_node_idx = cur.first;
            string current_path = cur.second.second;

            //Critical Section to access solution variable
            #pragma omp critical
            {
                if(node == dst){
                    solution = current_path;
                    found = true;
                }
            }
            if(found) break;


            //next node condition
            if(next_node_idx < G[node].size()){
                dfs_parallel_stack.push({next_node_idx + 1,{node, current_path}});
                int next_node = G[node][next_node_idx];
                dfs_parallel_stack.push({0,{next_node,current_path + "->" + to_string(next_node)}});
            }
        }
    }
    double end_time = omp_get_wtime();
    cout << "-----Parallel DFS-----\n";
    cout << "Number of Threads: " << num_threads << "\n";
    if(solution.size() == 0) cout << "No solution available\n";
    else cout << "Path is: " << solution << "\n";
    cout << "Computed in " << end_time - start_time << " units of time\n";

    //Starting Normal DFS
    solution = "";
    start_time = omp_get_wtime();
    stack<pair<int,pair<int,string>>> dfs_normal_stack;
    dfs_normal_stack.push({0,{src, to_string(src)}});

    //Running till stack is empty
    while(!dfs_normal_stack.empty()){
        pair<int,pair<int,string>> cur = dfs_normal_stack.top();
        dfs_normal_stack.pop();
        int node = cur.second.first, next_node_idx = cur.first;
        string current_path = cur.second.second;

        //Breaking the while loop if destination found
        if(solution.size() != 0) break;
        if(node == dst) {
            solution = current_path;
            break;
        }
```

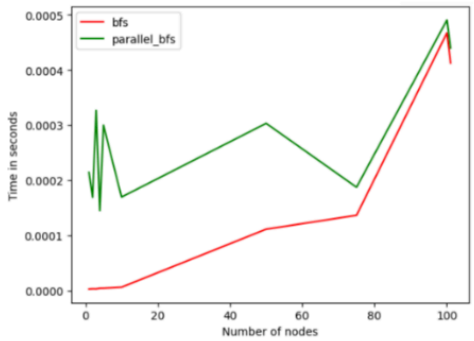
```
        }

        //next node condition
        if(next_node_idx < G[node].size()){
            dfs_normal_stack.push({next_node_idx + 1,{node, current_path}});
            int next_node = G[node][next_node_idx];
            dfs_normal_stack.push({0,{next_node,current_path + "->" + to_string(next_node)}});
        }
    }
    end_time = omp_get_wtime();
    cout << "-----Normal DFS-----\n";
    if(solution.size() == 0) cout << "No solution available\n";
    else cout << "Path is: " << solution << "\n";
    cout << "Computed in " << end_time - start_time << " units of time\n";
    return 0;
}
```

Results:

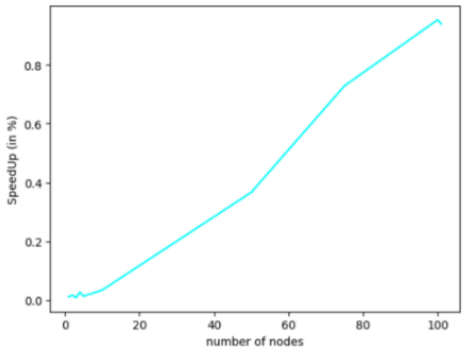
```
(base) cid@mugez:~/Downloads/Parallel_DFS$ ./run.exe
Enter number of nodes: 10
Enter number of edges: 9
---Input for Acyclic Directed Graph---
Enter parent and child: 1
2
Enter parent and child: 1
3
Enter parent and child: 1
4
Enter parent and child: 2
5
Enter parent and child: 2
6
Enter parent and child: 3
7
Enter parent and child: 3
8
Enter parent and child: 4
9
Enter parent and child: 4
10
Enter source and destination: 1
10
-----Parallel DFS-----
Number of Threads: 3
Path is: 1->4->10
Computed in 0.000748479 units of time
-----Normal DFS-----
Path is: 1->4->10
Computed in 2.6561e-05 units of time
```

```
(base) cid@mugez:~/Downloads/Parallel_DFS$ ./run.exe
Enter number of nodes: 7
Enter number of edges: 6
---Input for Acyclic Directed Graph---
Enter parent and child: 1
2
Enter parent and child: 1
3
Enter parent and child: 2
4
Enter parent and child: 2
5
Enter parent and child: 3
6
Enter parent and child: 3
7
Enter source and destination: 1
7
-----Parallel DFS-----
Number of Threads: 2
Path is: 1->3->7
Computed in 0.000454542 units of time
-----Normal DFS-----
Path is: 1->3->7
Computed in 2.4049e-05 units of time
```

For graphs with more of nodes the Parallel DFS performs better than Normal DFS.

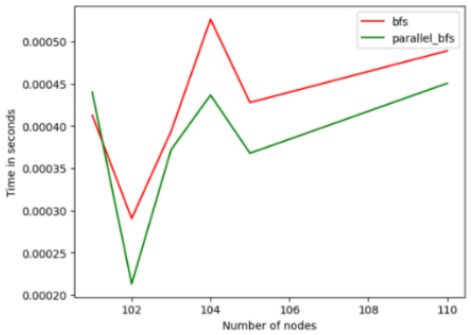


(a) Performance in seconds(upto 99 nodes)

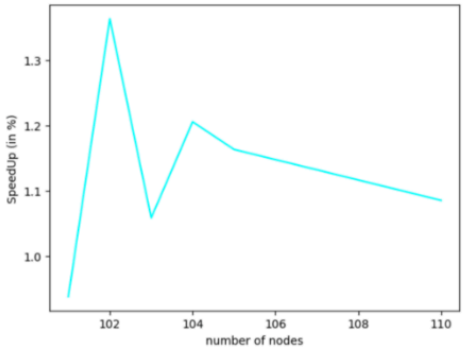


(b) Speed Up %(upto 99 nodes)

- When there are more than 100 nodes we can see parallelized bfs starts outperforming sequential bfs:

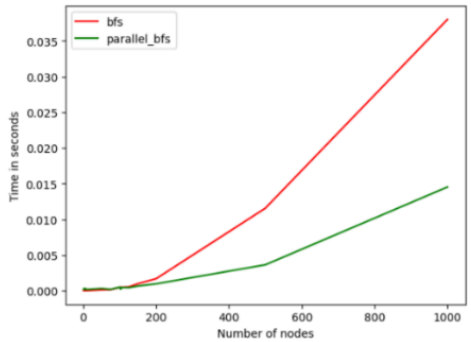


(c) Performance in seconds (100-110 nodes)

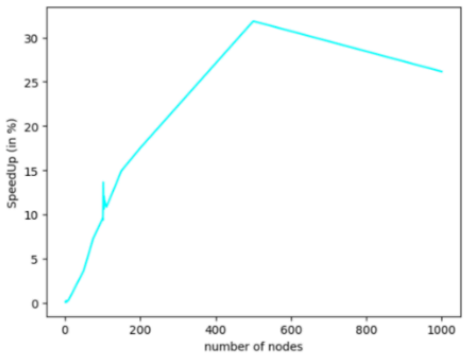


(d) Speed Up %

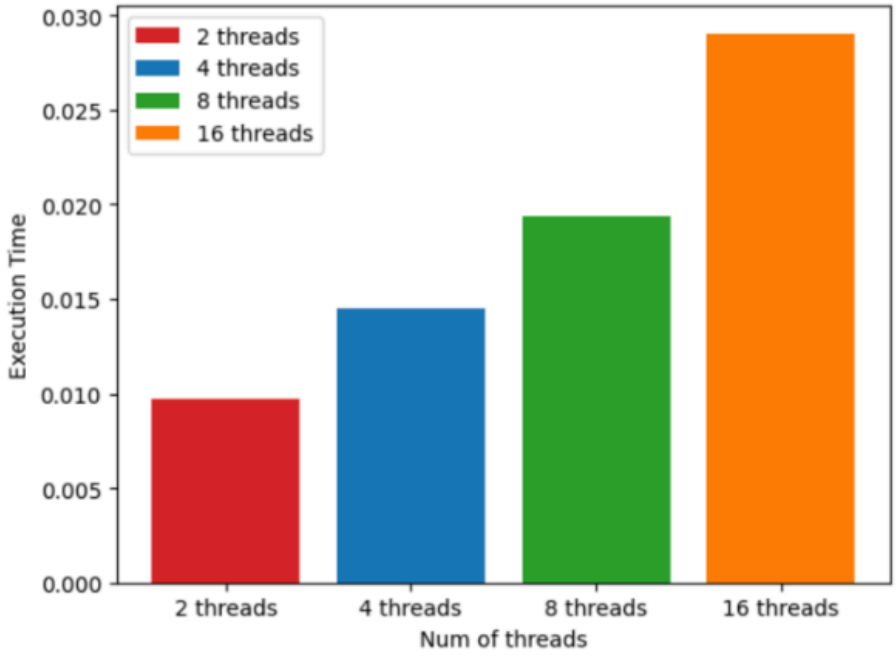
- The code is then run for a graph with 200,500 and 1000 nodes and here are the results:



(e) Performance in seconds (200,500 and 1000 nodes)



(f) Speed Up % (200,500 and 1000 nodes)



(g) Comparison based on threads (for 1000 nodes)