Name of the Student: _____ **Roll No:** ___

CLASS: - B. E. [COMP] **Division:** **Course: LP-V**

High Performance Computing

Assignment No. 01

PARALLEL BFS AND DFS

Marks:

Date of Performance: **Sign with Date:**

Title: Implement parallel Breadth First Search and Depth First Search.

Objectives:

- To study and implementation of directive based parallel programming model.

Outcomes:

- Students will understand implementation of parallel Breadth First Search and Depth First Search.

Pre-requisites:

64-bit Open source Linux or its derivative
Programming Languages: C/C++

PEOs, POs, PSOs and COs satisfied

PEOs: I, III **POs: 1, 2, 3, 5** **PSOs: 1** **Cos: 1,2**

Problem Statement:

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 .

Theory:

Introduction:

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

The applications of breadth-first-algorithm are given as follows -

- BFS can be used to find the neighboring locations from a given source location.
- In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
- BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- BFS is used to determine the shortest path and minimum spanning tree.
- BFS is also used in Cheney's technique to duplicate the garbage collection.
- It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Sequential BFS Algorithm

• Set all the vertices to not visited.
• Create a queue and add the start node or nodes.
• While the queue becomes not empty -
    - Take the first node from queue and remove it
    - If not visited already
    - Make the node visited
    - Add all the neighbors of the node into the queue.
• Time Complexity will be O(N 2)
• Space Complexity will be O(N2)
• N is total number of vertices and my implementation is based on adjacency matrix

Parallel BFS Algorithm

• Similar algorithm as the sequential BFS.
• Instead of popping out one vertex at a time, pop out all the nodes in the same level. (These nodes are known as frontier nodes)
• Level synchronous traversal. Each the processor will take a set of frontier vertices and calculate their next frontier vertices in parallel.
• For the above step we will need to partition the adjacency matrix and the vertices and allocate them to the processors.

```
Procedure    Parallel-Breadth-First-Search-Vertex(ALM, EM, U)
begin
    mark every vertex "unvisited"
    v ⇐ start vertex
    mark v "visited"
    instruct processor(i) where   1 ≤ i ≤ k
        for j = 1 to k do
            if (k * (j − 1) + i)≤EM(v)
            then delete v from U(ALM(v, k * (j − 1) + i))
            endif
        endfor
    end-instruction
    initialize queue with v
    while queue is not empty do
        begin
        v ⇐ first vertex from the queue
        for each w∈ U(v) do
            begin
            mark w "visited"
            instruct processor (i) where   1 ≤ i ≤ k
                for   j = 1 to k do
                        if (k * (j − 1) + i) ≤ EM(w)

                        then delete w from U(ALM(w, k * (j − 1) + i))
                        endif
                    endfor
              end-instruction
              add w to queue
              end
          endfor
      endwhile
end
```

Sequential DFS algorithm

The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows –

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Parallel Depth-First Search: Dynamic Load Balancing

- When a processor runs out of work, it gets more work from another processor.
- This is done using work requests and responses in message passing machines and locking and extracting work in shared address space machines.
- On reaching final state at a processor, all processors terminate.
- Unexplored states can be conveniently stored as local stacks at processors.
- The entire space is assigned to one processor to begin with.

Conclusion:

We have implemented parallel BFS and DFS operations.

Write short answer of following questions:

Q1.What is BFS and DFS explain with example.
Q2.Write difference between BFS and DFS.
Q3.Explain Parallel BFS.
Q4.Explain Parallel DFS

Code:-

```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
#include<queue>

using namespace std;
class node
{
  public:
        node *left, *right;
        int data;

};

class Breadthfs
{

 public:

 node *insert(node *, int);
 void bfs(node *);

};

node *insert(node *root, int data)
// inserts a node in tree
{

        if(!root)
        {

                root=new node;
                root->left=NULL;
                root->right=NULL;
                root->data=data;
                return root;
        }

        queue<node *> q;
        q.push(root);

        while(!q.empty())
        {
```

```
                node *temp=q.front();
                q.pop();

                if(temp->left==NULL)
                {

                        temp->left=new node;
                        temp->left->left=NULL;
                        temp->left->right=NULL;
                        temp->left->data=data;
                        return root;
                }
                else
                {

                q.push(temp->left);

                }

                if(temp->right==NULL)
                {

                        temp->right=new node;
                        temp->right->left=NULL;
                        temp->right->right=NULL;
                        temp->right->data=data;
                        return root;
                }
                else
                {

                q.push(temp->right);

                }

        }

}


void bfs(node *head)
{
```

```
        queue<node*> q;
        q.push(head);

        int qSize;

        while (!q.empty())
        {
            qSize = q.size();
            #pragma omp parallel for
    //creates parallel threads
            for (int i = 0; i < qSize; i++)
            {
                node* currNode;
                #pragma omp critical
                {
                 currNode = q.front();
                 q.pop();
                 cout<<"\t"<<currNode->data;

                }// prints parent node
                #pragma omp critical
                {
                if(currNode->left)// push parent's left node in queue
                    q.push(currNode->left);
                if(currNode->right)
                    q.push(currNode->right);
                }// push parent's right node in queue

            }
        }

}

int main(){

    node *root=NULL;
    int data;
    char ans;

    do
    {
        cout<<"\n enter data=>";
        cin>>data;
```

```
            root=insert(root,data);

            cout<<"do you want insert one more node?";
            cin>>ans;

        }while(ans=='y'||ans=='Y');

        bfs(root);

        return 0;
}
```

Steps to compile and run

C:/MinGw/Bin>g++ -fopenmp reduction.cpp
C:/MinGw/Bin>a

DFS

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

using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
        stack<int> s;
        s.push(node);

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

        if (!visited[curr_node]) {
        visited[curr_node] = true;

        s.pop();
        cout<<curr_node<<" ";

        #pragma omp parallel for
        for (int i = 0; i < graph[curr_node].size(); i++) {
```

```
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                s.push(adj_node);
                }
        }
        }
        }
}

int main() {
        int n, m, start_node;
        cout<<"Enter no. of Node,no. of Edges and Starting Node of graph:\n";
        cin >> n >> m >> start_node;
       //n: node,m:edges
      cout<<"Enter pair of node and edges:\n";

        for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;

//u and v: Pair of edges
        graph[u].push_back(v);
        graph[v].push_back(u);
        }

        #pragma omp parallel for
        for (int i = 0; i < n; i++) {
        visited[i] = false;
        }

        dfs(start_node);



        return 0;
}
```