HPC Practical 1

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

using namespace std;

void parallel_bfs(vector<vector<int> >& graph, int start_node
    int num_nodes = graph.size();
    vector<bool> visited(num_nodes, false);
    visited[start_node] = true;
    queue<int> q;
    q.push(start_node);

    while (!q.empty()) {
        int current;
        #pragma omp parallel shared(q, visited) private(curre
        {
            #pragma omp critical
            {
                current = q.front();
                q.pop();
            }

            // Print the current node being visited
            cout << "Visiting Node: " << current << endl;

            #pragma omp for
            for (int i = 0; i < graph[current].size(); ++i) {
                int neighbor = graph[current][i];
                if (!visited[neighbor]) {
                    #pragma omp critical
                    {
                        visited[neighbor] = true;
```

```
                        q.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 BFS: ";
    cin >> start_node;

    parallel_bfs(graph, start_node);

    return 0;
}
```

```
E:\Final Practical\BFSalternati

Enter the number of nodes: 7
Enter the adjacency list for each node (enter -1 to stop):
Node 0: 1 2 -1
Node 1: 0 3 4 -1
Node 2: 0 5 6 -1
Node 3: 1 -1
Node 4: 1 -1
Node 5: 2 -1
Node 6: 2 -1
Enter the starting node for BFS: 0
Visiting Node: 0
Visiting Node: 1
Visiting Node: 2
Visiting Node: 3
Visiting Node: 4
Visiting Node: 5
Visiting Node: 6

--------------------------------
Process exited after 53.8 seconds with return value 0
Press any key to continue . . .
```

Explanation:

- The graph is represented using an adjacency list input by the user.
- The starting node for the BFS traversal is node 0.
- The BFS traversal starts from node 0 and visits its neighbors (1 and 2).
- Then, it visits the neighbors of node 1 (0, 3, and 4), and the neighbors of node 2 (0, 5, and 6).
- The traversal continues until all reachable nodes are visited, following the Breadth First Search algorithm.

You

what if i select starting node as 2

‹ 2 / 2 ›

ChatGPT

If you select the starting node as 2, the BFS traversal will start from node 2 and visit its neighbors. Here's how the input and output would look:

Input:

yaml Copy code

```yaml
Enter the number of nodes: 7
Enter the adjacency list for each node (enter -1 to stop):
Node 0: 1 2 -1
Node 1: 0 3 4 -1
Node 2: 0 5 6 -1
Node 3: 1 -1
Node 4: 1 -1
Node 5: 2 -1
Node 6: 2 -1
Enter the starting node for BFS: 2
```

yaml Copy code

```yaml
Visiting Node: 2
Visiting Node: 0
Visiting Node: 5
Visiting Node: 6
Visiting Node: 1
Visiting Node: 3
Visiting Node: 4
```

Let's break down the BFS traversal step by step for the given example when starting from node 2:

1. **Start at Node 2 (Starting Node)**:
   - Mark Node 2 as visited.

- Enqueue Node 2.

2. **Visit Neighbors of Node 2**:
   - Dequeue Node 2.
   - Visit Node 0 (Neighbor of Node 2): Mark Node 0 as visited, enqueue Node 0.
   - Visit Node 5 (Neighbor of Node 2): Mark Node 5 as visited, enqueue Node 5.
   - Visit Node 6 (Neighbor of Node 2): Mark Node 6 as visited, enqueue Node 6.
3. **Visit Neighbors of Node 0**:
   - Dequeue Node 0.
   - Visit Node 1 (Neighbor of Node 0): Mark Node 1 as visited, enqueue Node 1.
   - Visit Node 2 (Neighbor of Node 0): Already visited, skip.
4. **Visit Neighbors of Node 5**:
   - Dequeue Node 5.
   - Visit Node 2 (Neighbor of Node 5): Already visited, skip.
5. **Visit Neighbors of Node 6**:
   - Dequeue Node 6.
   - Visit Node 2 (Neighbor of Node 6): Already visited, skip.
6. **Visit Neighbors of Node 1**:
   - Dequeue Node 1.
   - Visit Node 0 (Neighbor of Node 1): Already visited, skip.
   - Visit Node 3 (Neighbor of Node 1): Mark Node 3 as visited, enqueue Node 3.
   - Visit Node 4 (Neighbor of Node 1): Mark Node 4 as visited, enqueue Node 4.
7. **Visit Neighbors of Node 3**:
   - Dequeue Node 3.

- No new neighbors to visit.

8. **Visit Neighbors of Node 4**:
    - Dequeue Node 4.
    - No new neighbors to visit.
9. **BFS Traversal Complete**:
    - All reachable nodes from Node 2 have been visited.

This process illustrates how the BFS traversal explores the graph starting from Node 2, visiting its neighbors, then visiting the neighbors of its neighbors, and so on, until all reachable nodes are visited.

Code Explanation

1. **Headers**: These are like tools you bring into your program. `#include <iostream>` lets you use `cout` and `cin` for printing and reading input, `#include <vector>` lets you use vectors to store data, `#include <queue>` lets you use queues, and `#include <omp.h>` lets you use OpenMP for parallel programming.
2. **Namespace**: This is like a group where similar things are kept. `using namespace std;` means you can use `cout` and `cin` directly without writing `std::` before them.
3. **Function Declaration**: This is like telling the computer, "Hey, I'm going to use this function later." `void parallel_bfs(vector<vector<int> >& graph, int start_node);` says that `parallel_bfs` is a function that takes a graph and a starting node.
4. **Function Definition**: This is where you tell the computer what the function does. The `parallel_bfs` function explores a graph using BFS, starting from a given node. It keeps track of visited nodes and uses a queue to remember which nodes to visit next.
5. **BFS Traversal**: BFS is like exploring a map starting from one point and checking all nearby places before moving further away. The function keeps track of which nodes it has visited so it doesn't visit the same node twice.

6. **Parallel Region**: This is where things get interesting. `#pragma omp parallel` tells the computer to do things in parallel, like having multiple workers doing different parts of the job at the same time.
7. **Critical Section**: Imagine two people trying to write on the same paper at the same time - it would be messy. The critical section (`#pragma omp critical`) makes sure that only one person (or thread) can access the paper (or data) at a time, avoiding messiness.
8. **Neighbor Exploration**: Each node has neighbors, which are nodes connected to it. The code uses a loop to go through all the neighbors of the current node and marks them as visited if they haven't been visited before.
9. **Main Function**: This is where everything starts. The `main` function asks the user for the number of nodes and the connections between them. It then starts the BFS traversal from a user-specified starting node.
10. **Explanation**: The code demonstrates how to use OpenMP to make the BFS traversal faster by doing parts of the traversal at the same time with multiple threads. This can be useful for exploring large graphs more quickly, especially on computers with multiple processors.

In simpler terms, the code is like a team of workers exploring a map together. Each worker checks different places, but they talk to each other to make sure they don't visit the same place twice. This teamwork helps them explore the map faster.

Theory for Practical

📌 What is BFS

BFS stands for Breadth First Search. It's an algorithm used to traverse or search tree or graph data structures. The algorithm starts at a given node, which is called the root node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Here's a simple explanation of how BFS works:

1. Start at the root node and add it to a queue.

2. While the queue is not empty, remove the first node from the queue.
3. Visit the removed node.
4. Add all of its neighbors to the queue (if they haven't been visited yet) and mark them as visited.
5. Repeat steps 2-4 until the queue is empty.

BFS is often used to find the shortest path between two nodes or to visit all nodes in a graph or tree. It guarantees to find the shortest path in an unweighted graph because it traverses the graph level by level.

📌 What is Parallel Breadth First Search

Parallel Breadth First Search (BFS) is an algorithm that explores a graph or tree data structure in parallel to find all the nodes at the same level from a given starting node. It uses multiple threads or processes to visit nodes at the same level concurrently, which can significantly improve performance, especially for large graphs.

Here's a basic outline of how Parallel BFS works:

1. **Initialization**: Each thread/process is assigned a subset of nodes to explore. The starting node is assigned to the first thread/process.
2. **Exploration**: Each thread/process explores its assigned nodes in parallel. They use a shared data structure (like a queue) to keep track of nodes to visit next.
3. **Communication**: Threads/processes communicate with each other to avoid visiting the same node multiple times. This is typically done using synchronization mechanisms like mutexes or atomic operations.
4. **Termination**: The algorithm terminates when all nodes at the same level have been visited by all threads/processes.

Parallel BFS can be implemented using various parallel programming paradigms, such as OpenMP for shared-memory systems, MPI for distributed-memory systems, or CUDA for GPU-accelerated systems. The key idea is to divide the work among multiple threads or processes to exploit parallelism and speed up the traversal process.

📌 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 map with lots of places to visit, like a treasure hunt map. You start at one place and want to visit all the places around it first before moving on to the places farther away.

In the code, the computer is like a little explorer that starts at one place on the map (called a node) and wants to explore all the nearby places first. The computer keeps a list of the places it wants to visit next in a special list (called a queue).

The computer sends out little helpers (threads) to explore different places at the same time. Each helper checks the list of places to visit (queue) and goes to one of the places that hasn't been visited yet. If the helper finds a new place, it adds that place to the list of places to visit (queue) so that it can be explored later.

The computer also has a special list (array) to remember which places have already been visited, so it doesn't visit the same place twice.

By sending out helpers to explore different places at the same time, the computer can explore the map faster and find all the hidden treasures more quickly.

Viva Questions ❓ ❓

1. **Question:** Explain the purpose of using OpenMP in the Parallel BFS code.
    - **Answer:** OpenMP is used to parallelize the BFS traversal, allowing multiple threads to explore different parts of the graph concurrently. This can significantly improve the performance of the traversal, especially for large graphs.

2. **Question:** How does the code ensure that nodes are visited in a breadth-first manner?
    - **Answer:** The code uses a queue to store nodes to be visited. It dequeues a node and then enqueues its unvisited neighbors, ensuring that nodes at the same level are visited before moving to the next level.

3. **Question:** Describe the role of the `visited` array in the BFS traversal.
    - **Answer:** The `visited` array is used to keep track of nodes that have been visited. It prevents the algorithm from visiting the same node multiple times, ensuring that each node is visited only once.

4. **Question:** How does the `#pragma omp parallel` directive affect the execution of the code?
    - **Answer:** The `#pragma omp parallel` directive creates a team of threads that execute the enclosed code block in parallel. Each thread explores a different part of the graph, speeding up the BFS traversal.

5. **Question:** Why is it important to use synchronization mechanisms like `#pragma omp critical` in the code?
    - **Answer:** Synchronization mechanisms like `#pragma omp critical` are used to prevent race conditions. They ensure that only one thread at a time can access shared resources like the queue and the `visited` array, avoiding conflicts.

6. **Question:** Discuss the advantages of parallelizing the BFS traversal using OpenMP.
    - **Answer:** Parallelizing the BFS traversal using OpenMP can improve performance by utilizing multiple processors or cores. It can also reduce the overall execution time, especially for large graphs.

7. **Question:** How does the code handle the parallel exploration of neighbors of different nodes?
    - **Answer:** Each thread/process is assigned a subset of nodes to explore. They use synchronization mechanisms like `#pragma omp critical` to avoid conflicts when exploring neighboring nodes.

8. **Question:** What would happen if the code did not use synchronization mechanisms like `#pragma omp critical`?
    - **Answer:** Without synchronization mechanisms, multiple threads could access and modify shared resources concurrently, leading to data corruption or incorrect results.

9. **Question:** How would you extend this code to handle graphs with weighted edges?
    - **Answer:** To handle graphs with weighted edges, you would need to modify the data structure representing the graph to include edge weights. The traversal algorithm would then need to consider these weights when determining the order in which nodes are visited.