Parallel Breadth First Search

Title of the Assignment: Design and implement Parallel Breadth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS

Objective of the Assignment: Students should be able to perform Parallel Breadth First Search based on existing algorithms using OpenMP

Prerequisite:

1. Basic of programming language
2. Concept of BFS
3. Concept of Parallelism

Contents for Theory:

1. **What is BFS?**
2. **Example of BFS**
3. **Concept of OpenMP**
4. **How Parallel BFS Work**
5. **Code Explanation with Output**

What is BFS?

BFS stands for Breadth-First Search. It is a graph traversal algorithm used to explore all the nodes of a graph or tree systematically, starting from the root node or a specified starting point, and visiting all the neighboring nodes at the current depth level before moving on to the next depth level.

The algorithm uses a queue data structure to keep track of the nodes that need to be visited, and marks each visited node to avoid processing it again. The basic idea of the BFS algorithm is to visit all the nodes at a given level before moving on to the next level, which ensures that all the nodes are visited in breadth-first order.

BFS is commonly used in many applications, such as finding the shortest path between two nodes, solving puzzles, and searching through a tree or graph.

Example of BFS

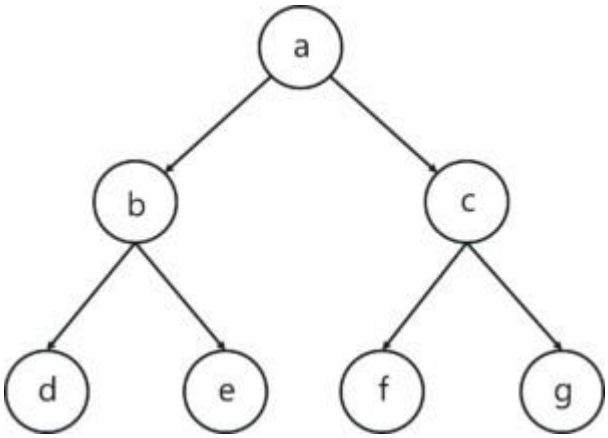
Now let's take a look at the steps involved in traversing a graph by using Breadth-First Search:

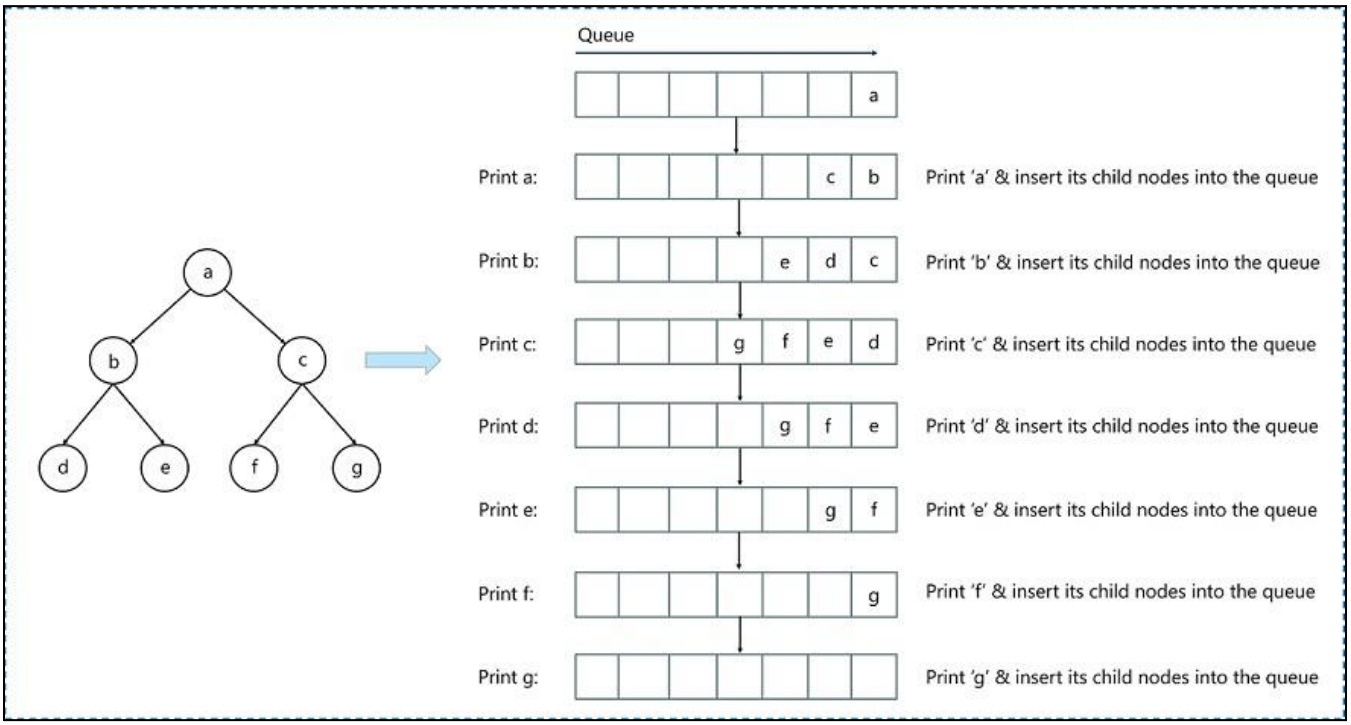
Step 1: Take an Empty Queue.

Step 2: Select a starting node (visiting a node) and insert it into the Queue.

Step 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

Step 4: Print the extracted node.





Concept of OpenMP

- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.
- OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.
- OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.
- OpenMP is widely used in scientific computing, engineering, and other fields that require high-performance computing. It is supported by most modern compilers and is available on a widerange of platforms, including desktops, servers, and supercomputers.

How Parallel BFS Work

- Parallel BFS (Breadth-First Search) is an algorithm used to explore all the nodes of a graph or tree systematically in parallel. It is a popular parallel algorithm used for graph traversal in distributed computing, shared-memory systems, and parallel clusters.
- The parallel BFS algorithm starts by selecting a root node or a specified starting point, and then assigning it to a thread or processor in the system. Each thread maintains a local queue of nodes to be visited and marks each visited node to avoid processing it again.
- The algorithm then proceeds in levels, where each level represents a set of nodes that are at a certain distance from the root node. Each thread processes the nodes in its local queue at the current level, and then exchanges the nodes that are adjacent to the current level with other threads or processors. This is done to ensure that the nodes at the next level are visited by the next iteration of the algorithm.
- The parallel BFS algorithm uses two phases: the computation phase and the communication phase. In the computation phase, each thread processes the nodes in its local queue, while in the communication phase, the threads exchange the nodes that are adjacent to the current level with other threads or processors.
- The parallel BFS algorithm terminates when all nodes have been visited or when a specified node has been found. The result of the algorithm is the set of visited nodes or the shortest path from the root node to the target node.
- Parallel BFS can be implemented using different parallel programming models, such as OpenMP, MPI, CUDA, and others. The performance of the algorithm depends on the number of threads or processors used, the size of the graph, and the communication overhead between the threads or processors.

Conclusion- In this way we can achieve parallelism while implementing BFS

Assignment Question

1. **What if BFS?**
2. **What is OpenMP?What is its significance in parallel programming?**
3. **Write down applications of Parallel BFS**
4. **How can BFS be parallelized using OpenMP? Describe the parallel BFS algorithm using OpenMP.**
5. **Write Down Commands used in OpenMP?**

Reference link

- https://www.edureka.co/blog/breadth-first-search-algorithm/

Output:

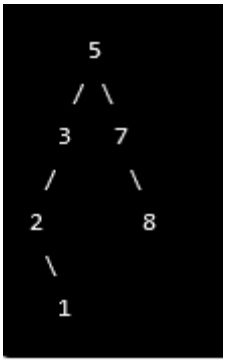
This code represents a breadth-first search (BFS) algorithm on a binary tree using OpenMP for parallelization. The program asks for user input to insert nodes into the binary tree and then performs the BFS algorithm using multiple threads. Here's an example output for a binary tree with nodes 5, 3, 2, 1, 7, and 8:

The nodes are printed in breadth-first order. The #pragma omp parallel for statement is used to parallelize the for loop that processes each level of the binary tree. The #pragma omp critical statement is used to synchronize access to shared data structures, such as the queue that stores the nodes of the binary tree.

Here is an example of the breadth-first traversal for a binary tree with the values 5, 3, 2, 1, 7, and 8:

```
Enter data => 5
Do you want to insert one more node? (y/n) y

Enter data => 3
Do you want to insert one more node? (y/n) y

Enter data => 2
Do you want to insert one more node? (y/n) y

Enter data => 1
Do you want to insert one more node? (y/n) y

Enter data => 7
Do you want to insert one more node? (y/n) y

Enter data => 8
Do you want to insert one more node? (y/n) n

        5       3       7       2       1       8
```

Starting with the root node containing value 5:



The traversal would be:

```
5, 3, 7, 2, 8, 1
```

Explanation

This C++ code demonstrates how to perform a breadth-first search (BFS) in a binary tree using OpenMP parallel programming.

1. The program starts by defining a class "node" that defines the properties of a binary tree node. This class has two pointers to the left and right child nodes of the current node, and an integer to store the data value of the node.
2. Next, a class named "Breadthfs" is defined, which contains two methods - insert() and bfs(). The insert() method is used to insert a new node in the binary tree, while the bfs() method is used to perform the BFS algorithm on the binary tree.
3. The insert() method takes two arguments - a pointer to the root node of the binary tree and an integer value to be inserted. If the root node is null, the method creates a new node, sets its data value to the given integer value and returns the root node.
4. If the root node is not null, the method creates an empty queue of node pointers and pushes the root node into the queue. It then enters a loop that runs until the queue is empty.
5. Inside the loop, the method dequeues the front node from the queue and checks if its left child is null. If it is null, the method creates a new node, sets its data value to the given integer value, and returns the root node.
6. If the left child of the front node is not null, the method pushes it onto the queue. The method then checks if the right child of the front node is null. If it is null, the method creates a new node, sets its data value to the given integer value, and returns the root node.
7. If the right child of the front node is not null, the method pushes it onto the queue.

8. The bfs() method takes a pointer to the root node of the binary tree as its argument. It creates an empty queue of node pointers and pushes the root node into the queue.
9. It then enters a loop that runs until the queue is empty. Inside the loop, it retrieves the size of the queue and creates an OpenMP parallel region using the "omp parallel for" directive. This directive creates a team of parallel threads to execute the loop body in parallel.
10. Inside the loop body, each thread dequeues a node from the queue using a critical section to ensure that no two threads access the same node simultaneously. It prints the data value of the current node to the console.
11. The thread then checks if the left and right child nodes of the current node are not null. If they are not null, the thread uses another critical section to push the left and right child nodes onto the queue.
12. The main function starts by initializing the root node pointer to null and declaring an integer variable to store the user input.
13. It uses a do-while loop to prompt the user to enter a value to be inserted in the binary tree. If the user enters 'y' or 'Y', the loop continues to accept more input. Otherwise, the loop terminates.
14. For each user input value, the program calls the insert() method to insert a new node in the binary tree.
15. After the user is finished inputting values, the program calls the bfs() method to perform a breadth-first search on the binary tree.
16. Finally, the program returns 0 to indicate successful execution.

Explanation 2

This C++ code implements the Breadth-First Search (BFS) algorithm to traverse a binary tree. Here is a step-by-step explanation of the code's execution flow:

1. The code defines a node class with left, right, and data members, which are pointers to node, pointers to the left and right child nodes, and the data to be stored in each node, respectively.
2. The code defines a Breadthfs class with insert and bfs member functions, which are responsible for inserting a new node into the binary tree and traversing the tree in a breadth-first manner, respectively.
3. The insert function takes two arguments: a pointer to the root node of the tree and an integer value to be inserted. If the root node is NULL, it creates a new node with the given value and returns the new node. Otherwise, it uses a queue to traverse the tree level by level, looking for the first empty child node (either left or right). When an empty node is found, it creates a new node with the given value and returns the root node.

4. The bfs function takes a pointer to the root node of the tree and performs a breadth-first traversal of the tree. It starts by initializing a queue with the root node and a variable to store the size of the queue. Then, it enters a loop that continues until the queue is empty.
5. Inside the loop, it obtains the current size of the queue, and for each node in the queue, it pops the front node and prints its data value. It then adds the node's left and right child nodes to the queue if they exist.
6. To improve the performance of the BFS traversal, the loop that processes each node is parallelized using OpenMP, a library that enables parallel programming in C++. The #pragma omp parallel for directive creates multiple threads that execute the loop iterations in parallel, and the #pragma omp critical directive ensures that only one thread at a time can access the shared resources (i.e., the queue and the console output).
7. Finally, the main function initializes a pointer to the root node and prompts the user to enter integer values to be inserted into the tree. It uses the insert function to create a new node for each value and adds it to the tree. It continues until the user chooses to stop inserting new values. Then, it calls the bfs function to traverse the tree in a breadth-first manner and prints the data values of each node.