```cpp
#include<iostream>
#include<stdlib.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;
```

```cpp
    }

    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

                }
```

```cpp
        }

}

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;
}
```

Run Commands:
1) g++ -fopenmp bfs.cpp -o bfs

2) ./bfs

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:

```
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
```

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:

Starting with the root node containing value 5:

```
        5
       / \
      3   7
     /     \
    2       8
     \
      1
```
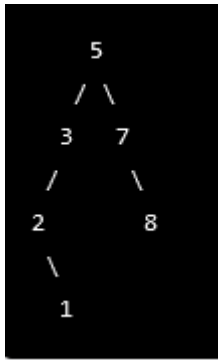
The traversal would be:

```
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.