BFS(Parallel Breadth first Search)

```
class node
{
  public:
        node *left, *right;
        int data;
};
```

This is a C++ class definition for a node in a binary tree. Each node has three data members: a pointer to its left child node, a pointer to its right child node, and an integer value stored in the node.

```
class Breadthfs
{
 public:
 node *insert(node *, int);
 void bfs(node *);
 };
```

This is a C++ class definition for a breadth-first search algorithm on a binary tree. The class contains two public member functions:

1. `insert`: a function that takes a pointer to a node and an integer value, and returns a pointer to the root node of the modified binary tree after inserting the new node.
2. `bfs`: a function that takes a pointer to the root node of a binary tree and performs a breadth-first search traversal, visiting each node in the tree level by level.

node *insert(node *root, int data)

This is a C++ function definition for inserting a new node with the given integer data into a binary tree. The function takes two parameters:

1. `root`: a pointer to the root node of the binary tree where the new node will be inserted.
2. `data`: an integer value to be stored in the new node.

The function returns a pointer to the root node of the modified binary tree after inserting the new 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);
```

This is the beginning of the `insert` function in C++.

The first `if` statement checks if the root node of the binary tree is null. If it is, the function creates a new node with the given data value and sets the left and right pointers to null, then returns a pointer to the new root node.

If the root node is not null, the function creates a queue of node pointers and adds the root node to the queue. The queue will be used to perform a level-order traversal of the binary tree to find the appropriate position to insert the new node.

```
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);
                }
```

This is the body of the while loop in the `insert` function that performs the level-order traversal of the binary tree to find the appropriate position to insert the new node.

First, the function retrieves the front node pointer from the queue and removes it from the queue using `q.pop()`.

Next, the function checks if the left child of the current node is null. If it is, the function creates a new node with the given data value, sets its left and right pointers to null, and attaches it as the left child of the current node. Finally, the function returns a pointer to the root node of the modified binary tree.

If the left child of the current node is not null, the function adds the left child to the back of the queue using `q.push(temp->left)`, so that it can be processed later in the traversal.

```
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);
                }
        }
}
```

This is the final part of the while loop in the `insert` function that performs the level-order traversal of the binary tree to find the appropriate position to insert the new node.

If the left child of the current node is not null, the function checks if the right child of the current node is null. If it is, the function creates a new node with the given data value, sets its left and right pointers to null, and attaches it as the right child of the current node. Finally, the function returns a pointer to the root node of the modified binary tree.

If the right child of the current node is not null, the function adds the right child to the back of the queue using `q.push(temp->right)`, so that it can be processed later in the traversal.

If the while loop completes without finding an appropriate position to insert the new node, the function simply returns a pointer to the root node of the original binary tree without making any modifications.

```
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
                }
        }
```

```
}
```

This is a C++ function definition for performing a breadth-first search (BFS) traversal of a binary tree in parallel using OpenMP. The function takes a pointer to the root node of the binary tree as input.

The function creates a queue of node pointers and adds the root node to the queue. It then enters a while loop that continues until the queue is empty. Within the while loop, the function retrieves the size of the queue and uses it to control a parallel for loop that iterates over the nodes at the current level of the binary tree.

Within the parallel for loop, each thread retrieves a node pointer from the queue and prints its data value to the console. The function uses OpenMP critical sections to ensure that only one thread at a time is modifying the queue or printing to the console.

After printing the data value of each node, the function checks if the node has a left or right child, and adds them to the queue if they exist. This ensures that the BFS traversal continues to the next level of the binary tree.

Overall, this function enables parallel processing of the BFS traversal, which can improve performance on multi-core systems.

```
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;
```

```
}
```

This is a C++ program that creates a binary tree by repeatedly prompting the user to enter a new data value for a node until they indicate they are finished. The program calls the `insert` function to add each node to the binary tree, passing in the root node pointer and the data value entered by the user.

After the binary tree is constructed, the program calls the `bfs` function to perform a breadth-first search (BFS) traversal of the tree and print the data values of each node in the order they are visited.

The program returns 0 to indicate successful completion.