Code:

```
#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();
        s.pop();
        if (!visited[curr_node]) {
            visited[curr_node] = true;
            if (visited[curr_node]) {
                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,Edges,and start node:" ;
    cin >> n >> m >> start_node;

    cout << "Enter Pair of edges:" ;
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;

        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;
}
```

Output:

```
Enter No of Node,Edges,and start node:5 5 2
Enter Pair of edges: 2 3
3 4
4 5
5 6
6 2
2 6 5 4 3
```

Code:

```
#include <iostream>
#include <stdlib.h>
#include <queue>
using namespace std;

class Node {
public:
    Node *left, *right;
    int data;
    Node(int data) {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }
};

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

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

    if (!root) {

        root = new Node(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(data);
            return root;
        }
        else {
            q.push(temp->left);
        }

        if (temp->right == NULL) {
            temp->right = new Node(data);
            return root;
        }
        else {
            q.push(temp->right);
        }
    }
    return NULL;
}

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();

                // prints parent Node
                cout << "\t" << currNode->data;

            }

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

                // push parent's right Node in queue
                if (currNode->right)
                    q.push(currNode->right);
            }
        }
    }
}

int main() {
    Node *root = NULL;
    int data;
    char ans;

    do {
        cout << "Enter node's data: ";
        cin >> data;

        root = insert(root, data);

        cout << "Do you want insert one more Node? ('y' | 'n'): ";
        cin >> ans;

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

    bfs(root);

    return 0;
}
```

Output:

```
Enter node's data: 3
Do you want insert one more Node? ('y' | 'n'): y

Enter node's data: 4
Do you want insert one more Node? ('y' | 'n'): y

Enter node's data: 2
Do you want insert one more Node? ('y' | 'n'): y

Enter node's data: 6
Do you want insert one more Node? ('y' | 'n'): y

Enter node's data: 7
Do you want insert one more Node? ('y' | 'n'): n
        3       4       2       6       7
```

Code:

```
#include <iostream>
#include <stdlib.h>
#include <omp.h>
using namespace std;

void merge(int a[], int i1, int j1, int i2, int j2) {
    int temp[1000];
    int i, j, k;
    i = i1;
    j = i2;
    k = 0;

    while (i <= j1 && j <= j2) {
        if (a[i] < a[j]) {
            temp[k++] = a[i++];
        }
        else {
            temp[k++] = a[j++];
        }
    }

    while (i <= j1) {
        temp[k++] = a[i++];
    }

    while (j <= j2) {
        temp[k++] = a[j++];
    }

    for (i = i1, j = 0; i <= j2; i++, j++) {
        a[i] = temp[j];
    }
}

void mergesort(int a[], int i, int j) {
    int mid;
    if (i < j) {
        mid = (i + j) / 2;

        #pragma omp parallel sections
        {

            #pragma omp section
            {
                mergesort(a, i, mid);
            }

            #pragma omp section
            {
                mergesort(a, mid + 1, j);
            }
        }

        merge(a, i, mid, mid + 1, j);
    }
}

int main() {
    int *a, n, i;
    cout << "Enter total no of elements: ";
    cin >> n;
    a = new int[n];

    cout << "Enter elements: ";
```

```
    for (i = 0; i < n; i++) {
        cin >> a[i];
    }

    mergesort(a, 0, n - 1);

    cout << "\nSorted array is: \n";
    for (i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Enter total no of elements: 5
Enter elements: 5 2 1 4 3
Sorted array is:
1 2 3 4 5
```

Code:

```
#include <iostream>
#include <stdlib.h>
#include <omp.h>
using namespace std;

void bubble(int *a, int n) {
    for (int i = 0; i < n; i++) {
        int first = i % 2;

        #pragma omp parallel for shared(a, first)
        for (int j = first; j < n - 1; j += 2) {

            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]);
            }
        }
    }
}

int main() {

    int *a, n;
    cout << "Enter total no of elements: ";
    cin >> n;
    a = new int[n];

    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    bubble(a, n);

    cout << "\nSorted array is: \n";
    for (int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Enter total no of elements: 10
Enter elements: 10 9 3 4 2 1 8 6 7 5

Sorted array is:
1 2 3 4 5 6 7 8 9 10
```

Code:

```
#include <iostream>
#include <omp.h>
#include <climits>
using namespace std;

void min_reduction(int arr[], int n) {
    int min_value = INT_MAX;

    #pragma omp parallel for reduction(min : min_value)
    for (int i = 0; i < n; i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    cout << "Minimum value: " << min_value << endl;
}

void max_reduction(int arr[], int n) {
    int max_value = INT_MIN;

    #pragma omp parallel for reduction(max : max_value)
    for (int i = 0; i < n; i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }
    cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(int arr[], int n) {
    int sum = 0;

    #pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    cout << "Sum: " << sum << endl;
}

void average_reduction(int arr[], int n) {
    int sum = 0;

    #pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    cout << "Average: " << (double)sum / (n - 1) << endl;
}

int main() {
    int *arr, n;
    cout << "Enter total no of elements: ";
    cin >> n;
    arr = new int[n];

    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    min_reduction(arr, n);
    max_reduction(arr, n);
```

```
    sum_reduction(arr, n);
    average_reduction(arr, n);

    return 0;
}
```

Output:

```
Enter total no of elements: 9
Enter elements: 5 2 9 1 7 6 8 3 4
Minimum value: 1
Maximum value: 9
Sum: 45
Average: 5.625
```

Code:

```
#include <iostream>
#include <cuda_runtime.h>
using namespace std;

__global__ void addVectors(int *A, int *B, int *C, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
    {
        C[i] = A[i] + B[i];
    }
}

int main()
{
    int n = 1000000;
    int *A, *B, *C;
    int size = n * sizeof(int);

    // Allocate memory on the host
    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);

    // Initialize the vectors
    for (int i = 0; i < n; i++)
    {
        A[i] = i;
        B[i] = i * 2;
    }
    // Allocate memory on the device
    int *dev_A, *dev_B, *dev_C;
    cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

    // Copy data from host to device
    cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);

    // Launch the kernel
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    addVectors<<<numBlocks, blockSize>>>(dev_A, dev_B, dev_C, n);

    // Copy data from device to host
    cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);

    // Print the results
    for (int i = 0; i < 10; i++)
    {
        cout << C[i] << " ";
    }
    cout << endl;

    // Free memory
    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);

    return 0;
}
```