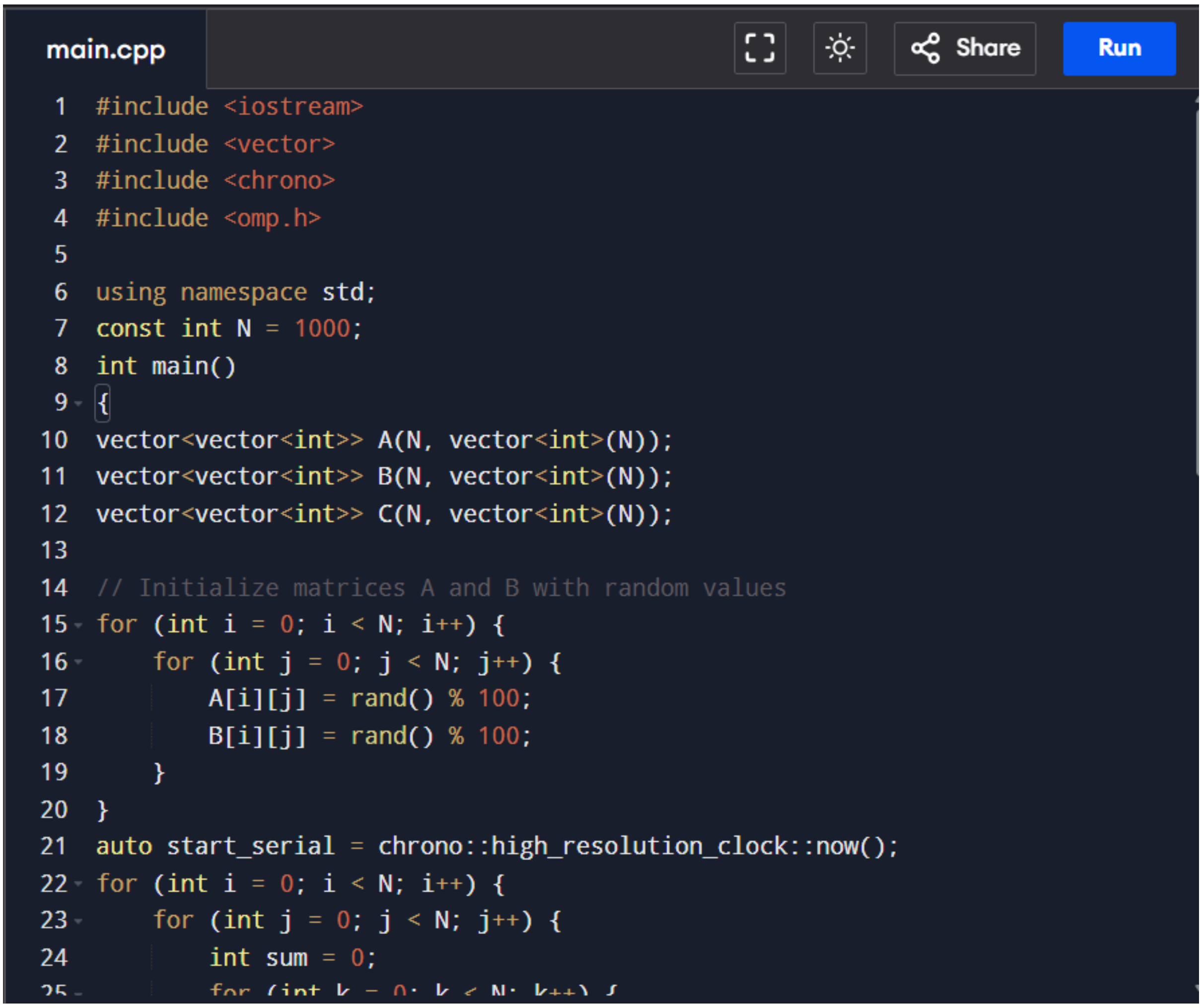
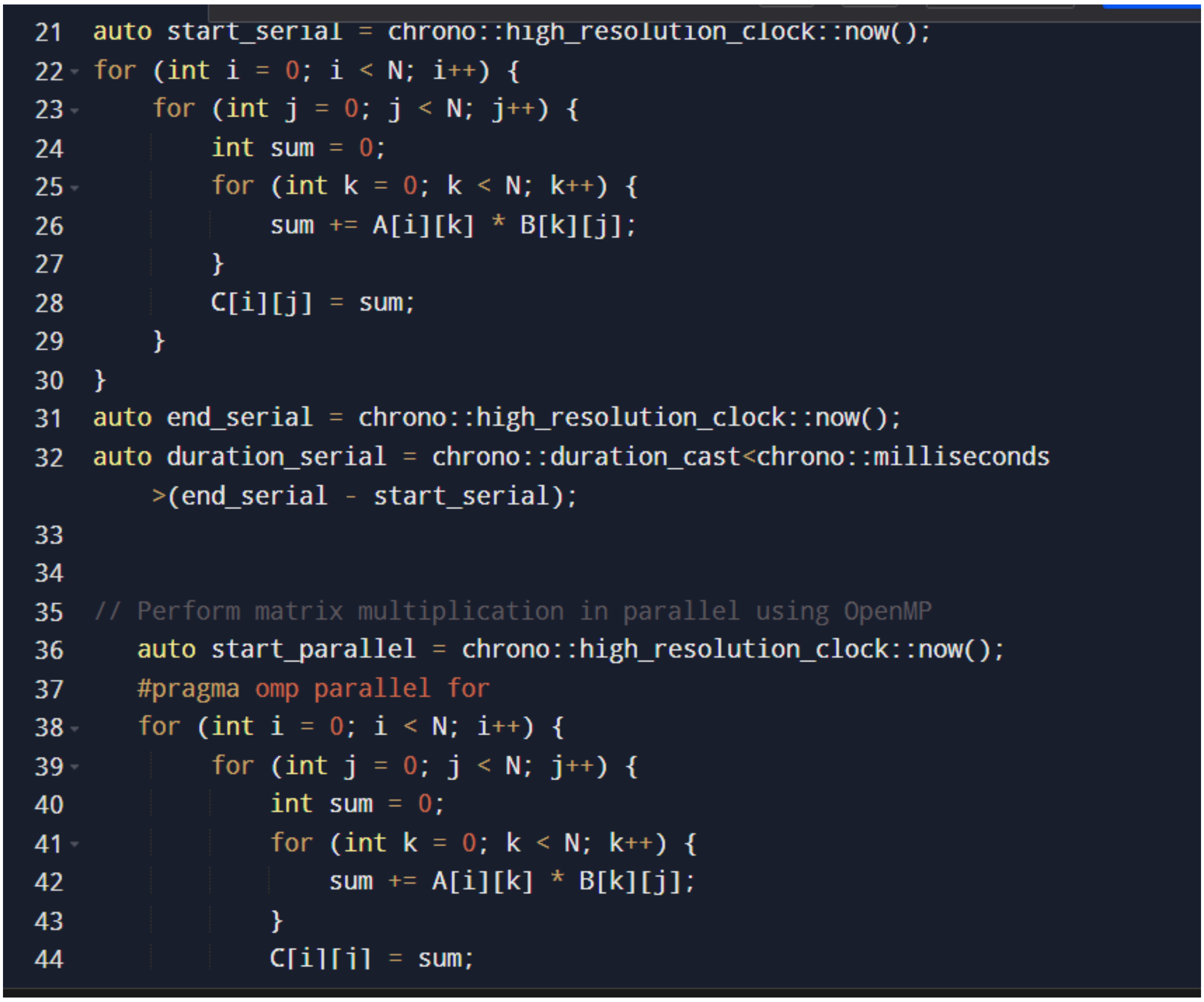
Suggested Practical List for the Introduction to Parallel Programming Paper (DSC18)

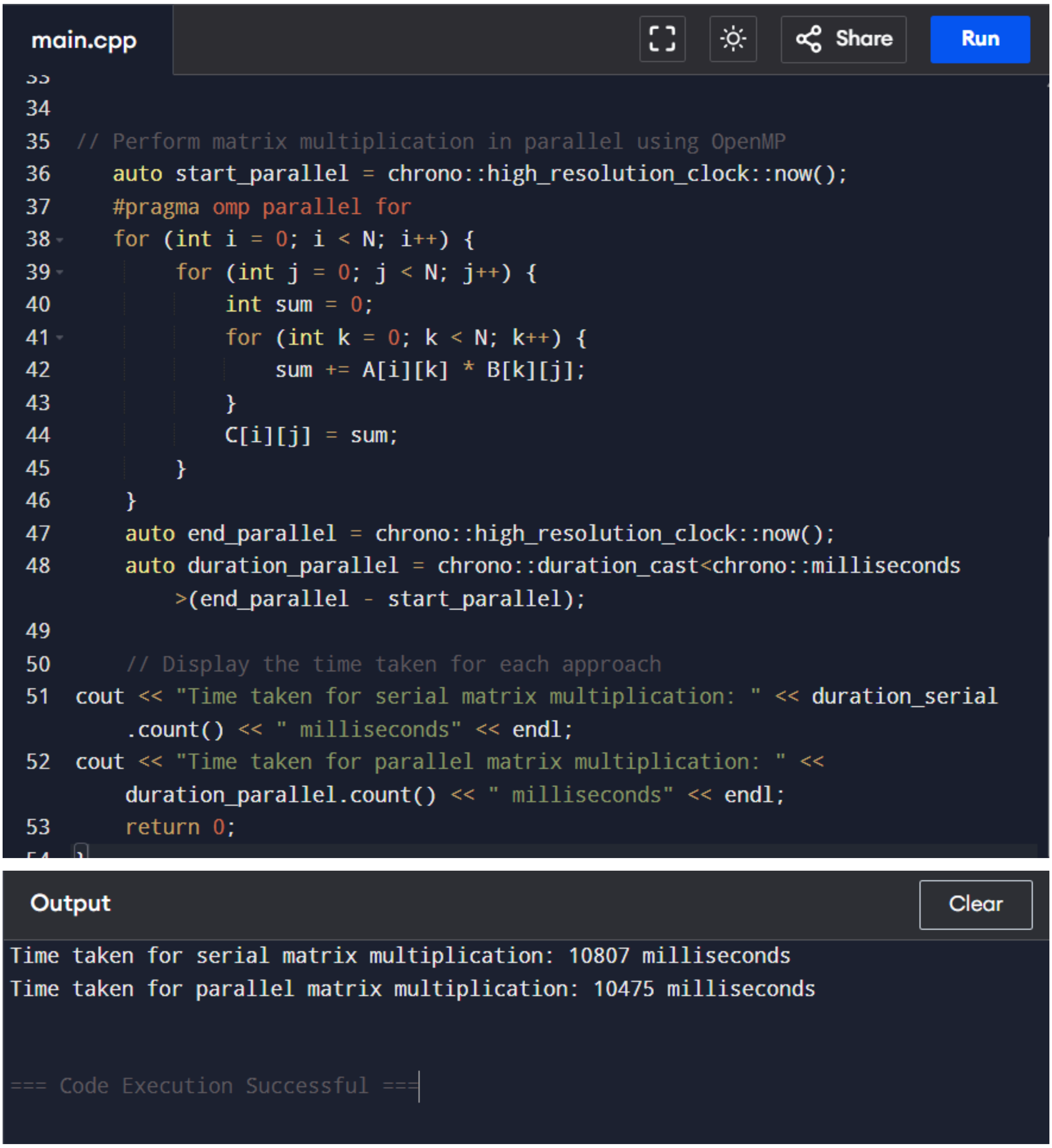
1. Implement matrix-matrix multiplication in parallel using OpenMP



```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <omp.h>

using namespace std;
const int N = 1000;
int main()
{
vector<vector<int>> A(N, vector<int>(N));
vector<vector<int>> B(N, vector<int>(N));
vector<vector<int>> C(N, vector<int>(N));

// Initialize matrices A and B with random values
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[i][j] = rand() % 100;
        B[i][j] = rand() % 100;
    }
}
auto start_serial = chrono::high_resolution_clock::now();
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int sum = 0;
        for (int k = 0; k < N; k++) {
```



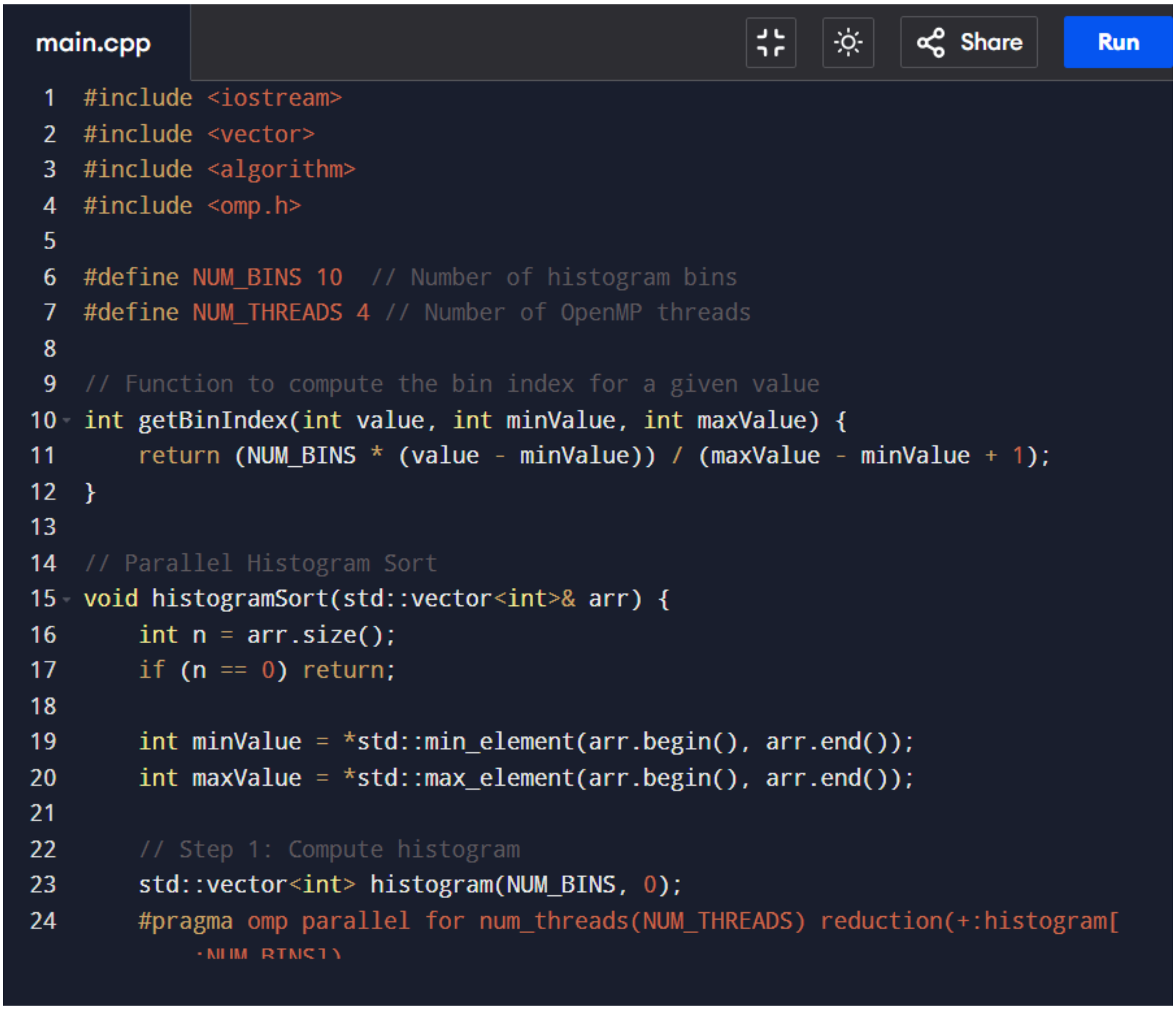
```
auto start_serial = chrono::high_resolution_clock::now();
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int sum = 0;
        for (int k = 0; k < N; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
auto end_serial = chrono::high_resolution_clock::now();
auto duration_serial = chrono::duration_cast<chrono::milliseconds>(end_serial - start_serial);


// Perform matrix multiplication in parallel using OpenMP
   auto start_parallel = chrono::high_resolution_clock::now();
   #pragma omp parallel for
   for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
```



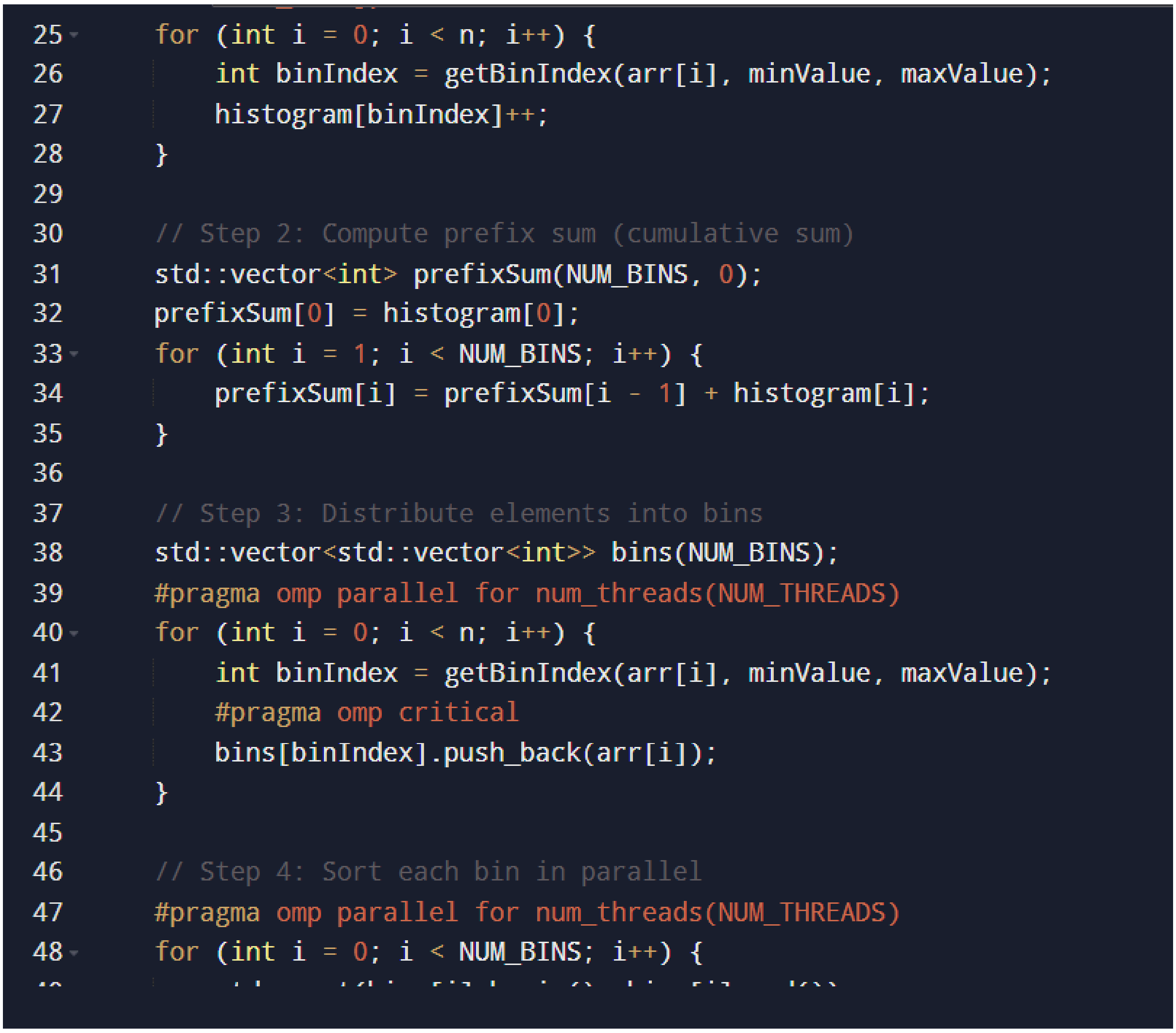
```cpp

// Perform matrix multiplication in parallel using OpenMP
   auto start_parallel = chrono::high_resolution_clock::now();
   #pragma omp parallel for
   for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    auto end_parallel = chrono::high_resolution_clock::now();
    auto duration_parallel = chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);

    // Display the time taken for each approach
cout << "Time taken for serial matrix multiplication: " << duration_serial.count() << " milliseconds" << endl;
cout << "Time taken for parallel matrix multiplication: " << duration_parallel.count() << " milliseconds" << endl;
    return 0;
}
```

```
Time taken for serial matrix multiplication: 10807 milliseconds
Time taken for parallel matrix multiplication: 10475 milliseconds


=== Code Execution Successful ===
```

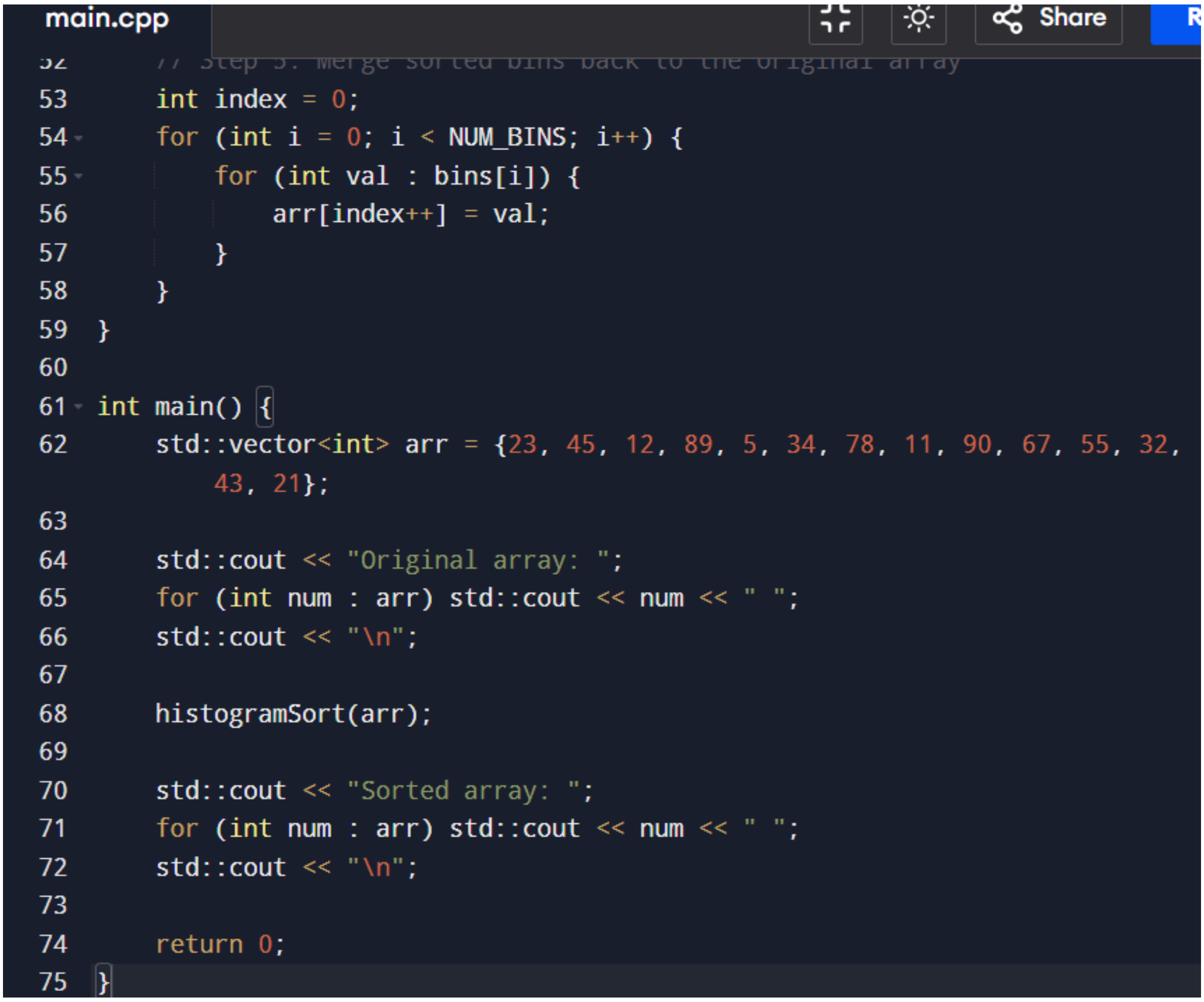
2. Implement distributed histogram sorting in parallel using OpenMP



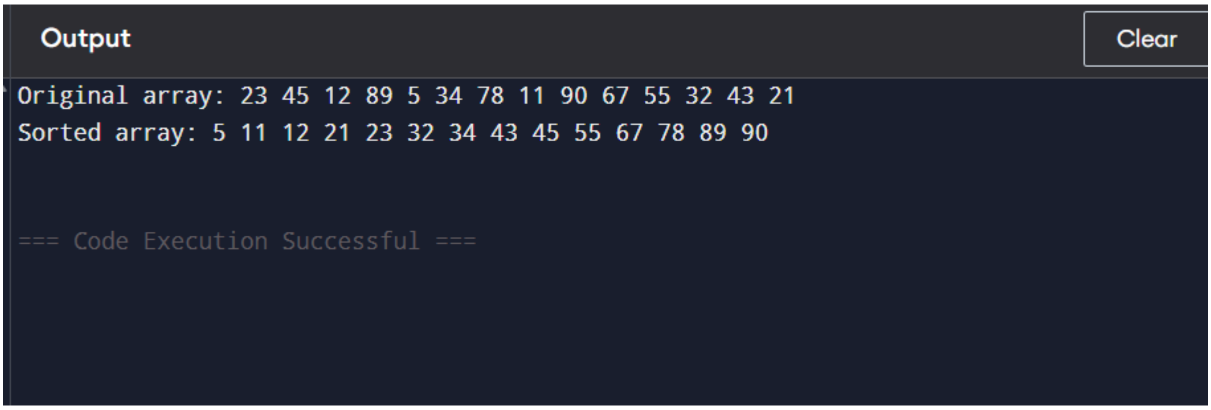
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <omp.h>

#define NUM_BINS 10  // Number of histogram bins
#define NUM_THREADS 4 // Number of OpenMP threads

// Function to compute the bin index for a given value
int getBinIndex(int value, int minValue, int maxValue) {
    return (NUM_BINS * (value - minValue)) / (maxValue - minValue + 1);
}

// Parallel Histogram Sort
void histogramSort(std::vector<int>& arr) {
    int n = arr.size();
    if (n == 0) return;

    int minValue = *std::min_element(arr.begin(), arr.end());
    int maxValue = *std::max_element(arr.begin(), arr.end());

    // Step 1: Compute histogram
    std::vector<int> histogram(NUM_BINS, 0);
    #pragma omp parallel for num_threads(NUM_THREADS) reduction(+:histogram[
        :NUM_BINS])
```



```cpp
    for (int i = 0; i < n; i++) {
        int binIndex = getBinIndex(arr[i], minValue, maxValue);
        histogram[binIndex]++;
    }

    // Step 2: Compute prefix sum (cumulative sum)
    std::vector<int> prefixSum(NUM_BINS, 0);
    prefixSum[0] = histogram[0];
    for (int i = 1; i < NUM_BINS; i++) {
        prefixSum[i] = prefixSum[i - 1] + histogram[i];
    }

    // Step 3: Distribute elements into bins
    std::vector<std::vector<int>> bins(NUM_BINS);
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < n; i++) {
        int binIndex = getBinIndex(arr[i], minValue, maxValue);
        #pragma omp critical
        bins[binIndex].push_back(arr[i]);
    }

    // Step 4: Sort each bin in parallel
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < NUM_BINS; i++) {
```

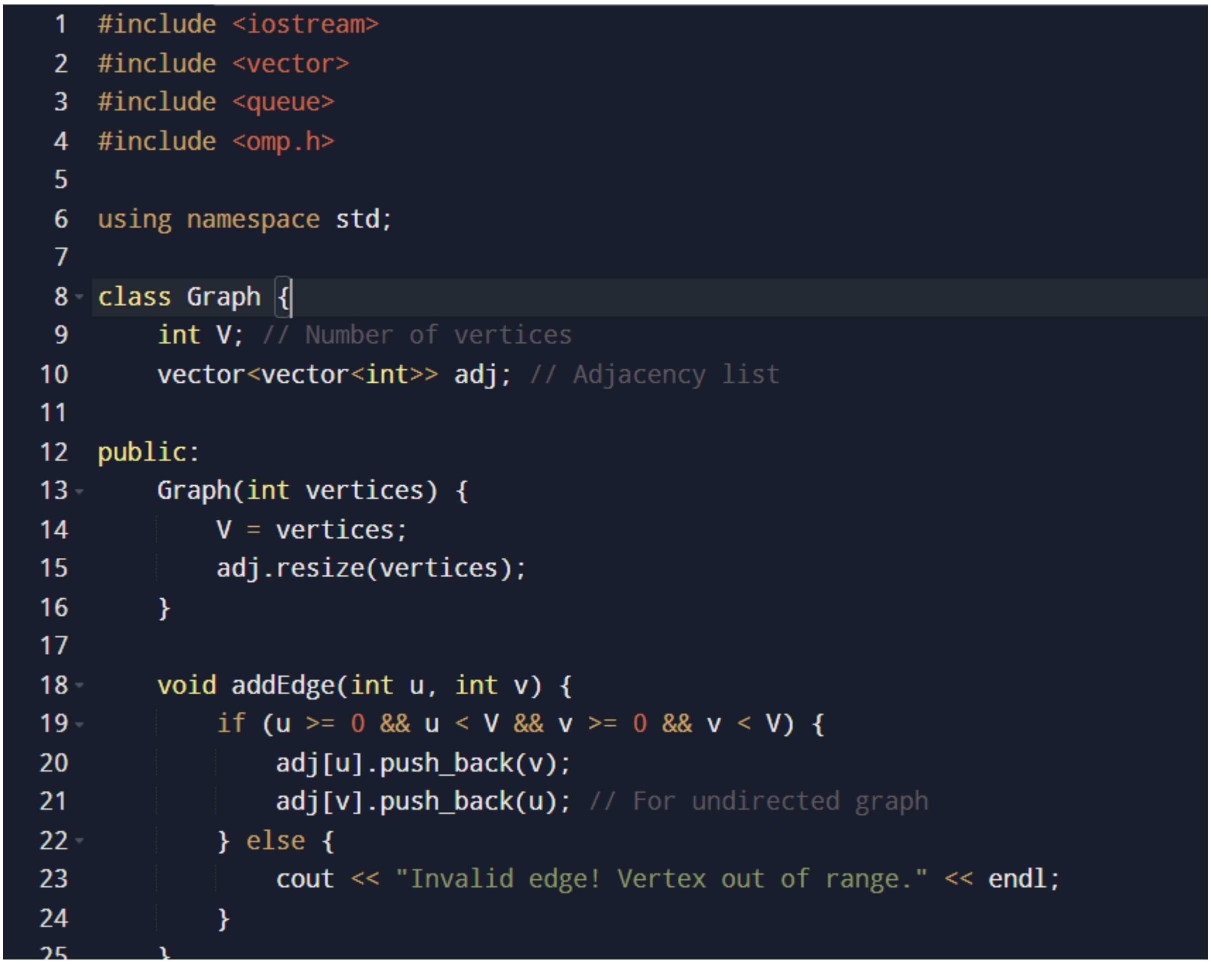


```cpp
    // Step 5: Merge sorted bins back to the original array
    int index = 0;
    for (int i = 0; i < NUM_BINS; i++) {
        for (int val : bins[i]) {
            arr[index++] = val;
        }
    }
}

int main() {
    std::vector<int> arr = {23, 45, 12, 89, 5, 34, 78, 11, 90, 67, 55, 32,
        43, 21};

    std::cout << "Original array: ";
    for (int num : arr) std::cout << num << " ";
    std::cout << "\n";

    histogramSort(arr);

    std::cout << "Sorted array: ";
    for (int num : arr) std::cout << num << " ";
    std::cout << "\n";

    return 0;
}
```

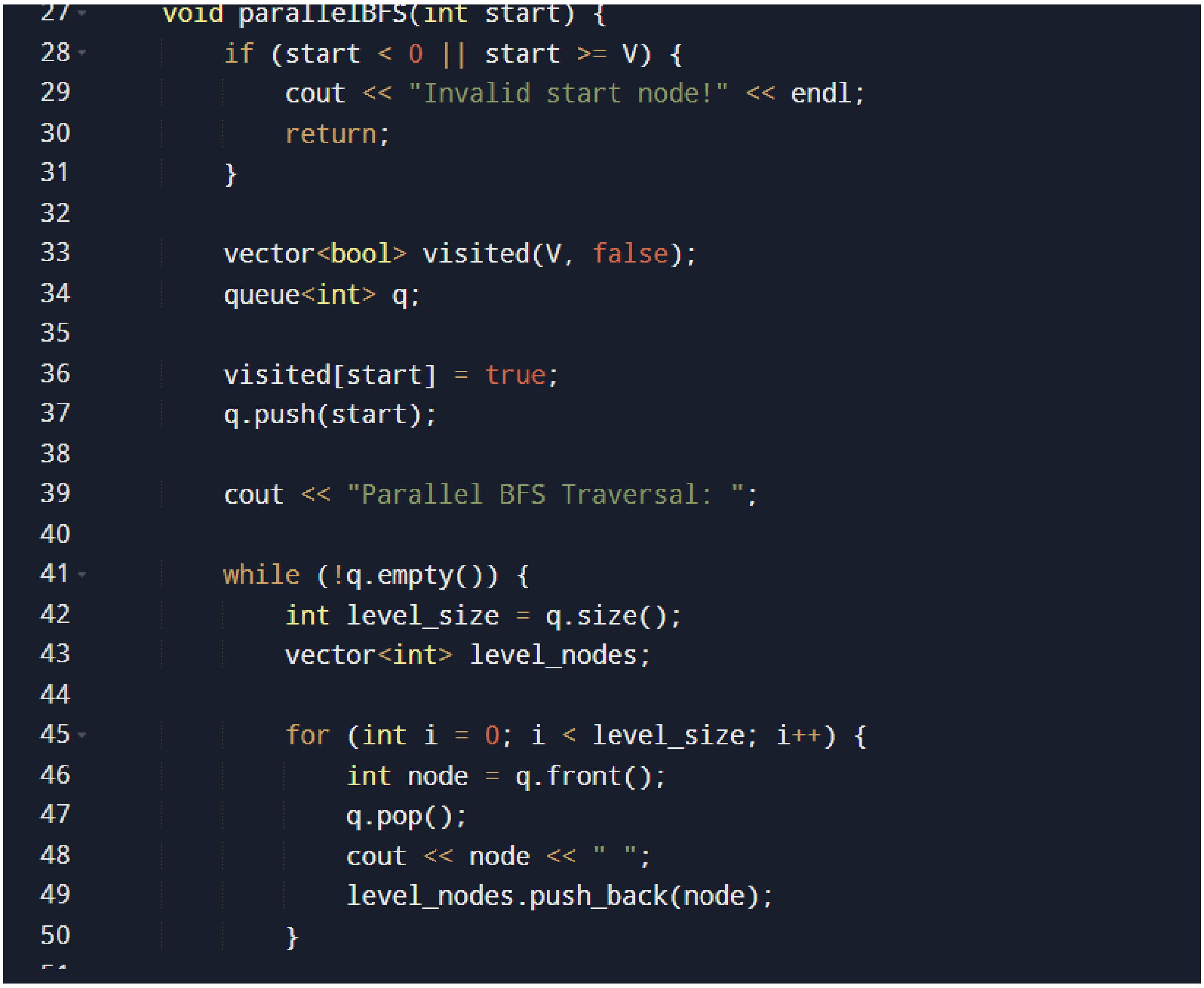


```
Original array: 23 45 12 89 5 34 78 11 90 67 55 32 43 21
Sorted array: 5 11 12 21 23 32 34 43 45 55 67 78 89 90


=== Code Execution Successful ===
```

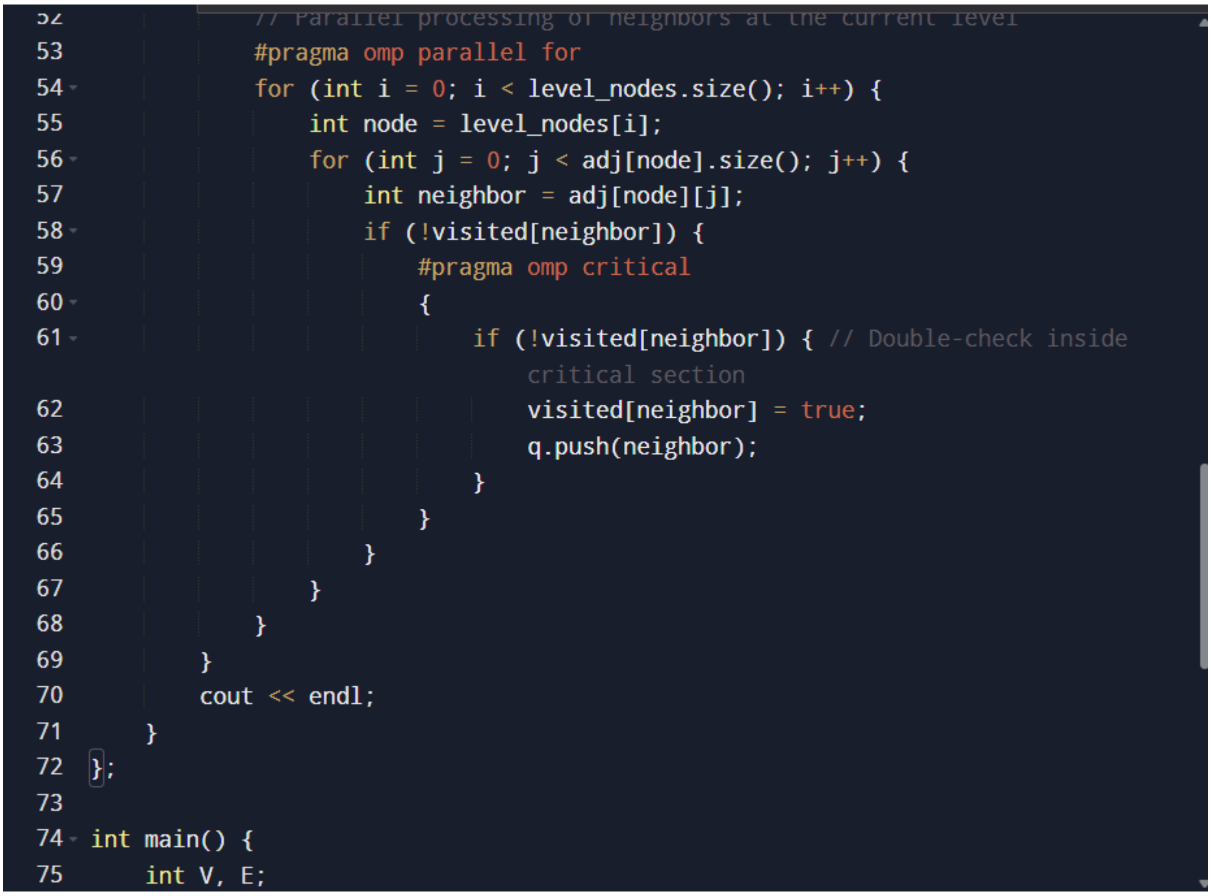
3. Implement breadth first search in parallel using OpenMP



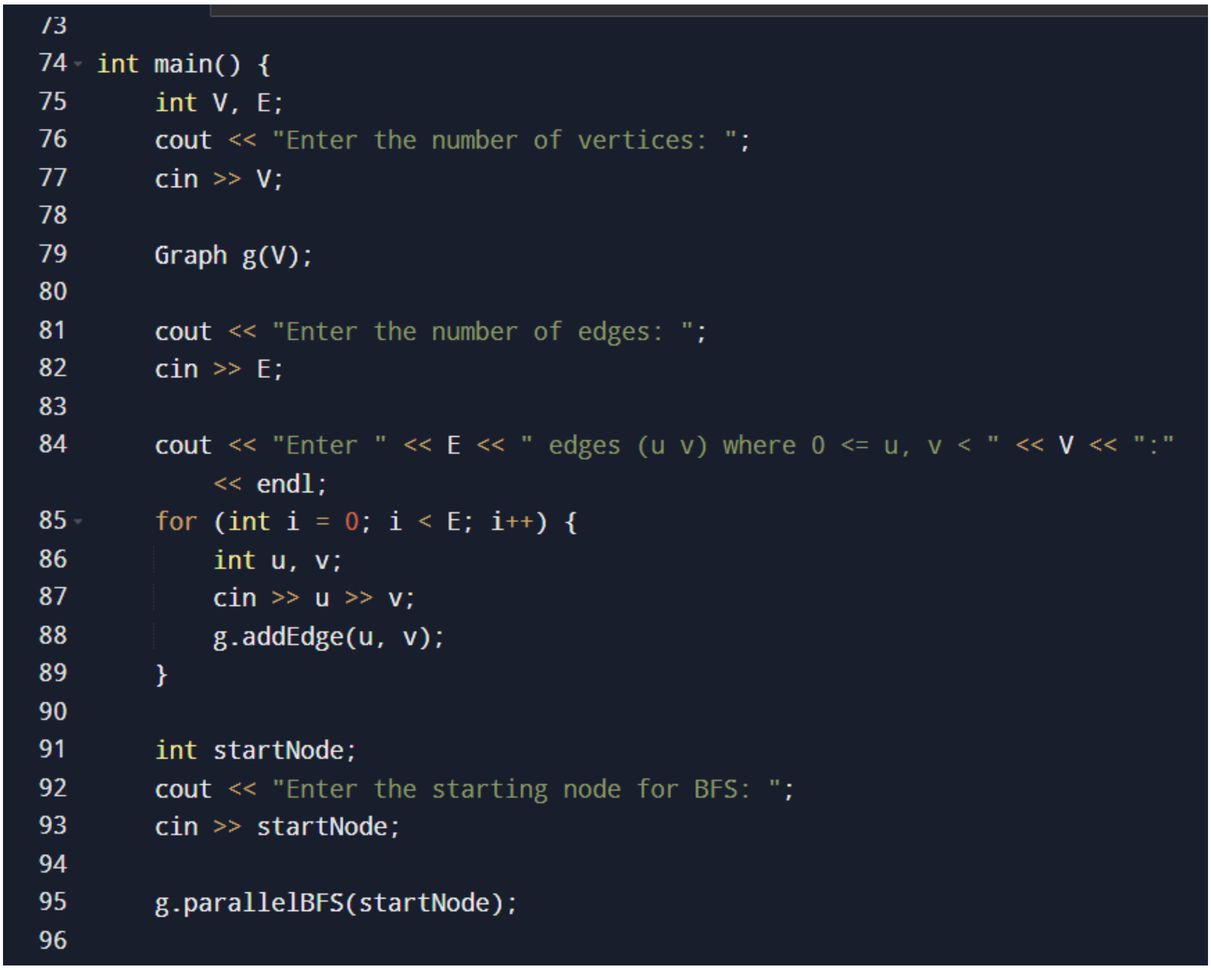
```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

class Graph {
    int V; // Number of vertices
    vector<vector<int>> adj; // Adjacency list

public:
    Graph(int vertices) {
        V = vertices;
        adj.resize(vertices);
    }

    void addEdge(int u, int v) {
        if (u >= 0 && u < V && v >= 0 && v < V) {
            adj[u].push_back(v);
            adj[v].push_back(u); // For undirected graph
        } else {
            cout << "Invalid edge! Vertex out of range." << endl;
        }
    }
```



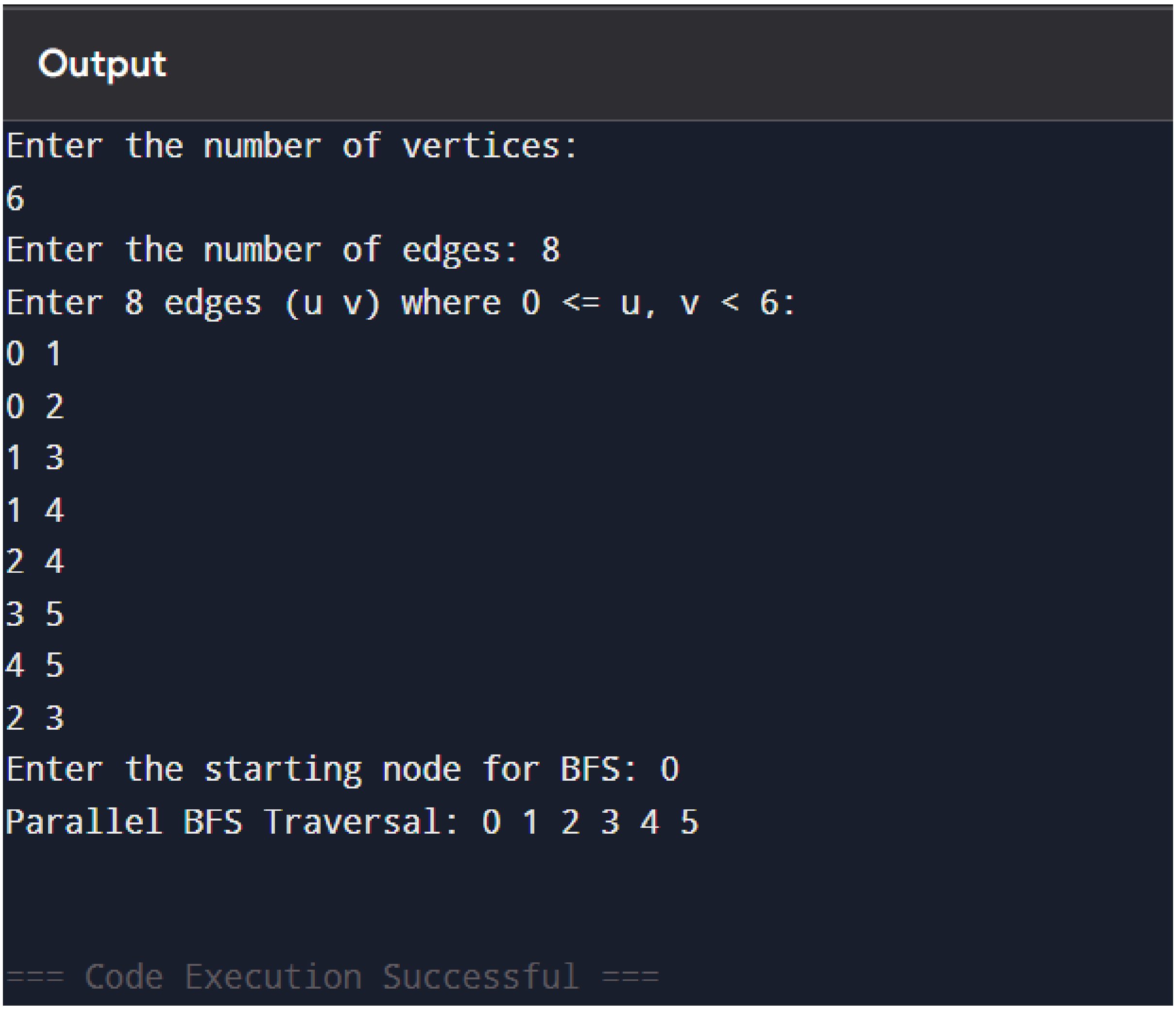
```cpp
    void parallelBFS(int start) {
        if (start < 0 || start >= V) {
            cout << "Invalid start node!" << endl;
            return;
        }

        vector<bool> visited(V, false);
        queue<int> q;

        visited[start] = true;
        q.push(start);

        cout << "Parallel BFS Traversal: ";

        while (!q.empty()) {
            int level_size = q.size();
            vector<int> level_nodes;

            for (int i = 0; i < level_size; i++) {
                int node = q.front();
                q.pop();
                cout << node << " ";
                level_nodes.push_back(node);
            }
```



```cpp
            // Parallel processing of neighbors at the current level
            #pragma omp parallel for
            for (int i = 0; i < level_nodes.size(); i++) {
                int node = level_nodes[i];
                for (int j = 0; j < adj[node].size(); j++) {
                    int neighbor = adj[node][j];
                    if (!visited[neighbor]) {
                        #pragma omp critical
                        {
                            if (!visited[neighbor]) { // Double-check inside critical section
                                visited[neighbor] = true;
                                q.push(neighbor);
                            }
                        }
                    }
                }
            }
        }
        cout << endl;
    }
};

int main() {
    int V, E;
```



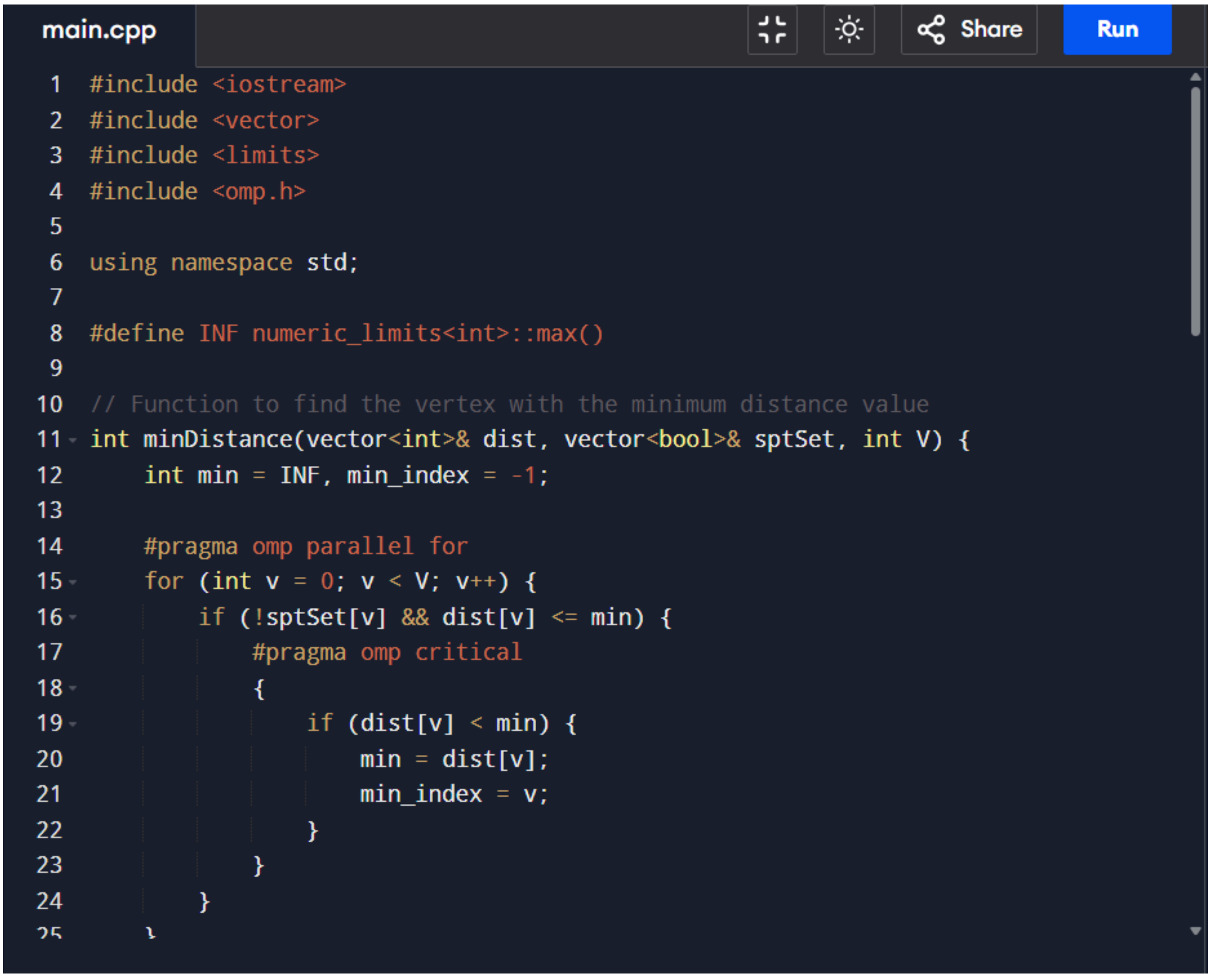
```cpp
int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
    cin >> V;

    Graph g(V);

    cout << "Enter the number of edges: ";
    cin >> E;

    cout << "Enter " << E << " edges (u v) where 0 <= u, v < " << V << ":" << endl;
    for (int i = 0; i < E; i++) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    int startNode;
    cout << "Enter the starting node for BFS: ";
    cin >> startNode;

    g.parallelBFS(startNode);

```



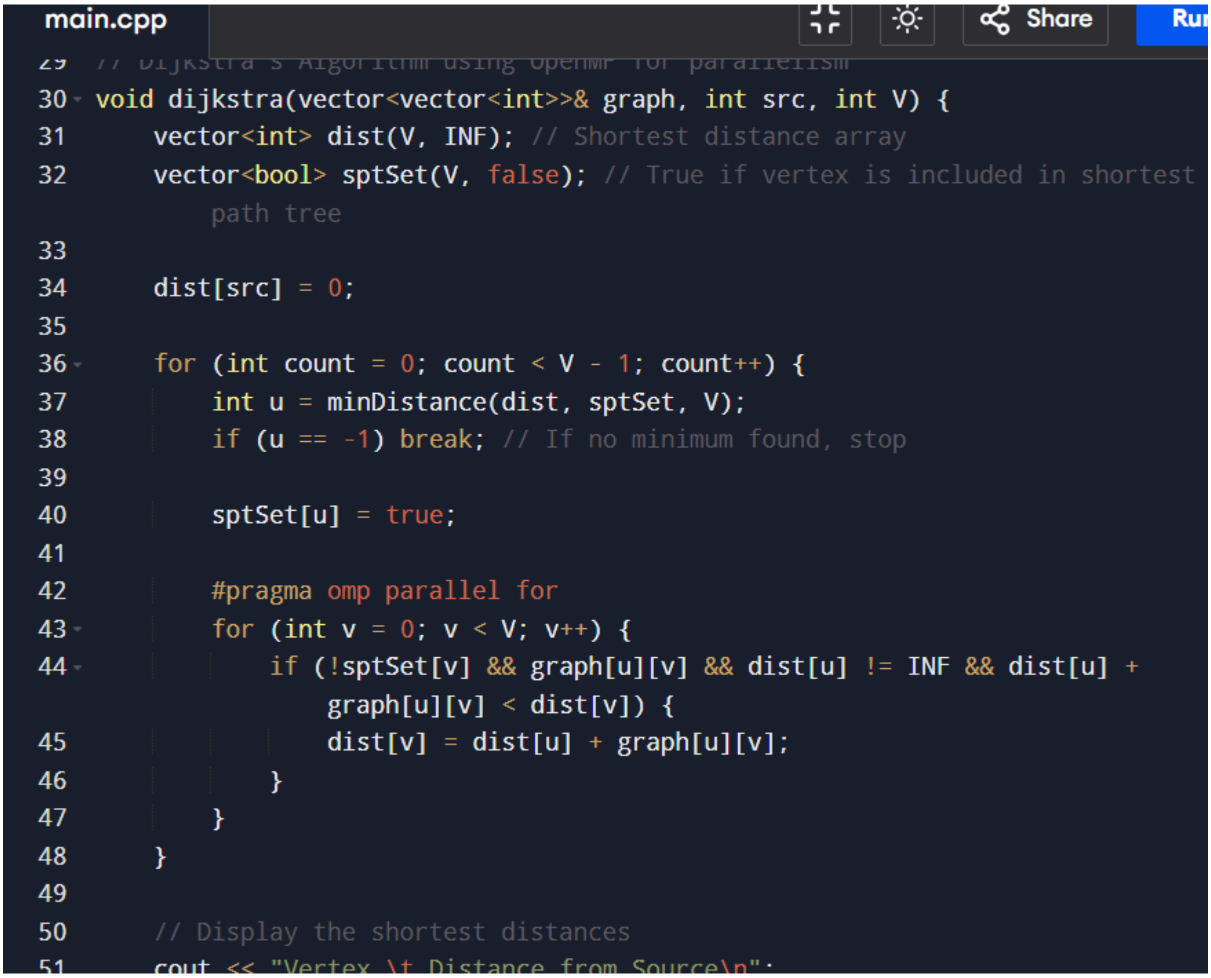
Output

```
Enter the number of vertices:
6
Enter the number of edges: 8
Enter 8 edges (u v) where 0 <= u, v < 6:
0 1
0 2
1 3
1 4
2 4
3 5
4 5
2 3
Enter the starting node for BFS: 0
Parallel BFS Traversal: 0 1 2 3 4 5


=== Code Execution Successful ===
```

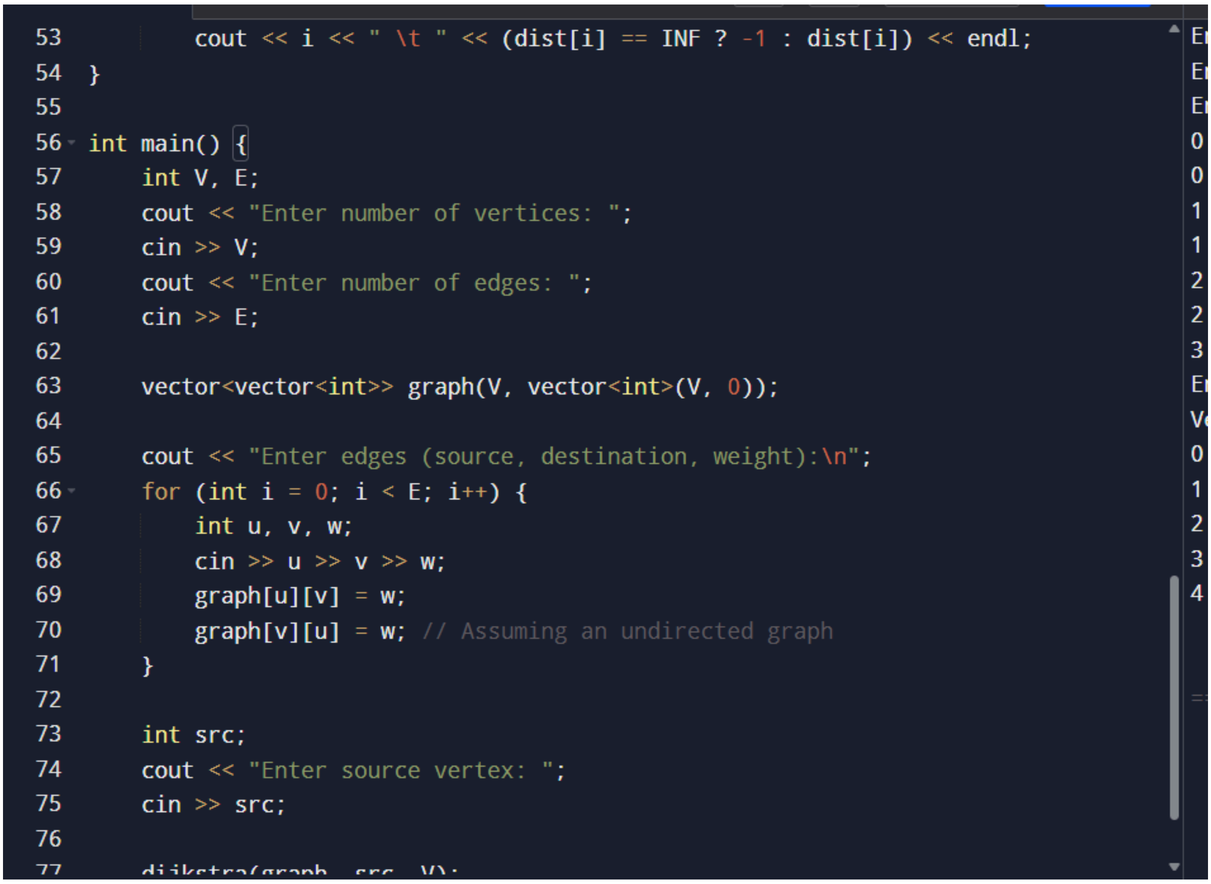
4. Implement Dijkstra’s algorithm in parallel using OpenMP



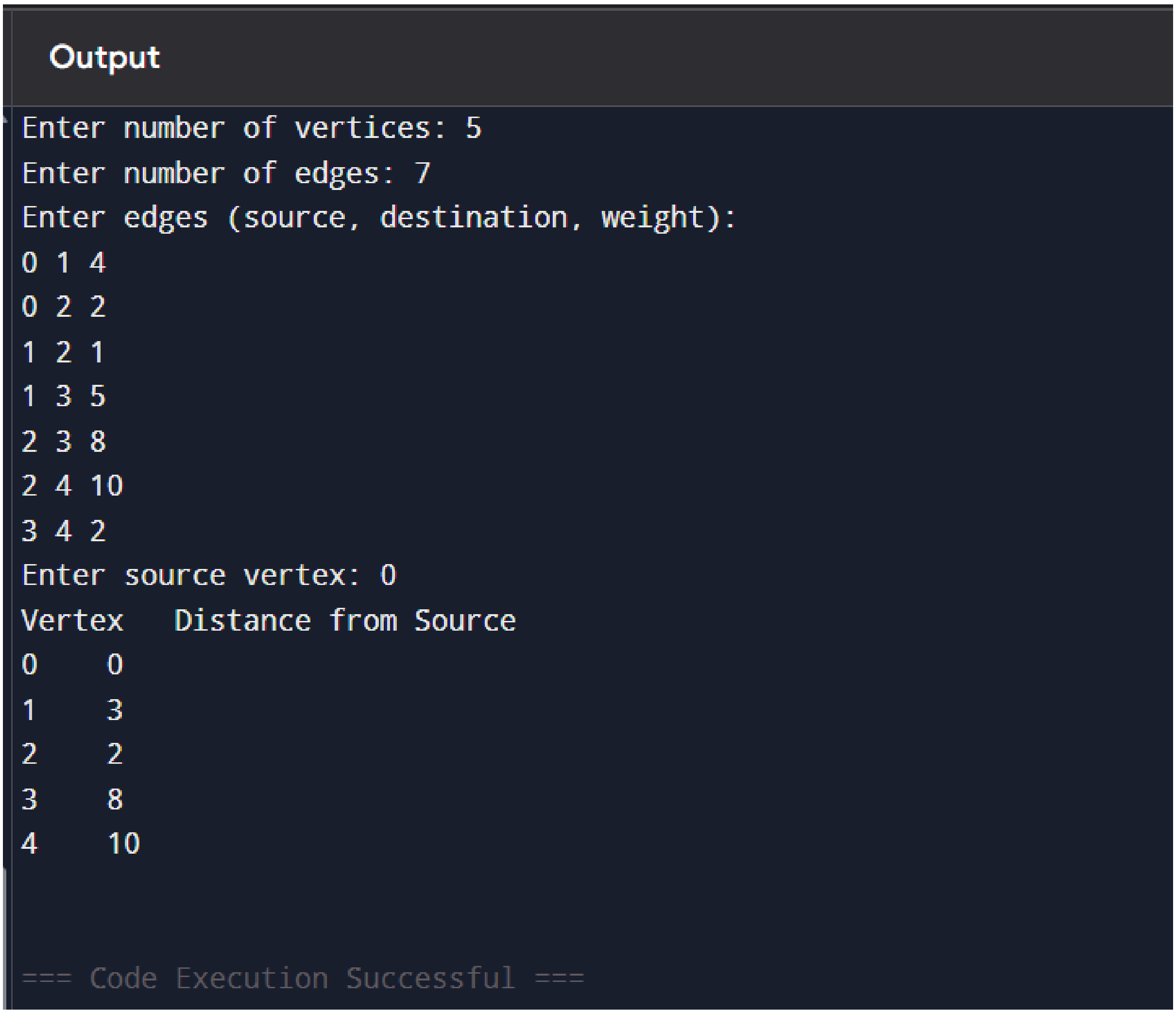
```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <omp.h>

using namespace std;

#define INF numeric_limits<int>::max()

// Function to find the vertex with the minimum distance value
int minDistance(vector<int>& dist, vector<bool>& sptSet, int V) {
    int min = INF, min_index = -1;

    #pragma omp parallel for
    for (int v = 0; v < V; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            #pragma omp critical
            {
                if (dist[v] < min) {
                    min = dist[v];
                    min_index = v;
                }
            }
        }
    }
```



```cpp
// Dijkstra's Algorithm using OpenMP for parallelism
void dijkstra(vector<vector<int>>& graph, int src, int V) {
    vector<int> dist(V, INF); // Shortest distance array
    vector<bool> sptSet(V, false); // True if vertex is included in shortest path tree

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet, V);
        if (u == -1) break; // If no minimum found, stop

        sptSet[u] = true;

        #pragma omp parallel for
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +
                graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Display the shortest distances
    cout << "Vertex \t Distance from Source\n";
```



```
        cout << i << " \t " << (dist[i] == INF ? -1 : dist[i]) << endl;
}

int main() {
    int V, E;
    cout << "Enter number of vertices: ";
    cin >> V;
    cout << "Enter number of edges: ";
    cin >> E;

    vector<vector<int>> graph(V, vector<int>(V, 0));

    cout << "Enter edges (source, destination, weight):\n";
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u][v] = w;
        graph[v][u] = w; // Assuming an undirected graph
    }

    int src;
    cout << "Enter source vertex: ";
    cin >> src;

    dijkstra(graph, src, V);
```



```
Enter number of vertices: 5
Enter number of edges: 7
Enter edges (source, destination, weight):
0 1 4
0 2 2
1 2 1
1 3 5
2 3 8
2 4 10
3 4 2
Enter source vertex: 0
Vertex   Distance from Source
0    0
1    3
2    2
3    8
4    10


=== Code Execution Successful ===
```