**BFS & DFS**

```cpp
#include <iostream>       // For input/output operations
#include <vector>         // For using vector container
#include <queue>          // For BFS queue implementation
#include <omp.h>          // OpenMP library for parallel programming
#include <stack>          // For DFS stack implementation
// Graph class representing an undirected graph using adjacency lists
class Graph {
    int V;  // Number of vertices in the graph
    std::vector<std::vector<int>> adj;  // Adjacency list representation
public:
    // Constructor to initialize graph with V vertices
    Graph(int v) : V(v), adj(v) {}  // Initialize V and resize adjacency list to v
    // Method to add an edge between vertices v and w
    void addEdge(int v, int w) { adj[v].push_back(w); }  // Add w to v's adjacency list
    // Parallel Breadth-First Search implementation
    void parallelBFS(int start) {
        std::vector<bool> visited(V, false);  // Track visited vertices
        std::queue<int> q;                    // Queue for BFS traversal
        visited[start] = true;  // Mark start vertex as visited
        q.push(start);          // Enqueue start vertex
        while (!q.empty()) {
            #pragma omp parallel  // Start parallel region
            {
                #pragma omp for nowait  // Parallelize loop with no implicit barrier
                for (int i = 0; i < q.size(); i++) {
                    int v;
                    #pragma omp critical  // Critical section to prevent race condition
                    {
                        v = q.front();  // Get front element
                        q.pop();       // Remove front element
                    }
                    std::cout << v << " ";  // Process current vertex

                    // Visit all adjacent vertices
                    for (int u : adj[v]) {
                        #pragma omp critical  // Critical section for shared variables
                        if (!visited[u]) {
                            visited[u] = true;  // Mark as visited
                            q.push(u);         // Enqueue adjacent vertex
                        }
                    }
                }
            }
        }
        std::cout << "\n";  // Newline after traversal
    }
    // Parallel Depth-First Search implementation
    void parallelDFS(int start) {
        std::vector<bool> visited(V, false);  // Track visited vertices

        #pragma omp parallel  // Start parallel region
        {
            std::stack<int> s;  // Each thread gets its own stack

            #pragma omp single  // Single thread executes this block
            s.push(start);      // Push start vertex to stack

            while (!s.empty()) {
                int v = -1;  // Initialize with invalid vertex

                #pragma omp critical  // Critical section for stack operations
                if (!s.empty()) {
                    v = s.top();  // Get top element
                    s.pop();     // Remove top element
                }

                if (v == -1) continue;  // Skip if no vertex was popped

                if (!visited[v]) {
                    #pragma omp critical  // Critical section for shared variables
                    if (!visited[v]) {    // Double-check pattern to prevent race
                        visited[v] = true;      // Mark as visited
                        std::cout << v << " "; // Process current vertex

                        // Push adjacent vertices in reverse order (for DFS)
                        for (auto it = adj[v].rbegin(); it != adj[v].rend(); ++it)
                            s.push(*it);
                    }
                }
            }
        }
        std::cout << "\n";  // Newline after traversal
    }
};

int main() {
    // Create a graph with 4 vertices
    Graph g(4);

    // Add edges to the graph
    g.addEdge(0, 1);  // Edge 0->1
    g.addEdge(0, 2);  // Edge 0->2
    g.addEdge(1, 3);  // Edge 1->3
    g.addEdge(2, 3);  // Edge 2->3

    // Perform and print BFS traversal starting from vertex 0
    std::cout << "BFS: ";
    g.parallelBFS(0);

    // Perform and print DFS traversal starting from vertex 0
    std::cout << "DFS: ";
    g.parallelDFS(0);

    return 0;  // End of program
}
```

**Bubble & Merge Sort**

```cpp
#include <iostream>    // Standard input/output operations
#include <vector>      // Dynamic array container
#include <algorithm>   // For swap function
#include <omp.h>       // OpenMP library for parallel processing

// Parallel Bubble Sort using Odd-Even Transposition algorithm
void parallelBubbleSort(std::vector<int>& arr) {
    int n = arr.size();       // Get array size
    bool swapped;             // Flag to track if swaps occurred

    // Outer loop for each sorting pass
    for (int i = 0; i < n; ++i) {
        swapped = false;      // Reset swap flag for new pass

        // Parallel inner loop - compares adjacent elements
        #pragma omp parallel for shared(arr, swapped)
        // Odd-even approach: alternates starting points (0 or 1)
        for (int j = i % 2; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {      // Compare neighbors
                std::swap(arr[j], arr[j + 1]); // Swap if out of order
                #pragma omp atomic write      // Thread-safe flag update
                swapped = true;               // Mark swap occurred
            }
        }

        // Early termination if no swaps in pass (array is sorted)
        if (!swapped) break;
    }
}

// Sequential Merge helper function for merge sort
void merge(std::vector<int>& arr, int l, int m, int r) {
    // Create temp array containing elements to merge
    std::vector<int> temp(arr.begin() + l, arr.begin() + r + 1);

    // Initialize pointers:
    int i = 0;              // Left subarray (starts at temp[0])
    int j = m - l + 1;      // Right subarray (starts at temp[m-l+1])
    int k = l;              // Position in original array

    // Merge while both subarrays have elements
    while (i <= m - l && j <= r - l) {
        // Select smaller element from either subarray
        arr[k++] = (temp[i] <= temp[j]) ? temp[i++] : temp[j++];
    }

    // Copy remaining left subarray elements if any
    while (i <= m - l) arr[k++] = temp[i++];
}

// Parallel Merge Sort using divide-and-conquer
void parallelMergeSort(std::vector<int>& arr, int l, int r) {
    if (l < r) {  // Base case: more than one element
        int m = l + (r - l) / 2;  // Calculate midpoint

        // Parallel recursive sorting:
        #pragma omp parallel sections  // Split into parallel sections
        {
            #pragma omp section        // First thread sorts left half
            parallelMergeSort(arr, l, m);

            #pragma omp section        // Second thread sorts right half
            parallelMergeSort(arr, m + 1, r);
        }

        merge(arr, l, m, r);  // Merge the sorted halves
    }
}

int main() {
    // Initialize test data
    std::vector<int> arr = {5, 2, 9, 1, 5, 6};
    std::vector<int> arr_copy = arr;  // Duplicate for merge sort

    // Parallel Bubble Sort demo
    std::cout << "Before Bubble Sort: ";
    for (int num : arr) std::cout << num << " ";  // Print original

    parallelBubbleSort(arr);  // Perform parallel bubble sort

    std::cout << "\nAfter Bubble Sort:  ";
    for (int num : arr) std::cout << num << " ";  // Print sorted

    // Parallel Merge Sort demo
    std::cout << "\n\nBefore Merge Sort:  ";
    for (int num : arr_copy) std::cout << num << " ";  // Print original

    parallelMergeSort(arr_copy, 0, arr_copy.size() - 1);  // Perform merge sort

    std::cout << "\nAfter Merge Sort:   ";
    for (int num : arr_copy) std::cout << num << " ";  // Print sorted

    return 0;  // Exit program
}
```

**MinMax**

```
#include <iostream>       // For input/output operations
#include <vector>         // For using vector container
#include <queue>          // For BFS queue implementation
#include <omp.h>          // OpenMP library for parallel programming
#include <stack>          // For DFS stack implementation
// Graph class representing an undirected graph using adjacency lists
class Graph {
    int V;  // Number of vertices in the graph
    std::vector<std::vector<int>> adj;  // Adjacency list representation
public:
    // Constructor to initialize graph with V vertices
    Graph(int v) : V(v), adj(v) {}  // Initialize V and resize adjacency list to v
    // Method to add an edge between vertices v and w
    void addEdge(int v, int w) { adj[v].push_back(w); }  // Add w to v's adjacency list

    // Parallel Breadth-First Search implementation
    void parallelBFS(int start) {
        std::vector<bool> visited(V, false);  // Track visited vertices
        std::queue<int> q;                    // Queue for BFS traversal
        visited[start] = true;  // Mark start vertex as visited
        q.push(start);          // Enqueue start vertex

        while (!q.empty()) {
            #pragma omp parallel  // Start parallel region
            {
                #pragma omp for nowait  // Parallelize loop with no implicit barrier
                for (int i = 0; i < q.size(); i++) {
                    int v;
                    #pragma omp critical  // Critical section to prevent race condition
                    {
                        v = q.front();  // Get front element
                        q.pop();       // Remove front element
                    }
                    std::cout << v << " ";  // Process current vertex

                    // Visit all adjacent vertices
                    for (int u : adj[v]) {
                        #pragma omp critical  // Critical section for shared variables
                        if (!visited[u]) {
                            visited[u] = true;  // Mark as visited
                            q.push(u);         // Enqueue adjacent vertex
                        }
                    }
                }
            }
        }
        std::cout << "\n";  // Newline after traversal
    }

    // Parallel Depth-First Search implementation
    void parallelDFS(int start) {
        std::vector<bool> visited(V, false);  // Track visited vertices

        #pragma omp parallel  // Start parallel region
        {
            std::stack<int> s;  // Each thread gets its own stack

            #pragma omp single  // Single thread executes this block
            s.push(start);      // Push start vertex to stack

            while (!s.empty()) {
                int v = -1;  // Initialize with invalid vertex

                #pragma omp critical  // Critical section for stack operations
                if (!s.empty()) {
                    v = s.top();  // Get top element
                    s.pop();     // Remove top element
                }
                if (v == -1) continue;  // Skip if no vertex was popped

                if (!visited[v]) {
                    #pragma omp critical  // Critical section for shared variables
                    if (!visited[v]) {    // Double-check pattern to prevent race
                        visited[v] = true;      // Mark as visited
                        std::cout << v << " "; // Process current vertex
                        // Push adjacent vertices in reverse order (for DFS)
                        for (auto it = adj[v].rbegin(); it != adj[v].rend(); ++it)
                            s.push(*it);
                    }
                }
            }
        }
        std::cout << "\n";  // Newline after traversal
    }
};
int main() {
    // Create a graph with 4 vertices
    Graph g(4);

    // Add edges to the graph
    g.addEdge(0, 1);  // Edge 0->1
    g.addEdge(0, 2);  // Edge 0->2
    g.addEdge(1, 3);  // Edge 1->3
    g.addEdge(2, 3);  // Edge 2->3

    // Perform and print BFS traversal starting from vertex 0
    std::cout << "BFS: ";
    g.parallelBFS(0);

    // Perform and print DFS traversal starting from vertex 0
    std::cout << "DFS: ";
    g.parallelDFS(0);

    return 0;  // End of program
}
```