Practical NO 1 BFS:

```
#include <iostream>
#include <queue>
#include <vector>
#include <omp.h>
using namespace std;
// Structure to represent a graph edge
struct Edge {
   int src, dest;
};
// Structure to represent a graph
class Graph {
public:
   vector<vector<int>> adjList;
   // Constructor
   Graph(vector<Edge> const &edges, int N) {
      adjList.resize(N);

   for (auto &edge : edges) {
         adjList[edge.src].push_back(edge.dest);
      }
   }
};
// Parallel breadth-first search
void parallelBFS(Graph const &graph, int source) {
   int numVertices = graph.adjList.size();
   vector<bool> visited(numVertices, false);
   queue<int> q;
// Start BFS from the source node
   q.push(source);
   visited[source] = true;
while (!q.empty()) {
      #pragma omp parallel
```

```
        {
            #pragma omp for
            for (int i = 0; i < q.size(); ++i) {
                int u = q.front();
                q.pop();

                cout << u << " ";

                // Visit all the adjacent vertices of u
                for (int v : graph.adjList[u]) {
                    if (!visited[v]) {
                        visited[v] = true;
                        q.push(v);
                    }
                }
            }
        }
    }
}

int main() {
    // Example graph
    vector<Edge> edges = {{0, 1}, {0, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}};
    int numVertices = 7;
    Graph graph(edges, numVertices);

    cout << "Parallel BFS traversal starting from vertex 0: ";
    parallelBFS(graph, 0);
    cout << endl;

    return 0;
}
```

Ouput :Parallel BFS traversal starting from vertex 0: 0 1 2 3 4 5 6

Practical NO 1 .DFS:

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

using namespace std;

// Structure to represent a graph edge
struct Edge {
    int src, dest;
};

// Structure to represent a graph
class Graph {
public:
    vector<vector<int>> adjList;

    // Constructor
    Graph(vector<Edge> const &edges, int N) {
        adjList.resize(N);

        for (auto &edge : edges) {
            adjList[edge.src].push_back(edge.dest);
        }
    }
};

// Depth-first search
void DFSUtil(const Graph& graph, int v, vector<bool>& visited) {
    visited[v] = true;
    cout << v << " ";
```

```
    // Traverse all adjacent vertices
    #pragma omp parallel for
    for (int i = 0; i < graph.adjList[v].size(); ++i) {
        int u = graph.adjList[v][i];
        if (!visited[u])
            DFSUtil(graph, u, visited);
    }
}

// Parallel depth-first search
void parallelDFS(const Graph& graph) {
    int V = graph.adjList.size();
    vector<bool> visited(V, false);

    // Traverse all vertices
    #pragma omp parallel for
    for (int v = 0; v < V; ++v) {
        if (!visited[v])
            DFSUtil(graph, v, visited);
    }
}

int main() {
    // Example graph
    vector<Edge> edges = {{0, 1}, {0, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}};
    int numVertices = 7;
    Graph graph(edges, numVertices);

    cout << "Parallel DFS traversal: ";
    parallelDFS(graph);
    cout << endl;
return 0;   }
```

Ouput: Parallel DFS traversal: 0 1 3 4 2 5 6

Practical 2 parallel bubble sort:

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

using namespace std;

// Parallel bubble sort function
void parallelBubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped = true;

    #pragma omp parallel
    {
        while (swapped) {
            swapped = false;

            #pragma omp for
            for (int i = 0; i < n - 1; ++i) {
                if (arr[i] > arr[i + 1]) {
                    swap(arr[i], arr[i + 1]);
                    swapped = true;
                }
            }
        }
    }
}

int main() {
    // Example array
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    cout << "Original array: ";
```

```
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Perform parallel bubble sort
    parallelBubbleSort(arr);

    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Ouput: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

Merge Sort :

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

using namespace std;

// Merge function to merge two sorted subarrays
void merge(vector<int>& arr, int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;

    // Create temporary arrays
    vector<int> L(n1), R(n2);
```

```
    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[low + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[low..high]
    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
// Merge sort function
void mergeSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;

        // Parallelize the sorting of two halves
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSort(arr, low, mid);

            #pragma omp section
            mergeSort(arr, mid + 1, high);
        }

        // Merge the sorted halves
        merge(arr, low, mid, high);
    }
}

// Parallel merge sort function
void parallelMergeSort(vector<int>& arr) {
    mergeSort(arr, 0, arr.size() - 1);
}

int main() {
    // Example array
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
```

```
    }
    cout << endl;

    // Perform parallel merge sort
    parallelMergeSort(arr);

    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Ouput: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

Practical No 3 Min Max

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

using namespace std;

int main() {
    vector<int> arr = {5, 8, 3, 2, 9, 4, 6, 1, 7};

    // Size of the array
    int n = arr.size();

    int min_val = arr[0];
    int max_val = arr[0];
    int sum = 0;
```

```
    // Find minimum, maximum, and sum using parallel reduction
    #pragma omp parallel for reduction(min:min_val) reduction(max:max_val) reduction(+:sum)
    for (int i = 0; i < n; i++) {
        min_val = min(min_val, arr[i]);
        max_val = max(max_val, arr[i]);
        sum += arr[i];
    }

    double average = static_cast<double>(sum) / n;

    cout << "Minimum: " << min_val << endl;
    cout << "Maximum: " << max_val << endl;
    cout << "Sum: " << sum << endl;
    cout << "Average: " << average << endl;

    return 0;
}
```

Ouput: Minimum: 1

Maximum: 9

Sum: 45

Average: 5

Mini Project code Parallel Quick Sort:

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

using namespace std;

// Partition function for quicksort
int partition(vector<int>& arr, int low, int high) {
```

```
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Quicksort function
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        #pragma omp parallel sections
        {
            #pragma omp section
            quickSort(arr, low, pi - 1);

            #pragma omp section
            quickSort(arr, pi + 1, high);
        }
    }
}

// Parallel quicksort function
void parallelQuickSort(vector<int>& arr) {
    int n = arr.size();
    #pragma omp parallel
    {
```

```
        #pragma omp single nowait
        quickSort(arr, 0, n - 1);
    }
}

int main() {
    // Example array
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    // Perform parallel quicksort
    parallelQuickSort(arr);

    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Output: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90