

HPC 1:-

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

```
using namespace std;
```

```
class Graph {
    int V;
    vector<vector<int>> adj;
```

```
public:
```

```
    Graph(int V) : V(V), adj(V) {}
```

```
    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
```

```
    void bfs(int start) {
        vector<bool> visited(V, false);
        queue<int> q;
        q.push(start);
        visited[start] = true;
```

```
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            cout << u << " ";
            #pragma omp parallel for
            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    q.push(v);
                }
            }
        }
        cout << endl;
    }
```

```
    void dfs(int start) {
```

```

vector<bool> visited(V, false);
stack<int> stk;
stk.push(start);

while (!stk.empty()) {
    int u = stk.top();
    stk.pop();
    if (!visited[u]) {
        cout << u << " ";
        visited[u] = true;
        #pragma omp parallel for
        for (int v : adj[u]) {
            if (!visited[v]) {
                stk.push(v);
            }
        }
    }
}
cout << endl;
};

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

    Graph g(V);

    cout << "Enter the edges (in format 'u v'):\n";
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        g.addEdge(u, v);
    }

    cout << "BFS starting from vertex 0: ";
    g.bfs(0);

    cout << "DFS starting from vertex 0: ";
    g.dfs(0);


    return 0;
}

```

This code defines a `Graph` class with methods for adding edges, performing BFS, and DFS. OpenMP directives (`#pragma omp parallel for`) are used to parallelize the loops in BFS and DFS algorithms.

To compile and run the program, you'll need to have OpenMP support enabled in your compiler. For example, you can compile the program using g++ with OpenMP support:


bash

 Copy code

```
g++ -fopenmp graph.cpp -o graph
```

Then run the executable:

bash

 Copy code

```
./graph
```



HPC 2:-

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

```
using namespace std;
using namespace std::chrono;
```

```
// Function to perform Bubble Sort
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

```
// Function to perform Merge Sort
void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
}
```

```

while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

void mergeSortHelper(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSortHelper(arr, l, m);
        mergeSortHelper(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Function to perform Merge Sort (Parallel)
void mergeSort(vector<int>& arr) {
    mergeSortHelper(arr, 0, arr.size() - 1);
}

// Function to print the elements of an array
void printArray(const vector<int>& arr) {
    for (int x : arr) {
        cout << x << " ";
    }
    cout << endl;
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

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

```

```

}

// Measure time for sequential Bubble Sort
auto start = high_resolution_clock::now();
bubbleSort(arr);
auto stop = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(stop - start);
cout << "Sequential Bubble Sort took " << duration.count() << " milliseconds\n";

// Measure time for sequential Merge Sort
start = high_resolution_clock::now();
mergeSort(arr);
stop = high_resolution_clock::now();
duration = duration_cast<milliseconds>(stop - start);
cout << "Sequential Merge Sort took " << duration.count() << " milliseconds\n";

// Reset array for parallel sorting
cout << "\nResetting array for parallel sorting...\n";
cout << "Enter the elements of the array:\n";
for (int i = 0; i < n; ++i) {
    cin >> arr[i];
}

// Measure time for parallel Bubble Sort
start = high_resolution_clock::now();
#pragma omp parallel
{
    bubbleSort(arr);
}
stop = high_resolution_clock::now();
duration = duration_cast<milliseconds>(stop - start);
cout << "Parallel Bubble Sort took " << duration.count() << " milliseconds\n";

// Reset array for parallel sorting
cout << "\nResetting array for parallel sorting...\n";
cout << "Enter the elements of the array:\n";
for (int i = 0; i < n; ++i) {
    cin >> arr[i];
}

// Measure time for parallel Merge Sort
start = high_resolution_clock::now();
#pragma omp parallel
{

```

```
        mergeSort(arr);
    }
    stop = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(stop - start);
    cout << "Parallel Merge Sort took " << duration.count() << " milliseconds\n";

    return 0;
}
```

HPC 3:-

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

```
using namespace std;
```

```
int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

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

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

    #pragma omp parallel for reduction(min:min_val) reduction(max:max_val) reduction(+:sum)
    for (int i = 0; i < n; ++i) {
        if (arr[i] < min_val) min_val = arr[i];
        if (arr[i] > max_val) max_val = arr[i];
        sum += arr[i];
    }

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

    cout << "Min value: " << min_val << endl;
    cout << "Max value: " << max_val << endl;
    cout << "Sum: " << sum << endl;
    cout << "Average: " << average << endl;

    return 0;
}
```