Bfs dfs :

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

using namespace std;

const int N = 1e5 + 5;
vector<int> g[N];
bool vis[N];

void bfs(int s) {
    queue<int> q;
    q.push(s);
    vis[s] = true;

    while (!q.empty()) {
        int u = q.front();
        cout << u << " ";
        q.pop();

        #pragma omp parallel for shared(q) schedule(dynamic)
        for (int i = 0; i < g[u].size(); i++) {
            int v = g[u][i];
            if (!vis[v]) {
                #pragma omp critical
                {
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
}

void dfs(int u) {
    vis[u] = true;
    cout<< u << " ";

    #pragma omp parallel for shared(vis) schedule(dynamic)
    for (int i = 0; i < g[u].size(); i++) {
        int v = g[u][i];
        if (!vis[v]) {
            #pragma omp critical
            {
                dfs(v);
            }
```

```
        }
    }
}

int main() {
    int n, m, s, choice;
    cin >> n >> m >> s >> choice;

    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
        g[x].push_back(y);
        g[y].push_back(x);
    }

    if (choice == 1) {
        cout << "BFS : ";
        bfs(s);
    } else if (choice == 2) {
        cout << "DFS : ";
        dfs(s);
    } else {
        cout << "Invalid choice\n";
    }

    return 0;
}
```

Bubblemerge

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>

using namespace std;

// Parallel Bubble Sort function
void parallelBubbleSort(int *array, int n) {
    int i, j;

    #pragma omp parallel for private(i, j) shared(array)
    for (i = 0; i < n-1; i++) {
        for (j = 0; j < n-i-1; j++) {
            if (array[j] > array[j+1]) {
                // Swap elements
```

```
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
}

// Parallel Merge Sort function
void merge(int *array, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    // Create temp arrays
    int *L = new int[n1];
    int *R = new int[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = array[l + i];
    for (j = 0; j < n2; j++)
        R[j] = array[m + 1+ j];

    // Merge the temp arrays back into array[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }

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

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
```

```
        array[k] = R[j];
        j++;
        k++;
    }

    delete [] L;
    delete [] R;
}

void parallelMergeSort(int *array, int l, int r) {
    if (l < r) {
        int m = l+(r-l)/2;

        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(array, l, m);

            #pragma omp section
            parallelMergeSort(array, m+1, r);
        }

        merge(array, l, m, r);
    }
}

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

    int *array = new int[n];

    // srand(time(0));

    // for (int i = 0; i < n; i++) {
    //     array[i] = rand() % 100;
    // }

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

    cout << "Original Array: ";

    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }
```

```
    cout << endl;

    int choice;
    cout << "Enter 1 for Parallel Bubble Sort or 2 for Parallel Merge Sort: ";
    cin >> choice;

    if (choice == 1) {
        parallelBubbleSort(array, n);
    } else if (choice == 2) {
        parallelMergeSort(array, 0, n-1);
    } else {
        cout << "Invalid choice. Exiting program." << endl;
        return 0;
    }

    cout << "Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    delete [] array;

    return 0;
}









// Test Case 1:
// Input:
// Enter the size of the array: 5
// Enter 1 for Parallel Bubble Sort or 2 for Parallel Merge Sort: 1

// Output:
// Original Array: 68 67 69 73 29
// Sorted Array: 29 67 68 69 73
```

Min max parallel reduction :

```
#include <iostream>

#include <vector>
#include <limits.h>
#include <omp.h>

using namespace std;

void min_reduction(vector<int>& arr) {
    int min_value = INT_MAX;
#pragma omp parallel for reduction(min: min_value)
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    cout << "Minimum value: " << min_value << endl;
}

void max_reduction(vector<int>& arr) {
    int max_value = INT_MIN;
#pragma omp parallel for reduction(max: max_value)
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }
    cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(vector<int>& arr) {
    int sum = 0;
#pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
    }
    cout << "Sum: " << sum << endl;
}

void average_reduction(vector<int>& arr) {
    int sum = 0;
#pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
    }
    cout << "Average: " << (double)sum / arr.size() << endl;
}
```

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

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

    min_reduction(arr);
    max_reduction(arr);
    sum_reduction(arr);
    average_reduction(arr);
    return 0;
}

// Output:
// Enter the size of the array: 5
// Enter the elements of the array:
// 5 6 7 2 1
// Minimum value: 1
// Maximum value: 7
// Sum: 21
// Average: 4.2
```