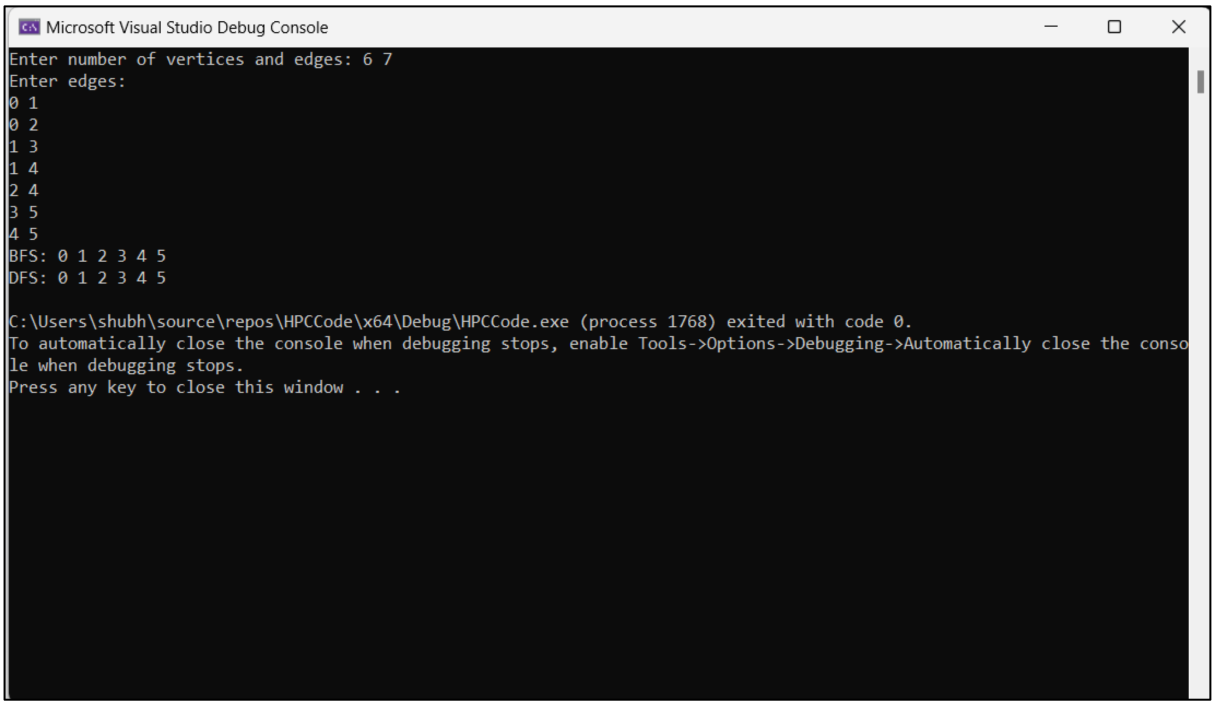
Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .

Code –

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

using namespace std;

void parallel_bfs(int start, vector<vector<int>> adj_list, vector<bool>& visited)
{
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int curr = q.front();
        q.pop();

#pragma omp parallel for shared(adj_list, visited, q)
        for (int neighbor : adj_list[curr]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

void parallel_dfs(int curr, vector<vector<int>> adj_list, vector<bool>& visited)
{
    visited[curr] = true;

#pragma omp parallel for shared(adj_list, visited)
    for (int neighbor : adj_list[curr]) {
        if (!visited[neighbor]) {
#pragma omp critical
            parallel_dfs(neighbor, adj_list, visited);
        }
    }
}

int main() {
    int n, m;
    cout << "Enter number of vertices and edges: ";
    cin >> n >> m;

    vector<vector<int>> adj_list(n);
    vector<bool> visited(n, false);

    cout << "Enter edges:" << endl;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        adj_list[a].push_back(b);
        adj_list[b].push_back(a);
    }
    cout << "BFS: ";
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            parallel_bfs(i, adj_list, visited);
        }
        cout << i << " ";
    }
    cout << endl;

    visited.assign(n, false);

    cout << "DFS: ";
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            parallel_dfs(i, adj_list, visited);
        }
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

Output :



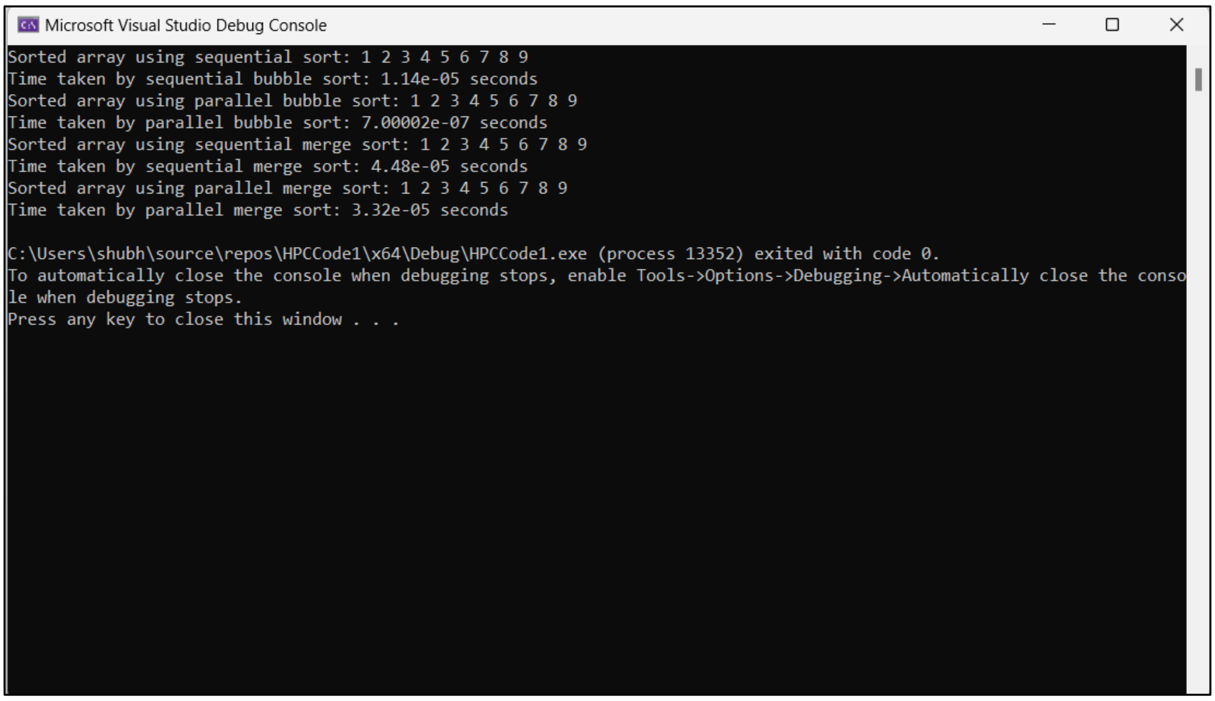
```
Enter number of vertices and edges: 6 7
Enter edges:
0 1
0 2
1 3
1 4
2 4
3 5
4 5
BFS: 0 1 2 3 4 5
DFS: 0 1 2 3 4 5

C:\Users\shubh\source\repos\HPCCode\x64\Debug\HPCCode.exe (process 1768) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Code –

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

using namespace std;

void parallel_bubble_sort(vector<int>& arr, int n) {
    int i, j, temp;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
#pragma omp parallel for private(j, temp) shared(swapped, arr)
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (swapped == false) {
            break;
        }
    }
}

void merge(vector<int>& arr, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    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 sequential_merge_sort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        sequential_merge_sort(arr, l, m);
        sequential_merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void parallel_merge_sort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
#pragma omp parallel sections
        {
#pragma omp section
            {
                parallel_merge_sort(arr, l, m);
            }
#pragma omp section
            {
                parallel_merge_sort(arr, m + 1, r);
            }
        }
        merge(arr, l, m, r);
    }
}

int main() {
    vector<int> arr = { 5, 2, 7, 3, 1, 8, 4, 9, 6 };
    int n = arr.size();
    double start_time, end_time;

    // Sequential Bubble Sort
    start_time = omp_get_wtime();
    sort(arr.begin(), arr.end());
    end_time = omp_get_wtime();

    cout << "Sorted array using sequential sort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << "Time taken by sequential bubble sort: " << end_time -
start_time << " seconds" << endl;
    // Parallel Bubble Sort
    start_time = omp_get_wtime();
    parallel_bubble_sort(arr, n);
    end_time = omp_get_wtime();

    cout << "Sorted array using parallel bubble sort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << "Time taken by parallel bubble sort: " << end_time -
start_time << " seconds" << endl;

    // Sequential Merge Sort
    arr = { 5, 2, 7, 3, 1, 8, 4, 9, 6 };
    start_time = omp_get_wtime();
    sequential_merge_sort(arr, 0, n - 1);
    end_time = omp_get_wtime();

    cout << "Sorted array using sequential merge sort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << "Time taken by sequential merge sort: " << end_time -
start_time << " seconds" << endl;

    // Parallel Merge Sort
    arr = { 5, 2, 7, 3, 1, 8, 4, 9, 6 };
    start_time = omp_get_wtime();
    parallel_merge_sort(arr, 0, n - 1);
    end_time = omp_get_wtime();

    cout << "Sorted array using parallel merge sort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << "Time taken by parallel merge sort: " << end_time -
start_time << " seconds" << endl;

    return 0;
}
```

Output :



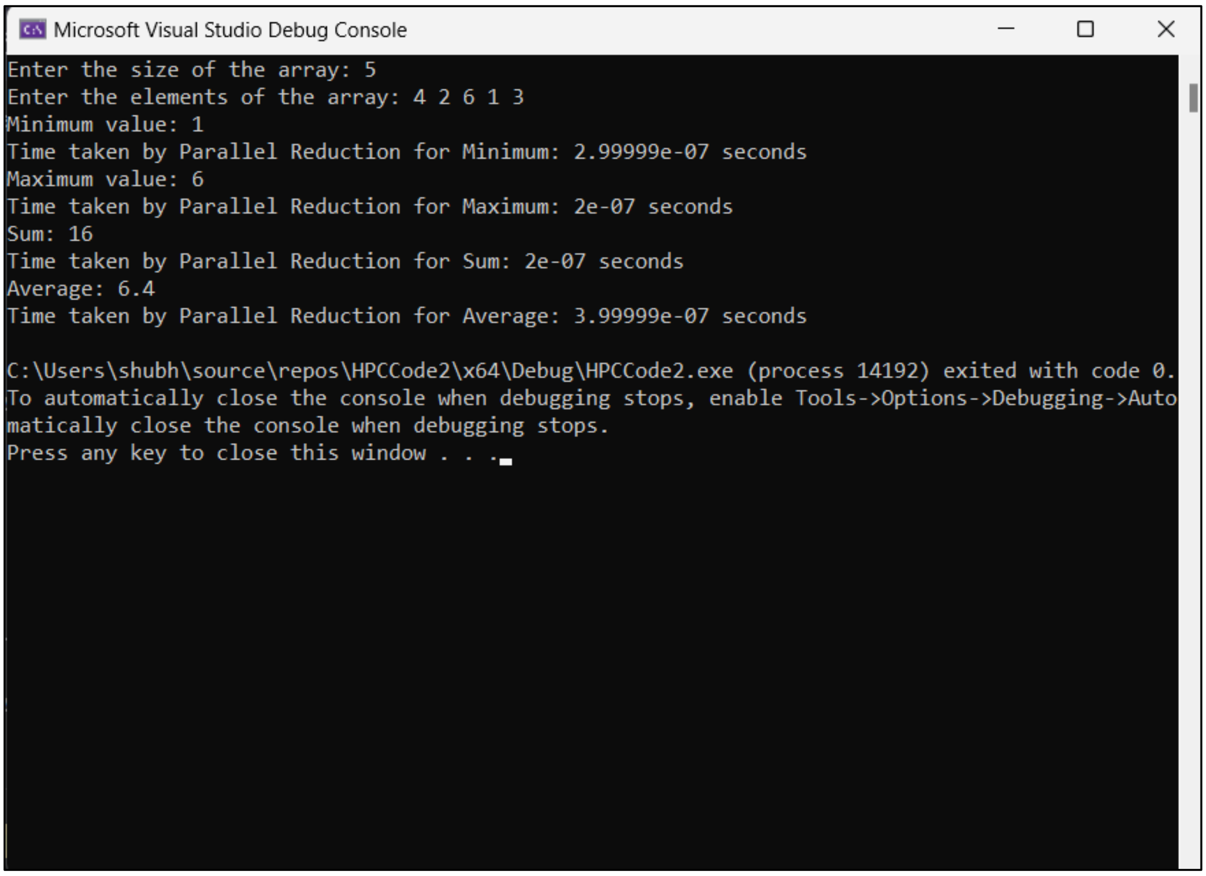
```
Sorted array using sequential sort: 1 2 3 4 5 6 7 8 9
Time taken by sequential bubble sort: 1.14e-05 seconds
Sorted array using parallel bubble sort: 1 2 3 4 5 6 7 8 9
Time taken by parallel bubble sort: 7.00002e-07 seconds
Sorted array using sequential merge sort: 1 2 3 4 5 6 7 8 9
Time taken by sequential merge sort: 4.48e-05 seconds
Sorted array using parallel merge sort: 1 2 3 4 5 6 7 8 9
Time taken by parallel merge sort: 3.32e-05 seconds

C:\Users\shubh\source\repos\HPCCode1\x64\Debug\HPCCode1.exe (process 13352) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Implement Min, Max, Sum and Average operations using Parallel Reduction.

Code –

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

using namespace std;

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

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

    double start_time, end_time;

    // Parallel Reduction for Minimum
    double min_val = arr[0];
    start_time = omp_get_wtime();
#pragma omp parallel for reduction(min:min_val)
    for (int i = 0; i < n; i++) {
        if (arr[i] < min_val) {
            min_val = arr[i];
        }
    }
    end_time = omp_get_wtime();
    cout << "Minimum value: " << min_val << endl;
    cout << "Time taken by Parallel Reduction for Minimum: " << end_time -
start_time << " seconds" << endl;

    // Parallel Reduction for Maximum
    double max_val = arr[0];
    start_time = omp_get_wtime();
#pragma omp parallel for reduction(max:max_val)
    for (int i = 0; i < n; i++) {
        if (arr[i] > max_val) {
            max_val = arr[i];
        }
    }
    end_time = omp_get_wtime();
    cout << "Maximum value: " << max_val << endl;
    cout << "Time taken by Parallel Reduction for Maximum: " << end_time -
start_time << " seconds" << endl;

    // Parallel Reduction for Sum
    double sum = 0;
    start_time = omp_get_wtime();
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    end_time = omp_get_wtime();
    cout << "Sum: " << sum << endl;
    cout << "Time taken by Parallel Reduction for Sum: " << end_time - start_time
<< " seconds" << endl;

    // Parallel Reduction for Average
    double avg = 0;
    start_time = omp_get_wtime();
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    avg = sum / n;
    end_time = omp_get_wtime();
    cout << "Average: " << avg << endl;
    cout << "Time taken by Parallel Reduction for Average: " << end_time -
start_time << " seconds" << endl;

    return 0;
}
```

Output :



```
Enter the size of the array: 5
Enter the elements of the array: 4 2 6 1 3
Minimum value: 1
Time taken by Parallel Reduction for Minimum: 2.99999e-07 seconds
Maximum value: 6
Time taken by Parallel Reduction for Maximum: 2e-07 seconds
Sum: 16
Time taken by Parallel Reduction for Sum: 2e-07 seconds
Average: 6.4
Time taken by Parallel Reduction for Average: 3.99999e-07 seconds

C:\Users\shubh\source\repos\HPCCode2\x64\Debug\HPCCode2.exe (process 14192) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Auto
matically close the console when debugging stops.
Press any key to close this window . . .
```

```
%%file vec_add.cu
#include <stdio.h>

__global__ void add_vectors(int *a, int *b, int *c, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        c[tid] = a[tid] + b[tid];
    }
}

int main(void) {
    // Define the size of the input vectors
    int size = 5;

    // Allocate memory on the host for the input and output vectors
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {6, 7, 8, 9, 10};
    int c[size];

    // Allocate memory on the device for the input and output vectors
    int *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, size * sizeof(int));
    cudaMalloc(&d_b, size * sizeof(int));
    cudaMalloc(&d_c, size * sizeof(int));

    // Copy the input vectors from the host to the device
    cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

    // Launch the kernel to add the vectors on the device
    int block_size = 256;
    int grid_size = (size + block_size - 1) / block_size;
    add_vectors<<<grid_size, block_size>>>(d_a, d_b, d_c, size);

    // Copy the output vector from the device to the host
    cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);

    // Print the input and output vectors
    printf("a = [ ");
    for (int i = 0; i < size; i++) {
        printf("%d ", a[i]);
    }
    printf("]\n");

    printf("b = [ ");
    for (int i = 0; i < size; i++) {
        printf("%d ", b[i]);
    }
    printf("]\n");

    printf("c = [ ");
    for (int i = 0; i < size; i++) {
        printf("%d ", c[i]);
    }
    printf("]\n");

    // Free memory on the device
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

```
Writing vec_add.cu
```

```
!nvcc vec_add.cu -o vec_add
```

```
!./vec_add
```

```
a = [ 1 2 3 4 5 ]
b = [ 6 7 8 9 10 ]
c = [ 7 9 11 13 15 ]
```

```
%%file matrix_multiplication.cu
#include <stdio.h>

__global__ void matrixMultiply(float *a, float *b, float *c, int m, int n, int k) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;

    if (i < m && j < k) {
        for (int p = 0; p < n; p++) {
            sum += a[i * n + p] * b[p * k + j];
        }
        c[i * k + j] = sum;
    }
}

int main() {
    int m = 3, n = 2, k = 4;
    float a[m][n] = {{1, 2}, {3, 4}, {5, 6}};
    float b[n][k] = {{7, 8, 9, 10}, {11, 12, 13, 14}};
    float c[m][k];

    float *dev_a, *dev_b, *dev_c;
    cudaMalloc(&dev_a, m * n * sizeof(float));
    cudaMalloc(&dev_b, n * k * sizeof(float));
    cudaMalloc(&dev_c, m * k * sizeof(float));

    cudaMemcpy(dev_a, a, m * n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, n * k * sizeof(float), cudaMemcpyHostToDevice);

    dim3 grid((k - 1) / 32 + 1, (m - 1) / 32 + 1, 1);
    dim3 block(32, 32, 1);

    matrixMultiply<<<grid, block>>>(dev_a, dev_b, dev_c, m, n, k);

    cudaMemcpy(c, dev_c, m * k * sizeof(float), cudaMemcpyDeviceToHost);

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            printf("%f ", c[i][j]);
        }
        printf("\n");
    }

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

```
Writing matrix_multiplication.cu
```

```
!nvcc matrix_multiplication.cu -o matrix_multiplication
```

```
!./matrix_multiplication
```

```
29.000000 32.000000 35.000000 38.000000
65.000000 72.000000 79.000000 86.000000
101.000000 112.000000 123.000000 134.000000
```