HPC-1
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

Parallel BFS and DFS using OpenMP (C++)

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <omp.h>
using namespace std;

// Parallel BFS
void parallelBFS(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    cout << "Parallel BFS: ";

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

        #pragma omp parallel for
        for (int i = 0; i < graph[node].size(); i++) {
            int neighbor = graph[node][i];
            #pragma omp critical
            {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }
    cout << endl;
}

// Parallel DFS
void parallelDFS(vector<vector<int>>& graph, int start) {
    int n = graph.size();
    vector<bool> visited(n, false);
    stack<int> s;

    s.push(start);

    cout << "Parallel DFS: ";
```

```
    while (!s.empty()) {
        int node;

        #pragma omp critical
        {
            node = s.top();
            s.pop();
        }

        if (!visited[node]) {
            visited[node] = true;
            cout << node << " ";

            #pragma omp parallel for
            for (int i = graph[node].size() - 1; i >= 0; i--) { // reverse for correct order
                int neighbor = graph[node][i];
                if (!visited[neighbor]) {
                    #pragma omp critical
                    s.push(neighbor);
                }
            }
        }
    }
    cout << endl;
}

int main() {
    int n = 7; // 7 nodes (0 to 6)
    vector<vector<int>> graph(n);

    // Undirected graph (tree like)
    graph[0] = {1, 2};
    graph[1] = {0, 3, 4};
    graph[2] = {0, 5, 6};
    graph[3] = {1};
    graph[4] = {1};
    graph[5] = {2};
    graph[6] = {2};

    parallelBFS(graph, 0);
    parallelDFS(graph, 0);

    return 0;
}
```

HPC-2

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.

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
#include <chrono>
#include <omp.h>
using namespace std;
using namespace chrono;

// Sequential Bubble Sort
void bubbleSortSequential(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]);
}

// Parallel Bubble Sort
void bubbleSortParallel(vector<int>& arr) {
    int n = arr.size();
    for(int i=0; i<n; ++i) {
        #pragma omp parallel for
        for(int j=i%2; j<n-1; j+=2)
            if(arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
    }
}

// Merge function
void merge(vector<int>& arr, int l, int m, int r) {
    vector<int> left(arr.begin() + l, arr.begin() + m + 1);
    vector<int> right(arr.begin() + m + 1, arr.begin() + r + 1);
    int i=0, j=0, k=l;
    while(i<left.size() && j<right.size())
        arr[k++] = (left[i] <= right[j]) ? left[i++] : right[j++];
    while(i<left.size()) arr[k++] = left[i++];
    while(j<right.size()) arr[k++] = right[j++];
}

// Sequential Merge Sort
void mergeSortSequential(vector<int>& arr, int l, int r) {
    if(l<r) {
        int m = (l+r)/2;
        mergeSortSequential(arr, l, m);
        mergeSortSequential(arr, m+1, r);
        merge(arr, l, m, r);
    }
}

// Parallel Merge Sort
void mergeSortParallel(vector<int>& arr, int l, int r) {
    if(l<r) {
        int m = (l+r)/2;
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSortParallel(arr, l, m);
```

```
            #pragma omp section
            mergeSortParallel(arr, m+1, r);
        }
        merge(arr, l, m, r);
    }
}

// Helper to measure and print time
template<typename Func>
void measureTime(string label, Func sortFunc, vector<int> arr) {
    auto start = high_resolution_clock::now();
    sortFunc(arr);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start).count();
    cout << label << ": " << duration << " ms" << endl;
}

int main() {
    int n = 5000;
    vector<int> arr(n);
    for(int i=0; i<n; ++i)
        arr[i] = rand() % 10000;

    cout << "Sorting " << n << " elements:\n";

    measureTime("Sequential Bubble Sort", [](vector<int> a){ bubbleSortSequential(a); }, arr);
    measureTime("Parallel Bubble Sort", [](vector<int> a){ bubbleSortParallel(a); }, arr);
    measureTime("Sequential Merge Sort", [](vector<int> a){ mergeSortSequential(a, 0, a.size()-1); }, arr);
    measureTime("Parallel Merge Sort", [](vector<int> a){ mergeSortParallel(a, 0, a.size()-1); }, arr);

    return 0;
}
```

_
HPC-3
Implement Min, Max, Sum and Average operations using Parallel Reduction
----

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

int main() {
    int n = 1000000;
    vector<int> arr(n);

    // Initialize the array with random values
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000; // Random numbers between 0 and 999
    }

    int min_val = arr[0];
```

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

    // Parallel Reduction
    #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 = (double)sum / n;

    // Output results
    cout << "Minimum Value: " << min_val << endl;
    cout << "Maximum Value: " << max_val << endl;
    cout << "Sum: " << sum << endl;
    cout << "Average: " << average << endl;

    return 0;
}
```

_
HPC-4A
Write a CUDA Program for: 1. Addition of two large vectors 2. Matrix Multiplication using CUDA C

----

1. Vector Addition using CUDA

```
#include <iostream>
#include <cuda_runtime.h>
using namespace std;

_global_ void vectorAdd(const int *A, const int *B, int *C, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n)
        C[idx] = A[idx] + B[idx];
}

int main() {
    int n = 1<<20; // 1 million elements
    size_t size = n * sizeof(int);

    int *h_A = new int[n];
    int *h_B = new int[n];
    int *h_C = new int[n];

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

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
```

```
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cout << "First 5 results:\n";
    for (int i = 0; i < 5; i++)
        cout << h_A[i] << " + " << h_B[i] << " = " << h_C[i] << endl;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

HPC-4B

----

2. Matrix Multiplication using CUDA

```
#include <iostream>
#include <cuda_runtime.h>
using namespace std;

#define N 512  // Size of matrix (NxN)

_global_ void matrixMul(int *A, int *B, int *C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;

    if(row < width && col < width) {
        for (int k = 0; k < width; k++) {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}

int main() {
    int size = N * N * sizeof(int);
    int *h_A = new int[N*N];
```

```
    int *h_B = new int[N*N];
    int *h_C = new int[N*N];

    for (int i = 0; i < N*N; i++) {
        h_A[i] = rand() % 10;
        h_B[i] = rand() % 10;
    }

    int *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((N + 15) / 16, (N + 15) / 16);
    matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cout << "First 5 results:\n";
    for (int i = 0; i < 5; i++)
        cout << h_C[i] << " ";
    cout << endl;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```