**BFS & DFS**

%%writefile bfs.cpp

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <omp.h>

using namespace std;

class Graph {

private:

    int numVertices;

    vector<vector<int>> adj;

public:

    Graph(int vertices) {

        numVertices = vertices;

        adj.resize(vertices);

    }

    void addEdge(int src, int dest) {

        adj[src].push\_back(dest);

        adj[dest].push\_back(src);

    }

    void viewGraph() {

        cout << "\nGraph:\n";

        for (int i = 0; i < numVertices; i++) {

            cout << "Vertex " << i << " -> ";

            for (int neighbor : adj[i]) {

                cout << neighbor << " ";

            }

            cout << endl;

        }

    }

    // Sequential BFS

    void sequentialBFS(int startVertex) {

        vector<bool> visited(numVertices, false);

        queue<int> q;

        visited[startVertex] = true;

        q.push(startVertex);

        cout << "\nSequential BFS Output:\n";

        while (!q.empty()) {

            int currentVertex = q.front();

            q.pop();

            cout << currentVertex << endl;

            for (int neighbor : adj[currentVertex]) {

                if (!visited[neighbor]) {

                    visited[neighbor] = true;

                    q.push(neighbor);

                }

            }

        }

    }

    // Sequential DFS

    void sequentialDFS(int startVertex) {

        vector<bool> visited(numVertices, false);

        stack<int> s;

        visited[startVertex] = true;

        s.push(startVertex);

        cout << "\nSequential DFS Output:\n";

        while (!s.empty()) {

            int currentVertex = s.top();

            s.pop();

            cout << currentVertex << endl;

            for (int neighbor : adj[currentVertex]) {

                if (!visited[neighbor]) {

                    visited[neighbor] = true;

                    s.push(neighbor);

                }

            }

        }

    }

    // Parallel BFS (no thread logs)

    void parallelBFS(int startVertex) {

        vector<bool> visited(numVertices, false);

        queue<int> q;

        visited[startVertex] = true;

        q.push(startVertex);

        cout << "\nParallel BFS Output:\n";

        while (!q.empty()) {

            int currentVertex = q.front();

            q.pop();

            #pragma omp critical

            {

                cout << currentVertex << endl;

            }

            vector<int> neighbors = adj[currentVertex];

            #pragma omp parallel for

            for (int i = 0; i < neighbors.size(); i++) {

                int neighbor = neighbors[i];

                #pragma omp critical

                {

                    if (!visited[neighbor]) {

                        visited[neighbor] = true;

                        q.push(neighbor);

                    }

                }

            }

        }

    }

    // Parallel DFS (no thread logs)

    void parallelDFS(int startVertex) {

        vector<bool> visited(numVertices, false);

        stack<int> s;

        visited[startVertex] = true;

        s.push(startVertex);

        cout << "\nParallel DFS Output:\n";

        while (!s.empty()) {

            int currentVertex = s.top();

            s.pop();

            #pragma omp critical

            {

                cout << currentVertex << endl;

            }

            vector<int> neighbors = adj[currentVertex];

            #pragma omp parallel for

            for (int i = 0; i < neighbors.size(); i++) {

                int neighbor = neighbors[i];

                #pragma omp critical

                {

                    if (!visited[neighbor]) {

                        visited[neighbor] = true;

                        s.push(neighbor);

                    }

                }

            }

        }

    }

};

int main() {

    int numVertices, numEdges;

    cout << "Enter number of vertices: ";

    cin >> numVertices;

    Graph graph(numVertices);

    cout << "Enter number of edges: ";

    cin >> numEdges;

    cout << "Enter the edges (source destination):\n";

    for (int i = 0; i < numEdges; i++) {

        int src, dest;

        cin >> src >> dest;

        graph.addEdge(src, dest);

    }

    graph.viewGraph();

    int startVertex;

    cout << "Enter the starting vertex: ";

    cin >> startVertex;

    double start, end;

    // Sequential BFS

    start = omp\_get\_wtime();

    graph.sequentialBFS(startVertex);

    end = omp\_get\_wtime();

    cout << "Sequential BFS Time: " << (end - start) << " seconds\n";

    // Sequential DFS

    start = omp\_get\_wtime();

    graph.sequentialDFS(startVertex);

    end = omp\_get\_wtime();

    cout << "Sequential DFS Time: " << (end - start) << " seconds\n";

    // Parallel BFS

    start = omp\_get\_wtime();

    graph.parallelBFS(startVertex);

    end = omp\_get\_wtime();

    cout << "Parallel BFS Time: " << (end - start) << " seconds\n";

    // Parallel DFS

    start = omp\_get\_wtime();

    graph.parallelDFS(startVertex);

    end = omp\_get\_wtime();

    cout << "Parallel DFS Time: " << (end - start) << " seconds\n";

    return 0;

}

//!g++ bfs.cpp -fopenmp

//./a.out

// g++ practical\_1.cpp -o practical\_1 -fopenmp

// ./Practical\_1

**Bubble Sort & Merge Sort**

//%%writefile bubble\_sort.cpp

#include <iostream>

#include <cstdlib>

#include <omp.h>

using namespace std;

// Sequential Bubble Sort

void bubbleSort(int arr[], int n) {

    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 using odd-even transposition

void parallelBubbleSort(int arr[], int n) {

    for (int i = 0; i < n; ++i) {

        int start = i % 2;

        #pragma omp parallel for

        for (int j = start; j < n - 1; j += 2) {

            if (arr[j] > arr[j + 1]) {

                swap(arr[j], arr[j + 1]);

            }

        }

    }

}

// Merge function

void merge(int arr[], int l, int m, int r) {

    int n1 = m - l + 1;

    int n2 = r - m;

    int \*L = new int[n1];

    int \*R = new int[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)

        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1)

        arr[k++] = L[i++];

    while (j < n2)

        arr[k++] = R[j++];

    delete[] L;

    delete[] R;

}

// Parallel Merge Sort

void mergeSort(int arr[], int l, int r, int d = 0) {

    if (l < r) {

        int m = l + (r - l) / 2;

        if (d > 4) {

            mergeSort(arr, l, m, d + 1);

            mergeSort(arr, m + 1, r, d + 1);

        } else {

            #pragma omp parallel sections

            {

                #pragma omp section

                mergeSort(arr, l, m, d + 1);

                #pragma omp section

                mergeSort(arr, m + 1, r, d + 1);

            }

        }

        merge(arr, l, m, r);

    }

}

// Utility function to print an array

void printArray(const string& label, int arr[], int n) {

    cout << label;

    for (int i = 0; i < n; ++i) {

        cout << arr[i] << " ";

    }

    cout << "\n";

}

int main() {

    int n;

    cout << "Enter the size of the array: ";

    cin >> n;

    int \*original = new int[n];

    for (int i = 0; i < n; ++i)

        original[i] = rand() % 100;

    printArray("\nOriginal Array: ", original, n);

    // Create copies

    int \*arr1 = new int[n]; // Sequential Bubble

    int \*arr2 = new int[n]; // Parallel Bubble

    int \*arr3 = new int[n]; // Sequential Merge

    int \*arr4 = new int[n]; // Parallel Merge

    for (int i = 0; i < n; ++i)

        arr1[i] = arr2[i] = arr3[i] = arr4[i] = original[i];

    double start, end;

    // Sequential Bubble Sort

    start = omp\_get\_wtime();

    bubbleSort(arr1, n);

    end = omp\_get\_wtime();

    double time\_seq\_bubble = end - start;

    printArray("\nSequential Bubble Sort Output: ", arr1, n);

    // Parallel Bubble Sort

    start = omp\_get\_wtime();

    parallelBubbleSort(arr2, n);

    end = omp\_get\_wtime();

    double time\_par\_bubble = end - start;

    printArray("Parallel Bubble Sort Output:   ", arr2, n);

    // Sequential Merge Sort

    start = omp\_get\_wtime();

    mergeSort(arr3, 0, n - 1, 10); // depth 10 to force sequential

    end = omp\_get\_wtime();

    double time\_seq\_merge = end - start;

    printArray("Sequential Merge Sort Output:  ", arr3, n);

    // Parallel Merge Sort

    start = omp\_get\_wtime();

    mergeSort(arr4, 0, n - 1, 0); // default depth triggers parallelism

    end = omp\_get\_wtime();

    double time\_par\_merge = end - start;

    printArray("Parallel Merge Sort Output:    ", arr4, n);

    // Performance Results

    cout << "\nPerformance Comparison (in seconds):\n";

    cout << "Sequential Bubble Sort: " << time\_seq\_bubble << endl;

    cout << "Parallel Bubble Sort:   " << time\_par\_bubble << endl;

    cout << "Sequential Merge Sort:  " << time\_seq\_merge << endl;

    cout << "Parallel Merge Sort:    " << time\_par\_merge << endl;

    delete[] original;

    delete[] arr1;

    delete[] arr2;

    delete[] arr3;

    delete[] arr4;

    return 0;

}  
  
//!g++ bubble\_sort.cpp -fopenmp  
//!./a.out