BFS TRAVERSAL

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

void parallelBFS(const 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 << "BFS Traversal: ";

    while (!q.empty()) {

        int levelSize = q.size();

        // Process all nodes at the current level in parallel

        vector<int> nextLevel;

        #pragma omp parallel for shared(q, visited) default(none) firstprivate(levelSize) schedule(dynamic)

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

            int u;

            #pragma omp critical

            {

                if (!q.empty()) {

                    u = q.front();

                    q.pop();

                    cout << u << " ";

                }

            }

            // Traverse neighbors

            for (int v : graph[u]) {

                if (!visited[v]) {

                    #pragma omp critical

                    {

                        if (!visited[v]) {

                            visited[v] = true;

                            nextLevel.push\_back(v);

                        }

                    }

                }

            }

        }

        // Add all next level nodes to the queue

        for (int v : nextLevel) {

            q.push(v);

        }

    }

    cout << endl;

}

int main() {

    // Example graph as an adjacency list

    vector<vector<int>> graph = {

        {1, 2},    // Neighbors of node 0

        {0, 3, 4}, // Neighbors of node 1

        {0, 4},    // Neighbors of node 2

        {1, 5},    // Neighbors of node 3

        {1, 2, 5}, // Neighbors of node 4

        {3, 4}     // Neighbors of node 5

    };

    int startNode = 0;

    parallelBFS(graph, startNode);

    return 0;

}

MATRIX MULTIPLICATION

#include <iostream>

#include <vector>

#include <omp.h>

#include <cstdlib>

#include <ctime>

using namespace std;

// Function to generate a random matrix

void generateRandomMatrix(vector<vector<int>>& matrix, int rows, int cols) {

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            matrix[i][j] = rand() % 10; // Random numbers between 0 and 9

        }

    }

}

// Function to perform parallel matrix multiplication

void parallelMatrixMultiplication(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C, int num\_threads) {

    int rows = A.size();

    int cols = B[0].size();

    int common\_dim = B.size();

    #pragma omp parallel for num\_threads(num\_threads) collapse(2)

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            C[i][j] = 0;

            for (int k = 0; k < common\_dim; ++k) {

                C[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

}

int main() {

    srand(time(0)); // Seed for random number generation

    int rowsA, colsA, rowsB, colsB;

    int num\_threads = 4; // Number of threads

    // User input for matrix dimensions

    cout << "Enter number of rows and columns for Matrix A: ";

    cin >> rowsA >> colsA;

    cout << "Enter number of rows and columns for Matrix B: ";

    cin >> rowsB >> colsB;

    if (colsA != rowsB) {

        cout << "Matrix multiplication not possible: Number of columns in A must be equal to number of rows in B." << endl;

        return 1;

    }

    vector<vector<int>> A(rowsA, vector<int>(colsA));

    vector<vector<int>> B(rowsB, vector<int>(colsB));

    vector<vector<int>> C(rowsA, vector<int>(colsB, 0));

    generateRandomMatrix(A, rowsA, colsA);

    generateRandomMatrix(B, rowsB, colsB);

    // Perform matrix multiplication

    parallelMatrixMultiplication(A, B, C, num\_threads);

    // Print Matrix A

    cout << "Matrix A:\n";

    for (const auto& row : A) {

        for (int val : row) {

            cout << val << " ";

        }

        cout << "\n";

    }

    // Print Matrix B

    cout << "Matrix B:\n";

    for (const auto& row : B) {

        for (int val : row) {

            cout << val << " ";

        }

        cout << "\n";

    }

    // Print the result matrix

    cout << "Resultant Matrix:\n";

    for (const auto& row : C) {

        for (int val : row) {

            cout << val << " ";

        }

        cout << "\n";

    }

    return 0;

}

DJIKSTRA ALGO

#include <iostream>

#include <vector>

#include <climits>

#include <omp.h>

using namespace std;

const int V = 6; // Number of vertices

// Find the vertex with minimum distance value not yet included in shortest path

int minDistance(const vector<int>& dist, const vector<bool>& visited) {

    int min = INT\_MAX, min\_index = -1;

    #pragma omp parallel for

    for (int v = 0; v < V; v++) {

        if (!visited[v] && dist[v] < min) {

            #pragma omp critical

            {

                if (dist[v] < min) {

                    min = dist[v];

                    min\_index = v;

                }

            }

        }

    }

    return min\_index;

}

// Dijkstra algorithm

void dijkstra(int graph[V][V], int src) {

    vector<int> dist(V, INT\_MAX); // Distance from source to each vertex

    vector<bool> visited(V, false); // Visited vertices

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, visited);

        visited[u] = true;

        // Update dist value of adjacent vertices of u

        #pragma omp parallel for

        for (int v = 0; v < V; v++) {

            if (!visited[v] && graph[u][v] && dist[u] != INT\_MAX

                && dist[u] + graph[u][v] < dist[v]) {

                dist[v] = dist[u] + graph[u][v];

            }

        }

    }

    // Print the result

    cout << "Vertex\tDistance from Source\n";

    for (int i = 0; i < V; i++)

        cout << i << "\t" << dist[i] << "\n";

}

int main() {

    int graph[V][V] = {

        {0, 2, 0, 1, 0, 0},

        {2, 0, 4, 0, 3, 0},

        {0, 4, 0, 5, 1, 0},

        {1, 0, 5, 0, 2, 8},

        {0, 3, 1, 2, 0, 6},

        {0, 0, 0, 8, 6, 0}

    };

    cout << endl;

    dijkstra(graph, 0);

    return 0;

}

HISTOGRAM SORTING

#include <iostream>

#include <vector>

#include <omp.h>

#include <algorithm>

using namespace std;

// Parameters

const int NUM\_BINS = 10; // Number of histogram bins

const int MAX\_VALUE = 100; // Max value for elements

// Function to get bin index for a value

int get\_bin(int value) {

    return value \* NUM\_BINS / (MAX\_VALUE + 1);

}

int main() {

    // Input data

    vector<int> data = {42, 23, 56, 78, 10, 3, 91, 17, 33, 65, 81, 12, 7, 88, 45, 60};

    int n = data.size();

    // Step 1: Create local histograms for each thread

    vector<vector<int>> bins(NUM\_BINS);

    #pragma omp parallel

    {

        // Thread-local bin storage

        vector<vector<int>> local\_bins(NUM\_BINS);

        #pragma omp for nowait

        for (int i = 0; i < n; ++i) {

            int bin\_index = get\_bin(data[i]);

            local\_bins[bin\_index].push\_back(data[i]);

        }

        // Merge local bins into global bins (critical section for safety)

        #pragma omp critical

        {

            for (int i = 0; i < NUM\_BINS; ++i) {

                bins[i].insert(bins[i].end(), local\_bins[i].begin(), local\_bins[i].end());

            }

        }

    }

    // Step 2: Sort each bin (can also be done in parallel)

    #pragma omp parallel for

    for (int i = 0; i < NUM\_BINS; ++i) {

        sort(bins[i].begin(), bins[i].end());

    }

    // Step 3: Merge all bins back into sorted data

    vector<int> sorted;

    for (int i = 0; i < NUM\_BINS; ++i) {

        sorted.insert(sorted.end(), bins[i].begin(), bins[i].end());

    }

    // Print sorted data

    cout << "Sorted Data: ";

    for (int x : sorted) {

        cout << x << " ";

    }

    cout << endl;

    return 0;

}