**IPP Practical**

**Matrix Multiplication**

Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 500  // Matrix size

int A[N][N], B[N][N], C[N][N];  // Declare matrices statically

int main() {

    // Initialize matrices with random values

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++) {

            A[i][j] = rand() % 10;

            B[i][j] = rand() % 10;

            C[i][j] = 0;

        }

    // Matrix multiplication with OpenMP parallelization

    #pragma omp parallel for

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            for (int k = 0; k < N; k++)

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

    printf("Matrix multiplication completed.\n");

    return 0;

}

**BFS**

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

// Graph representation using adjacency list

class Graph {

public:

    int V; // Number of vertices

    vector<vector<int>> adj; // Adjacency list

    Graph(int V) {

        this->V = V;

        adj.resize(V);

    }

    void addEdge(int u, int v) {

        adj[u].push\_back(v);

        adj[v].push\_back(u); // Undirected graph

    }

    void parallelBFS(int start) {

        vector<bool> visited(V, false);

        queue<int> q;

        visited[start] = true;

        q.push(start);

        cout << "Parallel BFS Traversal: ";

        while (!q.empty()) {

            // Process current level sequentially (to maintain BFS order)

            int level\_size = q.size();

            // Print and dequeue nodes at the current level

            for (int i = 0; i < level\_size; i++) {

                int node = q.front();

                q.pop();

                cout << node << " ";

                // Explore neighbors in parallel

                #pragma omp parallel for

                for (size\_t j = 0; j < adj[node].size(); j++) {

                    int neighbor = adj[node][j];

                    // Use critical section only for marking visited and enqueueing

                    #pragma omp critical

                    {

                        if (!visited[neighbor]) {

                            visited[neighbor] = true;

                            q.push(neighbor);

                        }

                    }

                }

            }

        }

        cout << endl;

    }

};

int main() {

    Graph g(7);

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 3);

    g.addEdge(1, 4);

    g.addEdge(2, 5);

    g.addEdge(2, 6);

    g.parallelBFS(0);

    return 0;

}

**Dijkstra**

**Code:**

#include <iostream>

#include <vector>

#include <climits>

#include <omp.h>

using namespace std;

#define V 6

// Serial version (must remain sequential)

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

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

    for (int v = 0; v < V; v++) {

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

            min = dist[v];

            min\_index = v;

        }

    }

    return min\_index;

}

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

    vector<int> dist(V, INT\_MAX);

    vector<bool> sptSet(V, false);

    dist[src] = 0;

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

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;

        #pragma omp parallel for

        for (int v = 0; v < V; v++) {

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX) {

                int new\_dist = dist[u] + graph[u][v];

                // Atomic update to avoid race conditions

                #pragma omp critical

                {

                    if (new\_dist < dist[v]) {

                        dist[v] = new\_dist;

                    }

                }

            }

        }

    }

    cout << "Vertex \t Distance from Source" << endl;

    for (int i = 0; i < V; i++) {

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

    }

}

int main() {

    int graph[V][V] = {

        {0, 10, 0, 30, 0, 0},

        {10, 0, 50, 0, 0, 0},

        {0, 50, 0, 20, 10, 0},

        {30, 0, 20, 0, 60, 0},

        {0, 0, 10, 60, 0, 40},

        {0, 0, 0, 0, 40, 0}

    };

    int src = 0;

    dijkstra(graph, src);

    return 0;

}

**Histogram Sorting**

**Code:**

#include <iostream>

#include <vector>

#include <omp.h>

void histogram\_sort(std::vector<int>& data, int range) {

    std::vector<int> histogram(range, 0);

    // Parallel histogram computation

    #pragma omp parallel for

    for (size\_t i = 0; i < data.size(); i++) {

        #pragma omp atomic

        histogram[data[i]]++;

    }

    // Reconstruct sorted data from histogram

    size\_t index = 0;

    for (int value = 0; value < range; value++) {

        for (int count = 0; count < histogram[value]; count++) {

            data[index++] = value;

        }

    }

}

int main() {

    std::vector<int> data = {5, 3, 8, 3, 1, 2, 2, 5, 8, 8, 1, 3};

    int range = 10; // Assume elements are in the range [0, 9]

    histogram\_sort(data, range);

    std::cout << "Sorted Data: ";

    for (int num : data) {

        std::cout << num << " ";

    }

    std::cout << std::endl;

    return 0;

}