**Parallel Computing Practical File**

**Hemant Kr Singh (4005)**

**22058570006**

**Practical List for the Introduction to Parallel Programming Paper (DSC18)**

1. **Implement matrix-matrix multiplication in parallel using OpenMP**

#include <iostream>

#include <vector>

#include <omp.h>

#include <chrono>

using namespace std;

using namespace chrono;

#define N 3 // Define matrix size (NxN)

// Function to perform serial matrix multiplication

void serialMatrixMultiply(const vector<vector<int>> &A, const vector<vector<int>> &B, vector<vector<int>> &C)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            C[i][j] = 0;

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

            {

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

            }

        }

    }

}

// Function to perform parallel matrix multiplication

void parallelMatrixMultiply(const vector<vector<int>> &A, const vector<vector<int>> &B, vector<vector<int>> &C)

{

#pragma omp parallel for collapse(2)

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            C[i][j] = 0;

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

            {

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

            }

        }

    }

}

int main()

{

    vector<vector<int>> A = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    vector<vector<int>> B = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

    vector<vector<int>> C\_serial(N, vector<int>(N, 0));

    vector<vector<int>> C\_parallel(N, vector<int>(N, 0));

    // Measure time for serial matrix multiplication

    auto start\_serial = high\_resolution\_clock::now();

    serialMatrixMultiply(A, B, C\_serial);

    auto stop\_serial = high\_resolution\_clock::now();

    auto duration\_serial = duration\_cast<microseconds>(stop\_serial - start\_serial);

    // Measure time for parallel matrix multiplication

    auto start\_parallel = high\_resolution\_clock::now();

    parallelMatrixMultiply(A, B, C\_parallel);

    auto stop\_parallel = high\_resolution\_clock::now();

    auto duration\_parallel = duration\_cast<microseconds>(stop\_parallel - start\_parallel);

    // Print the result matrix

    cout << "Resultant Matrix C (Parallel):" << endl;

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            cout << C\_parallel[i][j] << " ";

        }

        cout << endl;

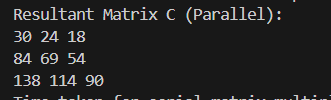
    }

    cout << "Time taken for serial matrix multiplication: " << duration\_serial.count() << " microseconds" << endl;

    cout << "Time taken for parallel matrix multiplication: " << duration\_parallel.count() << " microseconds" << endl;

    return 0;

}



**2. Implement distributed histogram sorting in parallel using OpenMP**

#include <iostream>

#include <vector>

#include <algorithm>

#include <omp.h>

using namespace std;

void parallel\_histogram\_sort(vector<int>& data, int num\_bins, int max\_value) {

    int n = data.size();

    vector<vector<int>> bins(num\_bins);

    int bin\_size = (max\_value + 1 + num\_bins - 1) / num\_bins;  // ceiling division

    // Parallel histogram binning

    #pragma omp parallel

    {

        vector<vector<int>> local\_bins(num\_bins);

        #pragma omp for nowait

        for (int i = 0; i < n; ++i) {

            int bin\_index = data[i] / bin\_size;

            if (bin\_index >= num\_bins) bin\_index = num\_bins - 1;

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

        }

        // Merge local bins into global bins

        #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());

            }

        }

    }

    // Sort each bin in parallel

    #pragma omp parallel for

    for (int i = 0; i < num\_bins; ++i) {

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

    }

    // Merge bins into final sorted data

    int index = 0;

    for (int i = 0; i < num\_bins; ++i) {

        for (int val : bins[i]) {

            data[index++] = val;

        }

    }

}

int main() {

    vector<int> data = {10, 7, 2, 30, 20, 15, 3, 5, 40, 25, 1, 8};

    int max\_value = 50;

    int num\_bins = 4;

    cout << "Original data: ";

    for (int val : data) cout << val << " ";

    cout << "\n";

    parallel\_histogram\_sort(data, num\_bins, max\_value);

    cout << "Sorted data:   ";

    for (int val : data) cout << val << " ";

    cout << "\n";

    return 0;

}

****

**3. Implement breadth first search in parallel using OpenMP**

#include <iostream>

#include <queue>

#include <vector>

#include <omp.h>

using namespace std;

void addEdge(vector<vector<int>> &adj, int u, int v)

{

    adj[u].push\_back(v);

    adj[v].push\_back(u);

}

void bfs(vector<vector<int>> &adj, int s)

{

    queue<int> q;

    vector<bool> visited(adj.size(), false);

    visited[s] = true;

    q.push(s);

    while (!q.empty()){

        int curr = q.front();

        q.pop();

        cout << curr << " ";

        #pragma omp parallel for

        for (int i = 0; i < adj[curr].size(); i++){

            int x = adj[curr][i];

            if (!visited[x]){

                #pragma omp critical

                {

                    if (!visited[x]){

                        visited[x] = true;

                        q.push(x);

                    }

                }

            }

        }

    }

}

int main()

{

    int V = 5;

    vector<vector<int>> adj(V);

    addEdge(adj, 0, 1);

    addEdge(adj, 0, 2);

    addEdge(adj, 2, 3);

    addEdge(adj, 1, 4);

    addEdge(adj, 2, 4);

    cout << "BFS starting from 0 : \n";

    bfs(adj, 0);

    return 0;

}



**4. Implement Dijkstra’s algorithm in parallel using OpenMP**

#include <iostream>

#include <vector>

#include <climits>

#include <omp.h>

#include <chrono>

using namespace std;

void dijkstra(const vector<vector<int>>& graph, int source, int numVertices) {

    // Create a vector to store distances from source

    vector<int> dist(numVertices, INT\_MAX);

    dist[source] = 0;

    // Create a visited set to keep track of vertices whose shortest distance is known

    vector<bool> visited(numVertices, false);

    // OpenMP parallel section to run the main loop

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

        int u = -1;

        int minDist = INT\_MAX;

        // Find the vertex u with the smallest distance value

        #pragma omp parallel for shared(dist, visited) reduction(min: minDist)

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

            if (!visited[i] && dist[i] < minDist) {

                #pragma omp critical

                {

                    if (dist[i] < minDist) {

                        minDist = dist[i];

                        u = i;

                    }

                }

            }

        }

        if (u == -1) break;  // All vertices visited or unreachable

        visited[u] = true; // Mark vertex u as visited

        // Relax the edges for the vertex u

        #pragma omp parallel for

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

            if (!visited[v] && graph[u][v] != 0 && dist[u] != INT\_MAX) {

                #pragma omp critical

                {

                    if (dist[u] + graph[u][v] < dist[v]) {

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

                    }

                }

            }

        }

    }

    // Output the computed shortest distances

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

        if (dist[i] == INT\_MAX) {

            cout << "Vertex " << i << " is unreachable from source " << source << endl;

        } else {

            cout << "Distance from source " << source << " to vertex " << i << " is " << dist[i] << endl;

        }

    }

}

int main() {

    // Number of vertices

    int numVertices = 6;

    // Graph represented as an adjacency matrix (0 means no edge between vertices)

    vector<vector<int>> graph = {

        {0, 7, 9, 0, 0, 14},

        {7, 0, 10, 15, 0, 0},

        {9, 10, 0, 11, 0, 2},

        {0, 15, 11, 0, 6, 0},

        {0, 0, 0, 6, 0, 9},

        {14, 0, 2, 0, 9, 0}

    };

    // Source vertex

    int source = 0;

    auto start\_parallel = chrono::high\_resolution\_clock::now();

    // Call Dijkstra's algorithm

    dijkstra(graph, source, numVertices);

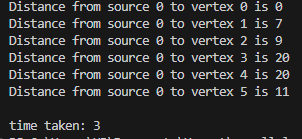
    auto end\_parallel = chrono::high\_resolution\_clock::now();

    auto duration = chrono::duration\_cast<chrono::milliseconds>(end\_parallel - start\_parallel);

    cout<<endl<<"time taken: "<<duration.count()<<" milliseconds";

    return 0;

}



**5. Write a c++ program to implement calculate the sum of the first n natural numbers using parallel programming.**

#include <iostream>

#include <omp.h>

using namespace std;

// Function to calculate the sum of first n natural numbers using parallel programming

long long parallelSum(int n)

{

    long long sum = 0;

#pragma omp parallel for reduction(+ : sum)

    for (int i = 1; i <= n; i++)

    {

        sum += i;

    }

    return sum;

}

int main()

{

    int n;

    cout << "Enter the value of n: ";

    cin >> n;

    long long result = parallelSum(n);

    cout << "Sum of first " << n << " natural numbers is: " << result << endl;

    return 0;

}

