Introduction to Parallel Programming

Practical File

Shashwat Sharma

22058570029

B.Sc. (H) Computer Science

6th Sem

1- Matrix Multiplication

Code:

```
#include <iostream>
#include <vector>
#include <chrono>
#include <omp.h>

using namespace std;
const int N = 1000;

int main(){
    vector<vector<int>> A(N, vector<int>(N));
    vector<vector<int>> B(N, vector<int>(N));
    vector<vector<int>> C(N, vector<int>(N));

    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            A[i][j] = rand() % 100;
            B[i][j] = rand() % 100;
        }
    }

    auto start_serial = chrono::high_resolution_clock::now();
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            int sum = 0;
            for(int k = 0; k < N; k++){
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    auto end_serial = chrono::high_resolution_clock::now();
    auto duration_serial =
chrono::duration_cast<chrono::milliseconds>(end_serial - start_serial);

    auto start_parallel = chrono::high_resolution_clock::now();
    #pragma omp parallel for
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            int sum = 0;
            for(int k = 0; k < N; k++){
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    auto end_parallel = chrono::high_resolution_clock::now();
    auto duration_parallel =
chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);
```

```
    cout << "Time taken for serial matrix multiplication: " <<
duration_serial.count() << " milliseconds" << endl;
    cout << "Time taken for parallel matrix multiplication: " <<
duration_parallel.count() << " milliseconds" << endl;
}
```

Output:

```
PS C:\Users\HP\Desktop\himanshu\project> cd "c:\Users\HP\Desktop\
Time taken for serial matrix multiplication: 4789 milliseconds
Time taken for parallel matrix multiplication: 4816 milliseconds
```

2- Sum of first N natural numbers

Code:

```cpp
#include <iostream>
#include <chrono>
#include <omp.h>

using namespace std;

int main(){
    int N;
    cout << "Enter a number: ";
    cin >> N;

    auto start_serial = chrono::high_resolution_clock::now();
    int sum_serial = 0;
    for(int i = 1; i < N + 1; i++){
        sum_serial += i;
    }
    auto end_serial = chrono::high_resolution_clock::now();
    auto duration_serial =
chrono::duration_cast<chrono::milliseconds>(end_serial - start_serial);

    auto start_parallel = chrono::high_resolution_clock::now();
    int sum_parallel = 0;
    for(int i = 1; i < N + 1; i++){
        #pragma omp parallel for
        for(int j = 1; j < N + 1; j++){
            sum_parallel += i;
        }
    }
    auto end_parallel = chrono::high_resolution_clock::now();
    auto duration_parallel =
chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);

    cout << "Time taken for serial matrix multiplication: " <<
duration_serial.count() << " milliseconds" << endl;
    cout << "Time taken for parallel matrix multiplication: " <<
duration_parallel.count() << " milliseconds" << endl;
}
```

Output:

```
PS C:\Users\HP\Desktop\Ankit CS\IPP\Practicals> cd "c:\Users\HP\L
Enter a number: 10000
Time taken for serial matrix multiplication: 0 milliseconds
Time taken for parallel matrix multiplication: 166 milliseconds
```

3- BFS

Code:

```
#include <iostream>
#include <queue>
#include <vector>
#include <omp.h>
#include <chrono>

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_parallel(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 traversal of graph starting from 0 : \n";

    auto start_parallel = chrono::high_resolution_clock::now();
    bfs_parallel(adj, 0);
    auto end_parallel = chrono::high_resolution_clock::now();

    auto duration_parallel =
chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);
    cout << endl << "Time taken for parallel BFS: " <<
duration_parallel.count() << " milliseconds" << endl;

    return 0;
}
```

Output:

```
BFS traversal of graph starting from 0 :
0 1 2 4 3
Time taken for parallel BFS: 1 milliseconds
```

4- Dijkstra’s Algorithm

Code:

```
#include <iostream>
#include <omp.h>
#include <limits.h>
#include <chrono>

#define V 6
#define INF INT_MAX

using namespace std;
using namespace chrono;

int min_distance(int dist[], int sptSet[], int n) {
    int min = INF;
    int min_index = -1;

    #pragma omp parallel for
    for (int v = 0; v < n; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            #pragma omp critical
            {
                if (dist[v] < min) {
                    min = dist[v];
                    min_index = v;
                }
            }
        }
    }

    return min_index;
}

void dijkstra(int graph[V][V], int start) {
    int dist[V];
    int sptSet[V];

    #pragma omp parallel for
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        sptSet[i] = 0;
    }
    dist[start] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = min_distance(dist, sptSet, V);

        sptSet[u] = 1;

        #pragma omp parallel for
```

```
        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    cout << "Vertex \t Distance from Source\n";
    for (int i = 0; i < V; i++) {
        cout << i << " \t " << dist[i] << endl;
    }
}
int main() {
    int graph[V][V] = {
        {0, 9, 6, 0, 0, 0},
        {9, 0, 0, 5, 0, 0},
        {6, 0, 0, 3, 0, 0},
        {0, 5, 3, 0, 4, 2},
        {0, 0, 0, 4, 0, 7},
        {0, 0, 0, 2, 7, 0}
    };

    auto start_parallel = chrono::high_resolution_clock::now();
    dijkstra(graph, 0);
    auto end_parallel = chrono::high_resolution_clock::now();

    auto duration_parallel =
chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);
    cout << endl << "Time taken for parallel Dijkstra: " <<
duration_parallel.count() << " milliseconds" << endl;
    return 0;
}
```

Output:

```
Vertex   Distance from Source
0        0
1        9
2        6
3        9
4        13
5        11

Time taken for parallel Dijkstra: 2 milliseconds
```

5- Distributed Histogram Sort

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <omp.h>
#include <chrono>

using namespace std;

void distributed_HistSort(std::vector<int>& arr, int num_bins) {
    int n = arr.size();

    int min_val = *std::min_element(arr.begin(), arr.end());
    int max_val = *std::max_element(arr.begin(), arr.end());
    int range = max_val - min_val + 1;

    int bin_size = range / num_bins;

    vector<vector<int>> bins(num_bins);

    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        int bin_index = (arr[i] - min_val) / bin_size;
        if (bin_index == num_bins) {
            bin_index--;
        }
        #pragma omp critical
        bins[bin_index].push_back(arr[i]);
    }

    #pragma omp parallel for
    for (int i = 0; i < num_bins; ++i) {
        sort(bins[i].begin(), bins[i].end());
    }

    int index = 0;
    for (int i = 0; i < num_bins; ++i) {
        for (int j = 0; j < bins[i].size(); ++j) {
            arr[index++] = bins[i][j];
        }
    }
}

int main() {
    vector<int> arr = {15, 3, 9, 27, 6, 18, 22, 14, 1, 30, 10, 5, 25, 2, 13,
21, 11, 28, 7, 20, 23, 17, 12, 19, 8, 24, 29, 4, 16, 26};

    int num_bins = 3;
```

```
    cout << "Original array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    auto start_parallel = chrono::high_resolution_clock::now();
    distributed_HistSort(arr, num_bins);

    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    auto end_parallel = chrono::high_resolution_clock::now();

    auto duration_parallel =
chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);
    cout << endl << "Time taken for parallel Dijkstra: " <<
duration_parallel.count() << " milliseconds" << endl;

    return 0;
}
```

Output:

```
Original array: 15 3 9 27 6 18 22 14 1 30 10 5 25 2 13 21 11 28 7 20 23 17 12 19 8 24 29 4 16 26
Sorted array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Time taken for parallel Dijkstra: 1 milliseconds
```