

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

1. Implement matrix-matrix multiplication in parallel using OpenMP

```
main.cpp  [Icons] [Share] [Run]

1  #include <iostream>
2  #include <vector>
3  #include <chrono>
4  #include <omp.h>
5
6  using namespace std;
7  const int N = 1000;
8  int main()
9  {
10 vector<vector<int>> A(N, vector<int>(N));
11 vector<vector<int>> B(N, vector<int>(N));
12 vector<vector<int>> C(N, vector<int>(N));
13
14 // Initialize matrices A and B with random values
15 for (int i = 0; i < N; i++) {
16     for (int j = 0; j < N; j++) {
17         A[i][j] = rand() % 100;
18         B[i][j] = rand() % 100;
19     }
20 }
21 auto start_serial = chrono::high_resolution_clock::now();
22 for (int i = 0; i < N; i++) {
23     for (int j = 0; j < N; j++) {
24         int sum = 0;
25         for (int k = 0; k < N; k++) {
```

```
21 auto start_serial = chrono::high_resolution_clock::now();
22 for (int i = 0; i < N; i++) {
23     for (int j = 0; j < N; j++) {
24         int sum = 0;
25         for (int k = 0; k < N; k++) {
26             sum += A[i][k] * B[k][j];
27         }
28         C[i][j] = sum;
29     }
30 }
31 auto end_serial = chrono::high_resolution_clock::now();
32 auto duration_serial = chrono::duration_cast<chrono::milliseconds>
    >(end_serial - start_serial);
33
34
35 // Perform matrix multiplication in parallel using OpenMP
36 auto start_parallel = chrono::high_resolution_clock::now();
37 #pragma omp parallel for
38 for (int i = 0; i < N; i++) {
39     for (int j = 0; j < N; j++) {
40         int sum = 0;
41         for (int k = 0; k < N; k++) {
42             sum += A[i][k] * B[k][j];
43         }
44         C[i][j] = sum;
```

main.cpp



Share

Run

```
33
34
35 // Perform matrix multiplication in parallel using OpenMP
36 auto start_parallel = chrono::high_resolution_clock::now();
37 #pragma omp parallel for
38 for (int i = 0; i < N; i++) {
39     for (int j = 0; j < N; j++) {
40         int sum = 0;
41         for (int k = 0; k < N; k++) {
42             sum += A[i][k] * B[k][j];
43         }
44         C[i][j] = sum;
45     }
46 }
47 auto end_parallel = chrono::high_resolution_clock::now();
48 auto duration_parallel = chrono::duration_cast<chrono::milliseconds>
    >(end_parallel - start_parallel);
49
50 // Display the time taken for each approach
51 cout << "Time taken for serial matrix multiplication: " << duration_serial
    .count() << " milliseconds" << endl;
52 cout << "Time taken for parallel matrix multiplication: " <<
    duration_parallel.count() << " milliseconds" << endl;
53 return 0;
54
```

Output

Clear

Time taken for serial matrix multiplication: 10807 milliseconds
Time taken for parallel matrix multiplication: 10475 milliseconds

=== Code Execution Successful ===

2. Implement distributed histogram sorting in parallel using OpenMP

main.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <omp.h>
5
6  #define NUM_BINS 10 // Number of histogram bins
7  #define NUM_THREADS 4 // Number of OpenMP threads
8
9  // Function to compute the bin index for a given value
10 int getBinIndex(int value, int minValue, int maxValue) {
11     return (NUM_BINS * (value - minValue)) / (maxValue - minValue + 1);
12 }
13
14 // Parallel Histogram Sort
15 void histogramSort(std::vector<int>& arr) {
16     int n = arr.size();
17     if (n == 0) return;
18
19     int minValue = *std::min_element(arr.begin(), arr.end());
20     int maxValue = *std::max_element(arr.begin(), arr.end());
21
22     // Step 1: Compute histogram
23     std::vector<int> histogram(NUM_BINS, 0);
24     #pragma omp parallel for num_threads(NUM_THREADS) reduction(+:histogram[
```

```

25     for (int i = 0; i < n; i++) {
26         int binIndex = getBinIndex(arr[i], minValue, maxValue);
27         histogram[binIndex]++;
28     }
29
30     // Step 2: Compute prefix sum (cumulative sum)
31     std::vector<int> prefixSum(NUM_BINS, 0);
32     prefixSum[0] = histogram[0];
33     for (int i = 1; i < NUM_BINS; i++) {
34         prefixSum[i] = prefixSum[i - 1] + histogram[i];
35     }
36
37     // Step 3: Distribute elements into bins
38     std::vector<std::vector<int>> bins(NUM_BINS);
39     #pragma omp parallel for num_threads(NUM_THREADS)
40     for (int i = 0; i < n; i++) {
41         int binIndex = getBinIndex(arr[i], minValue, maxValue);
42         #pragma omp critical
43         bins[binIndex].push_back(arr[i]);
44     }
45
46     // Step 4: Sort each bin in parallel
47     #pragma omp parallel for num_threads(NUM_THREADS)
48     for (int i = 0; i < NUM_BINS; i++) {
49         // Sort the elements in the bin
50         std::sort(bins[i].begin(), bins[i].end());
51     }
52
53     // Step 5: Merge the sorted bins
54     std::vector<int> sortedArr(n);
55     int index = 0;
56     for (int i = 0; i < NUM_BINS; i++) {
57         for (int j = 0; j < bins[i].size(); j++) {
58             sortedArr[index++] = bins[i][j];
59         }
60     }
61
62     return sortedArr;
63 }

```

main.cpp

```
52 // Step 5: Merge sorted bins back to the original array
53 int index = 0;
54 for (int i = 0; i < NUM_BINS; i++) {
55     for (int val : bins[i]) {
56         arr[index++] = val;
57     }
58 }
59 }
60
61 int main() {
62     std::vector<int> arr = {23, 45, 12, 89, 5, 34, 78, 11, 90, 67, 55, 32,
63                             43, 21};
64
65     std::cout << "Original array: ";
66     for (int num : arr) std::cout << num << " ";
67     std::cout << "\n";
68
69     histogramSort(arr);
70
71     std::cout << "Sorted array: ";
72     for (int num : arr) std::cout << num << " ";
73     std::cout << "\n";
74
75     return 0;
76 }
```

Output

Clear

Original array: 23 45 12 89 5 34 78 11 90 67 55 32 43 21
Sorted array: 5 11 12 21 23 32 34 43 45 55 67 78 89 90

=== Code Execution Successful ===

3. Implement breadth first search in parallel using OpenMP

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <omp.h>
5
6  using namespace std;
7
8  class Graph {
9      int V; // Number of vertices
10     vector<vector<int>> adj; // Adjacency list
11
12 public:
13     Graph(int vertices) {
14         V = vertices;
15         adj.resize(vertices);
16     }
17
18     void addEdge(int u, int v) {
19         if (u >= 0 && u < V && v >= 0 && v < V) {
20             adj[u].push_back(v);
21             adj[v].push_back(u); // For undirected graph
22         } else {
23             cout << "Invalid edge! Vertex out of range." << endl;
24         }
25     }
26 }
```

```

27 void parallelBFS(int start) {
28     if (start < 0 || start >= V) {
29         cout << "Invalid start node!" << endl;
30         return;
31     }
32
33     vector<bool> visited(V, false);
34     queue<int> q;
35
36     visited[start] = true;
37     q.push(start);
38
39     cout << "Parallel BFS Traversal: ";
40
41     while (!q.empty()) {
42         int level_size = q.size();
43         vector<int> level_nodes;
44
45         for (int i = 0; i < level_size; i++) {
46             int node = q.front();
47             q.pop();
48             cout << node << " ";
49             level_nodes.push_back(node);
50         }

```



```

52 // Parallel processing of neighbors at the current level
53 #pragma omp parallel for
54 for (int i = 0; i < level_nodes.size(); i++) {
55     int node = level_nodes[i];
56     for (int j = 0; j < adj[node].size(); j++) {
57         int neighbor = adj[node][j];
58         if (!visited[neighbor]) {
59             #pragma omp critical
60             {
61                 if (!visited[neighbor]) { // Double-check inside
62                                         critical section
63                                         visited[neighbor] = true;
64                                         q.push(neighbor);
65                                     }
66                                 }
67         }
68     }
69 }
70 cout << endl;
71 }
72 };
73
74 int main() {
75     int V, E;

```

```

73
74 int main() {
75     int V, E;
76     cout << "Enter the number of vertices: ";
77     cin >> V;
78
79     Graph g(V);
80
81     cout << "Enter the number of edges: ";
82     cin >> E;
83
84     cout << "Enter " << E << " edges (u v) where 0 <= u, v < " << V << ":"
        << endl;
85     for (int i = 0; i < E; i++) {
86         int u, v;
87         cin >> u >> v;
88         g.addEdge(u, v);
89     }
90
91     int startNode;
92     cout << "Enter the starting node for BFS: ";
93     cin >> startNode;
94
95     g.parallelBFS(startNode);
96

```

Output

```
Enter the number of vertices:
6
Enter the number of edges: 8
Enter 8 edges (u v) where 0 <= u, v < 6:
0 1
0 2
1 3
1 4
2 4
3 5
4 5
2 3
Enter the starting node for BFS: 0
Parallel BFS Traversal: 0 1 2 3 4 5
```

```
=== Code Execution Successful ===
```

4. Implement Dijkstra's alg

```
main.cpp  [Icons] [Share] [Run]

1  #include <iostream>
2  #include <vector>
3  #include <limits>
4  #include <omp.h>
5
6  using namespace std;
7
8  #define INF numeric_limits<int>::max()
9
10 // Function to find the vertex with the minimum distance value
11 int minDistance(vector<int>& dist, vector<bool>& sptSet, int V) {
12     int min = INF, min_index = -1;
13
14     #pragma omp parallel for
15     for (int v = 0; v < V; v++) {
16         if (!sptSet[v] && dist[v] <= min) {
17             #pragma omp critical
18             {
19                 if (dist[v] < min) {
20                     min = dist[v];
21                     min_index = v;
22                 }
23             }
24         }
25     }
```

orithm in parallel using OpenMP

main.cpp



Share

Run

29 // Dijkstra's Algorithm using OpenMP for parallelism

30 void dijkstra(vector<vector<int>>& graph, int src, int V) {

31 vector<int> dist(V, INF); // Shortest distance array

32 vector<bool> sptSet(V, false); // True if vertex is included in shortest
path tree

33

34 dist[src] = 0;

35

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

37 int u = minDistance(dist, sptSet, V);

38 if (u == -1) break; // If no minimum found, stop

39

40 sptSet[u] = true;

41

42 #pragma omp parallel for

43 for (int v = 0; v < V; v++) {

44 if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v]) {

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

46

47 }

48

49

50 // Display the shortest distances

51 cout << "Vertex \t Distance from Source\n";

```

53     cout << i << " \t " << (dist[i] == INF ? -1 : dist[i]) << endl;
54 }
55
56 int main() {
57     int V, E;
58     cout << "Enter number of vertices: ";
59     cin >> V;
60     cout << "Enter number of edges: ";
61     cin >> E;
62
63     vector<vector<int>> graph(V, vector<int>(V, 0));
64
65     cout << "Enter edges (source, destination, weight):\n";
66     for (int i = 0; i < E; i++) {
67         int u, v, w;
68         cin >> u >> v >> w;
69         graph[u][v] = w;
70         graph[v][u] = w; // Assuming an undirected graph
71     }
72
73     int src;
74     cout << "Enter source vertex: ";
75     cin >> src;
76
77     dijkstra(graph, src, V);

```

Output

```
Enter number of vertices: 5
Enter number of edges: 7
Enter edges (source, destination, weight):
0 1 4
0 2 2
1 2 1
1 3 5
2 3 8
2 4 10
3 4 2
Enter source vertex: 0
Vertex    Distance from Source
0         0
1         3
2         2
3         8
4        10
```

```
=== Code Execution Successful ===
```