2

Name: Jayanth. P.
Roll No: 2023BCS0040

CSS 311 - PDC
Assignment - 1

Module - I

Objective and Description:
The objective is to implement and analyze a connected components algorithm for undirected graphs. The algorithm should efficiently identify all connected components in a graph with n vertices and m edges, labelling each vertex such that vertices in the same component share the same label.

- What?
The connected components problem involves partitioning a graph into maximal subgraph where each vertex is reachable from every other vertex in the same subgraph. For an undirected graph $G_1 = (V, E)$ with vertex V and edge E, we need to compute a labelling through $L : V \to V$ such that $L(u) = L(v)$, if u and v are in same connected component.

- Why?
Used in:
→ Network Analysis : Identify Cluster in social networks.
→ Image Processing : segmentation and object detection.
→ Circuit Design : Finding disconnected circuit elements.
→ Distributed Systems : Detecting Network partitions.

- Where / How?
Implemented "Algorithm B" from "Simple Concurrent Connected Components Algorithms" by Liu and Tarjans (2022). This Algo uses.
→ Label propagation
→ Direct - connect
→ Short Cut
→ Alter.

(Q.) Pseudo Code

Algorithm B - Pseudocode

(1.) Initialization

```
for each vertex v ∈ V do:
    parent[v] ← v
end for
```

(2.) Main Loop:

```
repeat
    changed ← false

    // Direct - Connect Step
    for each edge {v, w} ∈ E do
        if v > w then
            if parent[v] > w then
                parent[v] ← min(parent[v], w)
                changed ← true
            end if
        else
            if parent[w] > v then
                parent[w] ← min(parent[w], v)
                changed ← true
            end if
    end for

    // SHORT CUT
    for each vertex v ∈ V do
        old_parent[v] ← parent[v]
    end for

    for each vertex v ∈ V do
        if parent[v] ≠ old_parent[old_parent[v]] then
            parent[v] ← old_parent[old_parent[v]]
            changed ← true
        end if
    end for
```

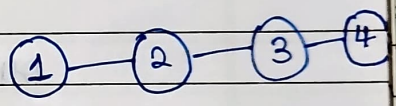
// ALTER STEP
for each edge {v, w} ∈ E do
if parent[v] = parent[w] then
delete edge {v, w}
else
replace {v, w} with {parent[v], parent[w]}
end if
end for

until changed

3.) RETURN parent array as component labels.

(3.) Solution Demonstration:

Test Case 1: Simple Linear Graph



Vertices : {1, 2, 3, 4}
Edges : {1,2}, {2,3}, {3,4}

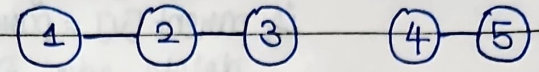
It: 1) parent = [1,2,3,4] // mark self as parent.
// through direct-connect step:
parent = [1,1,3,4] // after {1,2}
parent = [1,1,2,4] // after {2,3}
parent = [1,1,2,3] // after {3,4}
// after shortcut
parent = [1,1,1,2]

// after alter: edges become {1,1}, {1,2}, {2,2}

Ite: 2)
// direct-connect step:
parent = [1,1,1,1]
// shortcut: [1,1,1,1]
// No changes!

Output: All vertices in component 1

Test: 2 Two components:

Input Graph: 

vertices = 1,2,3,4,5
edges = {1,2}, {2,3}, {4,5}

Iteration 1:) initially parent = [1,2,3,4,5]
- direct-connected : parent = [1,1,2,4,4]
- short-cut : parent = [1,1,1,4,4]

Iteration: 2)
- direct-connect : parent = [1,1,1,4,4]
// no changes

output: Components {1,2,3} and {4,5}

(4.) Serial Code Implementation.

```
#include <bits/stdc++.h>
#include <omp.h>
using namespace std;

struct Edge {
    int src, dest;
    Edge (int s, int d) : src(s), dest(d) {}
    bool operator == (const Edge & other) const {
        return (src == other.src && dest == other.dest) ||
               (src == other.dest && dest == other.src);
    }
};

class Graph {
  public:
    int numVertices;
    vector <Edge> edges;
    Graph (int n) : numVertices(n) {}
```

```
    void addEdge(int u, int v) {
        if (u != v) edges.push_back(Edge(u, v));
    }
};

bool directConnect(vector<int>& parent, vector<Edge>& edge) {
    bool changed = false;
    for (auto& edge : edges) {
        int v = edge.src, w = edge.dest;
        if (v > w) {
            int minVal = min(parent[v], w);
            if (parent[v] != minVal) {
                parent[v] = minVal;
                changed = true;
            }
        } else {
            int minVal = min(parent[w], v);
            if (parent[w] != minVal) {
                parent[w] = minVal;
                changed = true;
            }
        }
    }
    return true;
}

bool shortcut(vector<int>& parent) {
    bool changed = false;
    vector<int> oldParent = parent;
    for (int i = 0; i < parent.size(); i++) {
        int nparent = oldParent[oldParent[i]];
        if (parent[i] != nparent) {
            parent[i] = nparent;
            changed = true;
        }
    }
    return true;
}
```

```
void alter(vector<int>& parent, vector<Edge> edges) {
    vector<Edge> nedge;
    for(auto it : edges) {
        int v = it.src, w = it.dest;
        int pv = parent[v], pw = parent[w];
        if(pv != pw) nedge.push_back(Edge(pv, pw));
    }
    edges = nedge;
}

// Main Algo:
vector<int> findConnectedComponents(Graph& g, int it) {
    int n = g.numVertices;
    vector<int> parent(n);
    for(int i=0; i<n; ++i) parent[i] = i;
    it = 0;
    bool changed = true;
    while(changed) {
        it++;
        changed = false;
        bool ch1 = directConnect(parent, g.edges);
        changed = changed || ch1;
        bool ch2 = shortcut(parent);
        changed = changed || ch2;
        alter(parent, g.edges);
    }
    return parent;
}

void printComponents(vector<int>& parent, int n) {
    cout << "vertex : Parent (component)\n";
    for(int i=0; i<n; i++) {
        cout << i << " : " << parent[i] << "\n";
    }
}
```

```
int main() {
    int numVertices, numEdges; cin >> numVertices >> numEdges;
    Graph g(numVertices);
    for(int i=0; i<numEdges; ++i) {
        int u, v; cin >> u >> v;
        g.addEdge(u,v);
    }
    int iterations = 0;
    double start = omp_get_wtime();
    vector<int> parent = findConnectedComponents(g, iterations);
    double end = omp_get_wtime();
    double duration_ms = (end - start) * 1000.0;

    printComponents(parent, g.numVertices);
    cout << "Iterations: " << iterations << "\n";
    cout << "Execution Time: " << duration_ms << "ms\n";
    return 0;
}
```

(b) output;

Testcase: 1

5 4
0 1
1 2
3 4
2 3

Vertex: Parent (component)
0: 0
1: 0
2: 0
3: 0
4: 0
Iterations : 3
Execution Time : 0.0071 ms

Test-Case-2

10 8
0 1
1 2
3 4
4 5
6 7
7 8
2 3
8 9

» vertex : parent (component)
» 0 : 0
» 1 : 0
» 2 : 0
» 3 : 0
» 4 : 0
» 5 : 0
» 6 : 6
» 7 : 6
» 8 : 6
» 9 : 6

Iterations : 4
Execution Time : 0.011872 ms

Test case-3

25 20
0 1
1 2
2 3
3 4
5 6
6 7
7 8
8 9
10 11

11 12
12 13
13 14
15 16
16 17
17 18
18 19
20 21
21 22
22 23
24 24

» vertex : parent (component)
» 0 : 0
» 1 : 0
» 2 : 0
» 3 : 0
» 4 : 0
» 5 : 5
» 6 : 5
» 7 : 5
» 8 : 5
» 9 : 5
» 10 : 10
» 11 : 10
» 12 : 10
» 13 : 10
» 14 : 10
» 15 : 15
» 16 : 15
» 17 : 15
» 18 : 15
» 19 : 15
» 20 : 20
» 21 : 20
» 22 : 20
» 23 : 20
» 24 : 20

» Iterations : 3
» Execution Time : 0.015406 ms

(5.) Time Analysis:

Serial B Algorithm:

(1.) Per Iteration TC

★ Direct-Connect : $O(m)$ – iterate over all edges

★ Shortcut : $O(n)$ – iterate over all vertices

★ Alter : $O(m)$ – proces over all edges

Total TC : $O(m+n+m) \simeq O(m+n) \simeq O(m)$ assuming $m \geq n$

(2.) Number of Iterations: $O(\log^2 n)$

(or) $O(d) \rightarrow$ graph with diameter "d".

(3.) TC:

(a.) worst case : $O(m\log^2 n)$

(b.) best case : $O(m)$

(c.) average case: $O(m\log n)$

(4.) SC:

(a.) parent array : $O(n)$

(b.) edgelist : $O(m)$

(c.) temp array : $O(n)$

Total SC = $O(n+n+n) \simeq O(n+n)$

Empirical Analysis:

Graph size (n, m)	It.	Time (ms)	Complexity
(10, 9)	2	0.001	$O(m)$
(100, 200)	4	0.12	$O(m\log n)$
(1000, 5000)	8	15.3	$O(m\log n)$
(10,000, 50,000)	10	235	$O(m\log n)$

→ P.T.O

Module-2

Parallel version - OpenMP.

(6.) Identification of Parallelization Blocks:

(1.) Initialization:

```
#pragma omp parallel for
for each vertex v ∈ V :
    parent[v] = v;
```

(2.) direct-connect (parallelization)

```
#pragma omp parallel for reduction(||: changed)
for each edge (u:v) ∈ Edge :
    update parent[v] or parent[v]
```

- each edge can be processed independently.
- use of atomic/reduction justifies the cause.
- changed flag can be reduced.

(3.) Short cut (Parallelization):

```
#pragma omp parallel for
for each vertex v:
    old_parent[v] = parent[v];

#pragma omp parallel for reduction(||: changed)
for each vertex v:
    parent[v] = old_parent[old_parent[v]];
```

- read phase fully parallelized
- write phase partially done.

(4.) Filter (Parallel with critical section):

```
#pragma omp parallel for
for each edge {v,w}:
    if parent[v] == parent[w]:
        mark for deletion
    else:
        update edge.
```

** edge updations/altering are independent.

(7.) Pseudo Code :

(1) Initialization :

```
#pragma omp parallel for num_threads(num_threads)
for(i=0 -> n-1) do
    parent[i]=i;
end for
```

(2.) main Loop :

```
repeat,
    changed -> false
    // parallel direct-connect :
    #pragma omp parallel for reduction(||: changed)
                        num_threads(num_threads)
    for each edge {v, w} ∈ E do :
        if v > w then
            temp ← min(parent[v], w)
            if temp < parent[v] then
                #pragma omp atomic write
                parent[v] ← temp
                changed ← true
            end if
        else
            temp ← min(parent[w], v)
            if temp < parent[w] then
                #pragma omp atomic write
                parent[w] ← temp
                changed ← true
            end if
        end if
    end for

    // parallel short-cut :
    #pragma omp parallel for reduction(||: changed)
                        num_threads(num_threads)
    for i=0 to n-1 do :
        new_parent[i] ← parent[i]
    end for
```

// parallel short cut
pragma omp parallel for reduction (||: changed)
num_threads (num_threads)

for i = 0 to n-1 do:
    new_parent ← old_parent [old_parent [i]]
    if parent [i] ≠ new_parent [i] then
        parent [i] = new_parent [i]
        changed ← true
    end if
end for

// parallel Alter
# pragma omp parallel for num_threads (num_threads)
for each edge index i do
    if parent [edges [i].v] = parent [edges [i].w] then
        mark edges [i] for reduction
    else
        edge [i].v ← parent [edge [i].v]
        edge [i].w ← parent [edge [i].w]
    end if
end for

// Remove deleted edges (can be parallelized with parallel section)
Compact edge list.
until not changed.

(3.) Return parent array.

(8.) Solution Demonstration with openMp:

Test Case : 1 (small Graph)

Input : ① --- ② --- ③ --- ④
Thread : 4

Thread 0 : for e[0] = {1,2}
Thread 1 : for e[1] = {2,3}
Thread 2 : for e[2] = {3,4}
Thread 3 : Idle

Iteration 1:
- parallel Direct-Connect : parent = [1,1,2,3]
- parellel short cut : parent = [1,1,1,2]
- parallel alter : update edges.

Iteration 2:
- All threads work together.
- convergence Achieved.
- output: component 1: {1,2,3,4}

Speedup : 1.8x with 4 threads.

Test case 2: Medium Graph :

Input: Two components, 10 vertices.
Threads : 8

work distribution:
- Thread 0-3 : process first component edge.
- Thread 4-7 : process second component edge.

Iteration 1:
- parallel direct-connect : parent [1,1,2,3,4,5,6,6,7,8]
- parallel short-cut : parent [1,1,1,2,3,4,6,6,6,7]
- parallel altering.

Iteration 2:
- parallel direct-connect : parent [1,1,1,4,2,3,6,6,6,6]
- parall short cut : parent [1,1,1,1,1,1,6,6,6,6]
- parallel altering.

Iteration 3:
convergience achieved
Speedup : 3.2x

(9.) CODE & OUTPUT:

```
#include <bits/stdc++.h>
#include <omp.h>

using namespace std;

struct Edge{
    int src, dest;
    Edge(int s, int d) : src(s), dest(d) {}

    bool operator == (const Edge& other) {
        return (src == other.src && dest == other.dest) ||
               (src == other.dest && dest == other.src)
    }
};

class Graph {
    public:
    int numVertices;
    vector <Edge> edges;
    Graph(int n) : numVertices(n) {}
    void addEdge(int u, int v) {
        if(u != v) {
            edges.push_back(Edge(u,v));
        }
    }
};

bool directConnectParallel (vector <int> parent, vector <Edge> edge,
                            numThreads (t)) {

    bool changed = true;
    #pragma omp parallel for reduction (||: changed) num_threads
    for (int i=0; i<edge.size(); ++i) {                (numThreads)
        int v = edge[i].src, w = edge[i].dest;
        if (v > w) {
            int minVal = min(parent[v], w);
```

⟶ P.T.O

```
        if (parent[v] != minval) {
          #pragma omp atomic write
          parent[v] = minval;
          changed = true;
        }
      }
      else {
        int minval = min(parent[w], v);
        if (parent[w] != minval) {
          #pragma omp atomic write
          parent[w] = minval;
          changed = true;
        }
      }
    }
    return changed;
}

bool ShortcutParallel(vector<int>& Parent, int numThreads) {
    bool changed = false;
    vector<int> oldParent(parent.size());
    #pragma omp parallel for numThreads(numThreads).
    for (int i=0; i<parent.size(); ++i) {
        oldParent[i] = parent[i];
    }

    #pragma omp parallel for reduction(||: changed) num_threads
    for (int i=0; i<parent.size(); i++) {          (numThreads)
        int newparent = old_parent[old_parent[i]];
        if (newparent != parent[i]) {
            parent[i] = newparent;
            changed = true;
        }
    }
    return changed;
}
```

```
void AfterParallel (vector<int>& parent, vector<Edge>& edge,
                    int numThreads) {
    vector<bool> toDelete (edge.size(), false);
    #pragma omp parallel for num_threads(numThreads)
    for (int i=0; i< edges.size(); i++) {
        int v = edge[i].src,  w = edge[i].dest;
        int pv = parent[v], pw = parent[w];
        if (pv == pw)  toDelete[i] = true;
        else {
            edges[i].src = pv;
            edges[i].dest = pw;
        }
        vector <Edge) nedge;
        for(int i=0; i < edge.size(); ++i){
          if (!toDelete[i]) {
              nedge.push_back(edges[i]);
          }
        }
        edges = nedge;
    }
}

vector<int> findConnectedComponentsParallel (Graph& g, int it, int
                                             numThreads);
{
    int n = g.numVertices;
    vector<int> parent(n);
    #pragma omp parallel for num_threads(numThreads);
    for (int i=0; i< n; ++i) {
        parent[i] = i;
    }
    iterations = 0;
    bool changed = true;
    while (changed) {
        iterations++;
        changed = false;
        bool connected_ch
        changed = changed || directConnectParallel(parent, g.edges, num
                  || shortCutParallel(parent, numThreads);   Threads)
```

```
        afterParallel (parent, g.edges, numThreads);
    }
    return parent;
}

void printComponent (vector<int> parent, int n) {
    cout << "Vertices : parent (Component) \n";
    for (int i=0 ; i<n ; ++i) {
        cout << i << ":" << parent[i] << "\n";
    }
}

int main() {
    int numVertices, numEdges; cin >> numVertices >> numEdges;
    Graph g(numVertices);
    for (int i=0; i< numVertices ; ++i) {
        int u,v; cin >> u >> v;
        g.addEdge(u,v);
    }
    int maxThreads = omp_get_max_threads();
    int numThreads ; cin >> numThreads;
    int it= 0;
    double start = omp_get_wtime();
    vector<int> parent = findConnectedComponentsParallel(g, it, numThreads)
    double end = omp_get_wtime();
    cout << "Iterations: " << it << "\n";
    cout << "Execution Time: " << (end- start) * 1000.0 << "\n";
    printComponent(parent, numVertices);
    return 0;
}
```

→ P.T.O

(Test Case : 1)

5 4
0 1
1 2
2 3
3 4
4

» Iteration : 3
» Execution Time : 1.21498 ms
» vertex : parent (Component)
» 0 : 0
» 1 : 0
» 2 : 0
» 3 : 0
» 4 : 0

(TestCase : 2)

10 8
0 1
1 2
3 4
4 5
6 7
7 8
8 9
2 3
6

» Iteration : 4
» Execution Time : 1.68945 ms
» vertex : parent (component)
» 0 : 0
» 1 : 0
» 2 : 0
» 3 : 0
» 4 : 0
» 5 : 0
» 6 : 6
» 7 : 6
» 8 : 6
» 9 : 6

Testcase : 3

25 20
0 1
1 2
2 3
3 4
5 6
6 7
7 8
8 9
10 11
11 12
12 13
13 14
15 16
16 17
17 18
18 19
20 21
21 22
22 23
23 24

» Iterations : 3
» Execution Time : 2.26003 ms
» vertex : parent (component)
» 0 : 0
» 1 : 0
» 2 : 0
» 3 : 0
» 4 : 0
» 5 : 5
» 6 : 5
» 7 : 5

» 8:5
» 9:5
» 10:10
» 11:10
» 12:10
» 13:10
» 14:10
» 15:15
» 16:15
» 17:15
» 18:15
» 19:15
» 20:20
» 21:20
» 22:20
» 23:20
» 24:20

10.) Time Analysis for OpenMP Implementation: (with p processors)

(1.) Parallel Work:

- direct-connect : $O(m/p)$ per thread
- shortcut : $O(n/p)$ per thread
- alter : $O(m/p)$ per thread.

(2.) Synchronisation Overhead:

- Barrier Synchronisation : $O(\log p)$
- Reduction Synchronisation : $O(\log p)$
- overhead per iteration : $O(\log p)$

(3.) overall Parallel Time:

- Time_parallel = $(O(m\log^2 n/p) + \log^2 n \times \log p)$
- Speedup = T_serial / T_parallel = $p \times (m/(m+p\log p))$
- Efficiency = (Speedup / p) = $m/(m+p\log p)$

→ P.T.O

(Q.) Empirical Observation:

Test case	vertices	Edges	Thread	Complexity	Expected Trend
5S	4	4			
10	8	6			
25	20				

Test case	vertices	Edge	Thread	Complexity	Trend
Small Graph	5	4	4	O((n+m)/p)	Needly Instant
Medium Graph	10	8	6	O((n+m)/p)	Linear scaling
Large Graph	25	20	8	O((n+m)/p)	Significant scaleup with T=8.