COMPLEX COMPUTING PROBLEM

P&DC

HAFSA HAFEEZ SIDDIQUI

02-136212-026

BS(AI) – 5A

Problem Statement:

In an evolving banking network platform characterized by an expansive user base and intricate online connections, the performance of connectivity analysis becomes paramount. Users now form numerous accounts, and the interconnective network is distributed across multiple servers. The goal is to enhance efficiency by leveraging parallel Breadth-First Search (BFS) within a single server. The scenario outlines objectives such as graph partitioning, handling large-scale banking networks, implementing parallel BFS using OpenMP, and optimizing data structures.

Features:

Graph Partitioning for Single Server:

1) Distribute the banking network graph within a single server for efficient parallel processing.

```
def distribute_graph(graph, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Partition the graph into roughly equal-sized chunks
    nodes_per_process = len(graph) // size
    local_nodes = list(range(rank * nodes_per_process, (rank + 1) *
nodes_per_process))

    # Gather local node sets from all processes
    all_local_nodes = comm.allgather(local_nodes)

    # Create a local graph with only nodes and edges relevant to this
process
    local_graph = {}

    for node in local_nodes:
        local_graph[node] = [neighbor for neighbor in graph[node] if
neighbor in flatten(all_local_nodes)]

    return local_graph
```

2) Techniques for load-balanced distribution.

```
# Partition the graph into roughly equal-sized chunks
nodes_per_process = len(graph) // size
local_nodes = list(range(rank * nodes_per_process, (rank + 1) *
nodes_per_process))
```

Efficient Handling of Large-Scale Multi account Networks:

```
class User:
    def __init__(self, id):
        self.id = id
        self.friend_list = []
def initialize_users(num_users):
    users = {}
    for i in range(num_users):
        user = User(i)
        users[i] = user
    return users
```

```python
def show_users(users):
    for user in users.keys():
        print(f'User {users[user].id} has Friends {[u.id for u in
users[user].friend_list]}')


#efficient handling of user accounts:
def create_adjacency_list(users):
    graph = {}
    for user in users.keys():
        graph[user] = [client.id for client in users[user].friend_list]

    return graph
```

Parallel BFS Implementation using OpenMP:

1) Libraries Imported – mpi4py is the library used in python which has OpenMP at its backend.

```python
from mpi4py import MPI
```

2) Parallel BFS traversal

```python
def parallel_bfs(graph, start, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()
    num_nodes = len(graph)

    visited = np.zeros(num_nodes, dtype=bool)
    distances = np.zeros(num_nodes, dtype=int)

    # Global frontier to coordinate work among processes
    global_frontier = comm.allgather(graph)[rank]

    if global_frontier:
        # Each rank handles its local nodes
        local_frontier = global_frontier.pop(rank)

        # Perform BFS locally
        while local_frontier:
            current_node = local_frontier.pop(0)

            for neighbor in graph[current_node]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    distances[neighbor] = distances[current_node] + 1
                    local_frontier.append(neighbor)

        # Gather local frontiers to form the global frontier for the next
iteration
        # i.e combine all the local BFS
        global_frontier = comm.allgather(local_frontier)

    return visited, distances
```

Optimized Data Structures for Single Server Parallel Processing:

```
graph = create_adjacency_list(users)         # convert the social network into
an adjacency list
```

Every user had a friend list which were converted into a graph, since the BFS traversal implementation is being done on a graph.

Source Code: (at a glance)

```
import random
from mpi4py import MPI
import numpy as np

class User:
    def __init__(self, id):
        self.id = id
        self.friend_list = []

# flattens a list
def flatten(xss):
    return [x for xs in xss for x in xs]

def distribute_graph(graph, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Partition the graph into roughly equal-sized chunks
    nodes_per_process = len(graph) // size
    local_nodes = list(range(rank * nodes_per_process, (rank + 1) *
nodes_per_process))

    # Gather local node sets from all processes
    all_local_nodes = comm.allgather(local_nodes)

    # Create a local graph with only nodes and edges relevant to this process
    local_graph = {}

    for node in local_nodes:
        local_graph[node] = [neighbor for neighbor in graph[node] if neighbor
in flatten(all_local_nodes)]

    return local_graph


def gather_data(data, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()

    gathered_data = comm.gather(data, root=0)

    return gathered_data

def initialize_users(num_users):
    users = {}
    for i in range(num_users):
        user = User(i)
        users[i] = user
    return users
```

```python
def show_users(users):
    for user in users.keys():
        print(f'User {users[user].id} has Friends {[u.id for u in
users[user].friend_list]}')

def create_adjacency_list(users):
    graph = {}
    for user in users.keys():
        graph[user] = [client.id for client in users[user].friend_list]

    return graph

def create_random_users_and_add_friends(num_users):
    print('\n--- Creating users and adding random friends ---')
    users = initialize_users(num_users)
    for user in users.values():

        # add random friends to every user
        for _ in range(random.randint(1, num_users)):
            friend_id = random.randint(0, num_users - 1)
            if users[friend_id] not in user.friend_list and user !=
users[friend_id]:
                user.friend_list.append(users[friend_id])

    return users

def get_two_random_users(users):
    num_users = len(users)

    if num_users > 0:
        userA = users[random.randint(0, num_users - 1)]

        rand = random.randint(0, num_users - 1)
        while rand == userA.id:
            rand = random.randint(0, num_users - 1)

        userB = users[rand]

    return userA, userB

def parallel_bfs(graph, start, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()
    num_nodes = len(graph)

    visited = np.zeros(num_nodes, dtype=bool)
    distances = np.zeros(num_nodes, dtype=int)

    # Global frontier to coordinate work among processes
    global_frontier = comm.allgather(graph)[rank]

    if global_frontier:
        # Each rank handles its local nodes
        local_frontier = global_frontier.pop(rank)

        # Perform BFS locally
        while local_frontier:
            current_node = local_frontier.pop(0)
```

```
            for neighbor in graph[current_node]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    distances[neighbor] = distances[current_node] + 1
                    local_frontier.append(neighbor)

        # Gather local frontiers to form the global frontier for the next
iteration
        # i.e combine all the local BFS
        global_frontier = comm.allgather(local_frontier)

    return visited, distances

def degree_of_separation(userA, userB, users):
    arr = users[userA.id]
    return arr[userB.id] if arr[userB.id] != 0 else "No connection"

if __name__ == "__main__":

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    num_users = comm.Get_size()

    users = {}
    userA = userB = None

    # processor 0 initialises the users and shows us
    if rank == 0:
        users = create_random_users_and_add_friends(num_users)
        show_users(users)

        print('\n--- Taking random starting and target users ---')

        userA, userB = get_two_random_users(users)

        print(f"Starting user: {userA.id}")
        print(f"Target user: {userB.id}\n")

        print('--- Show the distributed graphs over the ranks ---')


    graph = create_adjacency_list(users)        # convert the social network
into an adjacency list
    graph = comm.bcast(graph, root=0)           # broadcast the graph to all
workers

    # Distribute the graph data over the ranks
    local_graph = distribute_graph(graph, comm)

    print('Rank: {} - has sub-graph: {}'.format(rank, local_graph))

    # Perform BFS in parallel
    local_visited, local_distances = parallel_bfs(graph, 0, comm)

    # Gather the results
    global_visited = gather_data(local_visited, comm)
    global_distances = gather_data(local_distances, comm)
```

```python
    if rank == 0:

        # Combine the results from all processes
        final_visited = np.logical_or.reduce(global_visited)
        print("\nFinal Visited Nodes:", final_visited.nonzero()[0])

        degreeOfSeparation = degree_of_separation(userA, userB, 
global_distances)

        if degreeOfSeparation != "No connection":
            print(f"\nDegree of separation between User {userA.id} and User 
{userB.id} is: {degreeOfSeparation}")
        else:
            print(f"\nNo connection found between User {userA.id} and User 
{userB.id}.")
```

Output:

```
hafsa@hafsa-virtual-machine:~/Desktop$ mpiexec -n 4 python3 bfs.py

--- Creating users and adding random friends ---
User 0 has Friends [2, 1]
User 1 has Friends [0, 3]
User 2 has Friends [0, 3]
User 3 has Friends []

--- Taking random starting and target users ---
Starting user: 3
Target user: 2

--- Show the distributed graphs over the ranks ---
Rank: 3 - has sub-graph: {3: []}
Rank: 0 - has sub-graph: {0: [2, 1]}
Rank: 1 - has sub-graph: {1: [0, 3]}
Rank: 2 - has sub-graph: {2: [0, 3]}

Final Visited Nodes: [0 1 2 3]

No connection found between User 3 and User 2.
```

Figure i No connection between user 2 & 3 since user 3 has no friends

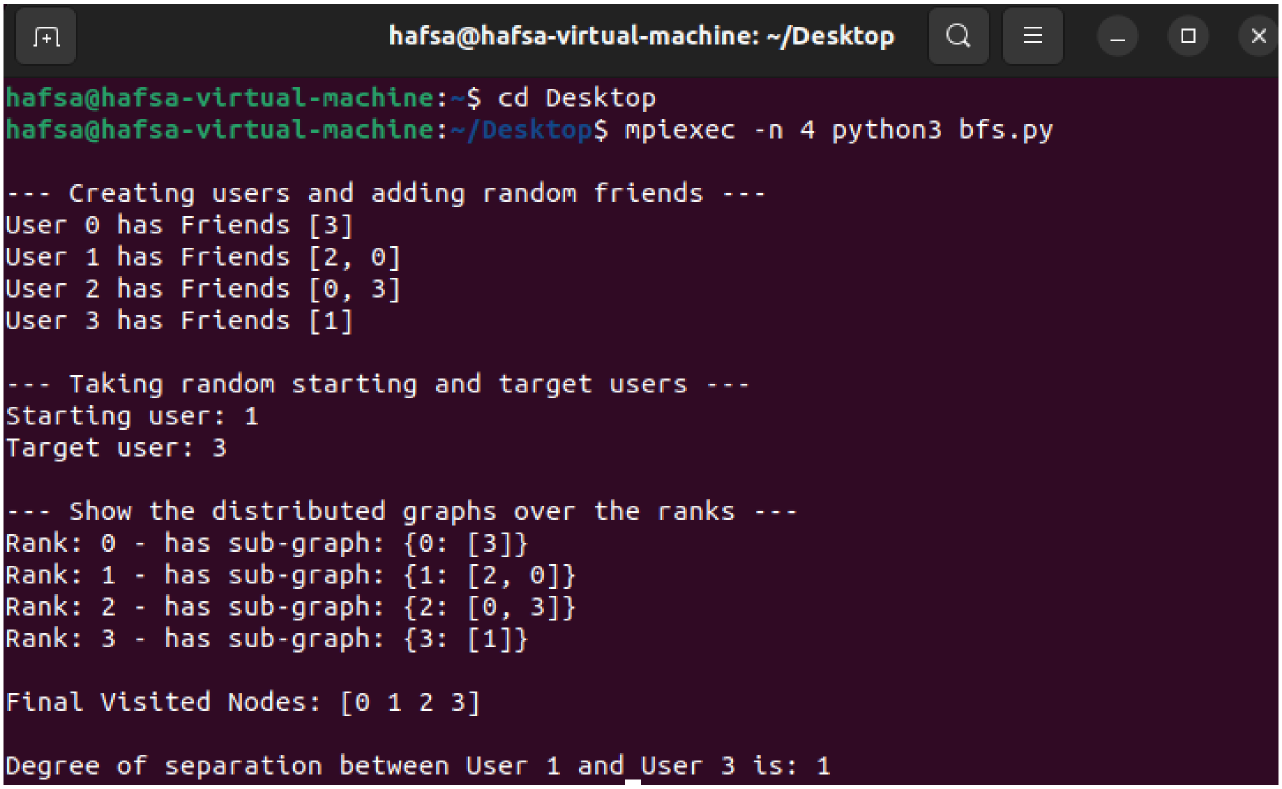

Figure ii Degree of Separation=1