Report: Exercises 4

Sevde Yanik & Sima Esmaeili

Problem 1

Using MPI in C++, the Vogel-Strauß algorithm is implemented. The algorithm first puts all messages into network using asynchronous send operations and then it receives everything that arrives. The implementation is tested by running it on multiple processes where each process sends a unique message.

The implementation is explained step-by-step:

1. At the beginning MPI is initialized and the rank and size of the MPI communicator is retrieved.
2. A unique message is generated for each process based on its rank.
3. Using a loop, each process asynchronously sends its message to all other processes, except for itself.
4. Another loop is used to receive messages from other processes. Again, each process receives messages from all other processes, except for itself.
5. All messages are printed to the console together with the rank of the process that retrieved it.
6. MPI is finalized.

Problem 2

Run the Hypercube Gossiping algorithm with $2^3 = 8$ processing elements represented in binary as (000, 001, 010, 011, 100, 101, 110, 111). Assume that each processing element starts with a different character from {A,B,C,D,E,F,G,H}. Show all steps of the execution until all processing elements have all the available characters.

Start:

PE	Character
000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

Every PE has a message m of length n.
In the end every PE should know all messages.

Let '$\cdot$' be the union operation; $p = 2^d$

```
1: function UNION(i)
2:     y ← m
3:     for 0 ≤ j < d do
4:         y' ← the y from PE i ⊕ 2^j
5:         y ← y · y'
6:     return y
```

Running the Hypercube Gossiping algorithm using the union operation provided, Step 1:

1. PE 000(A) receives B from PE 001, A . B = AB
2. PE 001(B) receives A from PE 000, B . A = BA
3. …

Result:

PE	Character
000	AB
001	BA
010	CD
011	DC
100	EF
101	FE
110	GH
111	HG

Step 2:

1. PE 000(AB) receives CD from PE 010, AB . CD = ABCD
2. PE 001(BA) receives DC from PE 011, BA . DC = BADC
3. …

Result:

PE	Character
000	ABCD
001	BADC
010	CDAB
011	DCBA
100	EFGH
101	FEHG
110	GHEF
111	HGFE

Step 3:

1. PE 000(ABCD) receives EFGH from PE 100, ABCD . EFGH = ABCDEFGH
2. PE 001(BADC) receives FEHG from PE 101, BADC . FEHG = BADCFEHG
3. …

Result:

PE	Character
000	ABCDEFGH
001	BADCFEHG
010	CDABGHEF
011	DCBAHGFE
100	EFGHABCD
101	FEHGBADC
110	GHEFCDAB
111	HGFEDCBA

At this point, all processing elements have all the available characters. The final state is:

- PE 000: ABCDEFGH
- PE 001: BADCFEHG
- PE 010: DCABGHFE
- PE 011: DCBAHGFE
- PE 100: EFGHABCD
- PE 101: FEHGBADC
- PE 110: GHFEDCAB
- PE 111: HGFEDCBA

The Hypercube Gossiping algorithm using the union operation has finished running. Now, every processing element has gathered all the messages that were initially given to them.

Problem 3

The pseudocode given below builds a reverse graph adjacency array using the map-reduce paradigm:

```
function Map(vertex, outgoing_edges)
        for each destination_vertex in outgoing_edges:
                emit(destination_vertex, vertex)
```

```
function Reduce(destination_vertex, source_vertices):
    reversed_outgoing_edges = []
    for each vertex in source_vertices:
        reversed_outgoing_edges.append(vertex)
    emit(destination_vertex, reversed_outgoing_edges)

function BuildReverseGraphAdjacencyArray(graph_adjacency_array):
    // Map stage
    map_output = []
    for each vertex, outgoing_edges in graph_adjacency_array:
        intermediate_map_output = Map(vertex, outgoing_edges)
        map_output.add(intermediate_map_output)

    // Sort map output by destination vertex

    // Reduce stage
    reverse_graph_adjacency_array = []
    for each destination_vertex, source_vertices in sorted_map_output:
        intermediate_reduce_output = Reduce(destination_vertex, source_vertices)
        reverse_graph_adjacency_array.add(intermediate_reduce_output)
    return reverse_graph_adjacency_array
```

Step-by-step explanation to the pseudocode:

1. The Map() function looks at each vertex in the original graph and its outgoing edges. For each outgoing edge, it creates a new pair where the edge becomes the key, and the vertex it came from becomes the value. This effectively reverses the direction of the edges.
2. The Reduce() function takes these pairs and groups them based on the destination vertex. It collects all the vertices associated with the same destination vertex and creates a list of reversed outgoing edges.
3. In the BuildReverseGraphAdjacencyArray() function, the map stage applies the map function to each vertex in the original graph and saves the results.
4. In the sort stage, the intermediate results based on their keys (outgoing edges) are organized, grouping the corresponding values (original vertices).
5. In the reduce stage, the reduce function is applied to each group of values that share the same destination vertex. This step builds the final reverse graph adjacency array by using the destination vertex as the key and creating a list of source vertices as the value.
6. Finally, the reverse graph adjacency array is returned as the result of the BuildReverseGraphAdjacencyArray() function.

Problem 4

Consider an undirected graph G = (V,E) and a partitioning function Π : V → {1, 2, . . . , k} that assigns each node to one of k blocks. Provide pseudo-codes to implement each of the following tasks using the map-reduce paradigm.

a Compute the edge-cut, i.e., the total weight of edges that run between blocks. (2 points)

b Compute the cardinality of each block. (2 points)

c List all boundary nodes in each block, i.e., the nodes contained in a block that have at least one edge to a node contained in another block. (3 points)

d Calculate the weight of the edges of the quotient graph associated with the partition. (3 points)

a.

```
Map(key: edge, value: weight):
    if Π(edge.source) != Π(edge.target):
        Emit("edge-cut", weight)

Reduce(key: "edge-cut", values: weights):
    total_weight = 0
    for weight in values:
        total_weight += weight
    Emit("edge-cut", total_weight)
```

Map Phase:

- Check if the source and target nodes of each edge belong to different blocks.
- Emit the weight of edges that cross block boundaries with the key "edge-cut".

Reduce Phase:

- Receive the intermediate key-value pairs with the key "edge-cut" and the weights of the edges.
- Sum up the weights to calculate the total weight of edges running between blocks.
- Emit a single key-value pair with the key "edge-cut" and the total weight as the value.

b.

```
Map(key: node, value: block):
    Emit(block, 1)

Reduce(key: block, values: counts):
    total_count = sum(values)
    Emit(block, total_count)
```

- In the Map function, for each node, we emit a key-value pair where the key is the block identifier to which the node belongs, and the value is 1.
- This assigns a count of 1 to each node in its corresponding block.
- In the Reduce function, we receive the intermediate key-value pairs grouped by block identifier.
- We calculate the total count by summing up all the counts for each block.
- Finally, we emit a single key-value pair for each block, where the key is the block identifier, and the value is the computed total count.
- This provides the cardinality (number of nodes) of each block in the partitioned graph.

c.

```
Map(key: edge, value: block):
    if Π(edge.source) != Π(edge.target):
        Emit(block, edge.source)
        Emit(block, edge.target)

Reduce(key: block, values: nodes):
    boundary_nodes = unique(nodes)
    Emit(block, boundary_nodes)
```

- In the Map function, for each edge, we check if the source and target nodes belong to different blocks.
- If they do, we emit the block identifier along with the source and target nodes as key-value pairs.
- The Reduce function receives these intermediate key-value pairs and collects the nodes associated with each block.
- It removes any duplicates to get the unique boundary nodes.
- Finally, it emits a key-value pair for each block, where the key is the block identifier and the value is the list of boundary nodes.

d.

```
Map(key: edge, value: weight):
    source_block = Π(edge.source)
    target_block = Π(edge.target)
    if source_block != target_block:
        Emit((source_block, target_block), weight)

Reduce(key: (source_block, target_block), values: weights):
    total_weight = sum(values)
    Emit((source_block, target_block), total_weight)
```

- In the Map function, for each edge, we determine the blocks to which the source and target nodes belong. If they belong to different blocks, we emit a key-value pair where the key is a tuple containing the source block and target block, and the value is the weight of the edge.
- In the Reduce function, we receive the intermediate key-value pairs grouped by the tuple (source block, target block). We calculate the total weight by summing up the weights for each key, and emit a key-value pair where the key is the tuple (source block, target block), and the value is the computed total weight.