Wenxi Chen
wchen466

# Assignment 2

CS 4438
October 24, 2021

## Question 1

### Time complexity and communication complexity

**Time complexity**
In the original algorithm we went through in class, the first phase of the message travels a distance of $2^0$ and then it is sent back to the processor p with the largest ID. The next phase will have the distance of $2^1$ and the one after that will have to travel $2^1$ before turning around until reaching the largest ID. In this modified network and algorithm, every processors except the largest processor are identical. Even with

```
if id = id' and d = 1 then status <- Not Leader
```

you will have to visit all the processors before finally sending out the END message. Therefore, the time complexity of this algorithm is number of rounds:

$$f(n) = 2^1 + 2^2 + 2^3 + ... + 2^{i+1} + n = \sum_{j=1}^{i+1} 2^j + n = 2^{i+2} - 2 + n = 4n + n - 2 = O(n)$$

**Communication complexity**
Likewise, this algorithm will send messages to the same number of active processors compare to the original algorithm in order to reach the largest ID while travelling to the right. In phase 0, there are $n$ active processors and less than $2n$ messages will be sent. In phase 1, there are $n/2$ active processors and less than $n/(2^0 + 1) * 2^2$ messages will be sent...and so on. Therefore, the communication complexity of this algorithm is the number of messages:

$$f(n) = 2n + (\frac{n}{2^0+1} * 2^2) + (\frac{n}{2^1+1} * 2^3) + ... + (\frac{n}{2^{i-1}+1} * 2^{i+1}) + n = 3n + \sum_{j=0}^{i-1} \frac{n}{2^j+1} * 2^{j+2}$$

$$3n + \sum_{j=0}^{i-1} \frac{n}{2^j+1} * 2^{j+2} < 3n + \sum_{j=0}^{i-1} \frac{n}{2^j} * 2^{j+2}$$

$$3n + \sum_{j=0}^{i-1} \frac{n}{2^j} * 2^{j+2} = 3n + \sum_{j=0}^{i-1} 4n = 3n + 4nlog_2n = O(nlogn)$$

# Assignment 2

## Question 2

### Algorithm

Algorithm **findPosition** (id)

```
In:  Processor id
Out:  The number of processors to the left of each processor

private static int count <- 0
boolean leftmostProcessor <- TRUE if you do not have any neighbours to the left
boolean rightmostProcessor <- TRUE if you do not have any neighbours to the right
if (both rightmostProcessor and leftmostProcessor are true) return 0

Initial message mssg consists of two parameters:  rightNeighbour and id
increment count by 1

Loop:
if (mssg != null) then send mssg to right neighbour
    if (mssg is travelling to the left) return count
    if (mssg's data = "END") decrement by 1 and return count
mssg <- null
m <- receive message
if (m != null)
     if (m is coming from the left and its data is not "END")
         send mssg to right neighbour and increment count by 1
        if (rightmostProcessor)
            send mssg to the left neighbour
    else
        send mssg to the left neighbour and decrement count by 1
        if (lefttmostProcessor)
            send mssg to the right neighbour with data = "END"
return 0
```

### Java implementation

See Position.java

## Proof of termination

Once the rightmost processor has been reached, the message will proceed to travel to the left until it reaches the leftmost processor. For each processor it visits on its way back (to the left), the algorithm returns the current count. You return $n-1$ times. In the end when you finally reach the leftmost processor, you will send a message containing "END" to your right neighbour, and in the next round, it will check if the mssg contains "END". If it does, it means the leftmost processor has been reached, therefore it will return one last time. There will be $n$ returns in total which equals the the number of processors. Therefore all the processors terminate the execution of the algorithm.

## Proof of correctness

Let $p_0$, $p_n$ be the leftmost and rightmost processor. When travelling from $p_0$ to $p_n$, for every processor it passes, count increments by 1. Once the algorithm arrives $p_n$, the count will equals to $n-1$. The count is returned and the direction of travel changes, now travelling to the left. For each process it visits, count decrements by 1. Once we arrive $p_0$, the count should be 0. Now we have successfully returned the number of processor to the left of our current processor, for each processor.

## Time complexity and communication complexity

### Time complexity

$f(n) = n + n = 2n = O(n)$

### Communication complexity

$f(n) = n + n = 2n = O(n)$

## Question 3

### Algorithm

Algorithm **findLargest** (id)

```
In:  Processor id
Out:  Leader, if id is the largest in the ring, or NotLeader, otherwise

Initial message mssg consists of two parameters:  leftNeighbour and id.
Initial status <- unknown.

Loop:
if (mssg != null) then send mssg to left neighbour
    if mssg contains status "END" return leaderID
mssg <- null
m <- receive message
if (m != null)
     if (status of m equals to "END")
         if (source of m equals the current right neighbour)
             send mssg to left neighbour with status = "END"
         else
             send mssg to right neighbour with status = "END"
     else
         if (the data received = id)
             send mssg to left neighbour with status = "END"
             status <- "Leader"
             LeaderID <- id
         else if (the data received is larger than id)
             if (source of m = right neighbour)
                 send mssg to left neighbour with data of m
             else
                 send mssg to right neighbour with data of m
             if (status = "unknown")
                 status <- "Not leader"
         else
             mssg <- null
return 0
```

# Assignment 2

## Java implementation

See LargestID.java

## Proof of termination

Once the processor with largest id changes its status to Leader, it creates and END message. The END message is never discarded, but it is always forwarded to the right (or left) neighbour; hence all processors will receive the END message.
After forwarding the END message the algorithm terminates, so all the processors terminate the execution of the algorithm.

## Proof of correctness

Let $p_{max}$ be the processor with the largest id. This algorithm will never discard the message containing $p_{max}$'s id.
Therefore, all processors will receive and forward the message with $p_{max}$'s id to its right (or left) neighbour, changing their status to NotLeader.
When $p_{max}$ receives a message with its own id, all other processors will have received it and hence $p_{max}$ knows that it has the largest id and so it correctly changes its status to Leader, then the program terminates.

## Time complexity and communication complexity

### Time complexity
$f(n) = n + n = 2n = O(n)$

### Communication complexity
$f(n) = n + n = 2n = O(n)$

## Question 4

### Algorithm

Algorithm **findDiameter** (id)

```
In:  Processor id
Out:  The maximum distance between two processors in the network

private static int count <- 0
Vector d <- new integer Vector stores the order of each spoke
isTerminal <- TRUE if size of v = 1
isSpoke <- TRUE if size of v = 2
isHub <- TRUE if size of v >= 3

Initial message mssg send to the closer neighbour to hub along with count as data
Loop:
if (mssg != null)
     send message
     return the current data, which is the count
mssg <- null
m <- receive message
if (m != null)
     if (isSpoke)
         count <- m.data()
         send mssg to the close neighbour to the hub with count + 1 as data
     else if (isHub)
         add m.data() + 1 to Vector d
     if (the size of Vector d equals to the size of Vector v) // no more processors
         return the combine value of the last element and the second to last element
return 0
```

### Java implementation

See Diameter.java

## Proof of termination

This algorithm will and always terminate once the size of Vector d equals to the size of Vector v. Each spoke will send a message at some point to the Hub with its length and it will be stored immediately to d. Once d and v has the same size, it will return the combined value of the least two elements added, which represents the longest two spoke. The algorithm will not terminate until it receives a value from all spokes. Nor it will not terminate since the algorithm compares v and d's size every round.

## Proof of correctness

This algorithm will always return the combined value of last two values being added into Vector v. When the hub received the last value from the longest spoke and add it to to d, it checks for the size of d and v. If they are equal, it means all spokes has sent their length to hub. We then return the correct answer.

## Time complexity and communication complexity

### Time complexity

$f(n) = n = O(n)$

### Communication complexity

$f(n) = n = O(n)$