1

Parallel and Distributed Computing

Lecture 9

Basic Communication Operations-II

2

All-to-All Broadcast and All-to-All Reduction

1. All-to-All Broadcast

Meaning

Every processor sends its data to **all other processors**.
If there are **p processors**, then every processor ends up with **p pieces of data**.

How it Works

- Each processor has some data `d_i`.
- After All-to-All Broadcast:
  - P0 has {d0, d1, ..., d(p-1)}
  - P1 has {d0, d1, ..., d(p-1)}
  - ...
  - P(p-1) has all data.

Cost

With a simple communication model:

$$T = (t_s + mt_w)(p-1)$$

But using efficient algorithms (like recursive doubling):

$$T = (t_s + mt_w)\log p$$

2. All-to-All Reduction

Meaning

Opposite of broadcast + reduction.

Every processor contributes a value (e.g., numbers), and **each processor ends up with the final reduced result** (sum, max, min, etc.).

Example:
Each processor has a number. You want the **sum** on **every processor**.

How it Works

- First, perform **reduction** (tree-based) → single result.
- Then **broadcast** the result to all processors.

Cost

Total cost:

$$T = 2(t_s + mt_w) \log p$$

---

3. Permuting a Matrix Using 2D Block Distribution

The Idea

Matrix is divided into **2D blocks** across processors.
Example processor grid: `√p × √p`.

Each processor holds one block of the matrix.

Now, if you want to **permute rows or colun**, or **transpose the matrix**, you must exchange blocks between processors.

Why 2D Block Distribution?

Because:

- It reduces communication volume.
- More balanced load for large matrices.
- Each processor communicates with fewer neighbors.

When Permuting a Matrix

Example operations:

- Transpose
- Shuffle rows/columns
- Rearrange blocks

Each block moves to a new processor positi

Communication Pattern

These permutations typically use:

✔ **All-to-All communication** between processors in a row or column
✔ Each block moves exactly once
✔ Communication cost depends on block size

Cost

$$T = t_s p + m t_w$$

Or for optimized:

$$T = (t_s + m t_w) \log p$$

Short Summary (Easy to Learn)

Concept	Meaning	Cost
All-to-All Broadcast	Every processor sends data to all others	$(t_s + mt_w) \log p$
All-to-All Reduction	All contribute and all get reduced result	$2(t_s + mt_w) \log p$
Matrix Permutation (2D Block)	Rearrangements of a distributed matrix	Uses All-to-All among processor rows/cols

Basic Communication Operations

(All-to-All Broadcast and All-to-All Reduction)

All-to-All Broadcast

- A generalization to of one-to-all broadcast.
- Every process broadcasts m-word message.
  - The broadcast-message for each of the processes can be different than others

All-to-All Reduction

- Dual of all-to-all broadcast
- Each node is the destination of an all-to-one reduction out of total P reductions.

Basic Communication Operations

(**All-to-All Broadcast and All-to-All Reduction**)

4

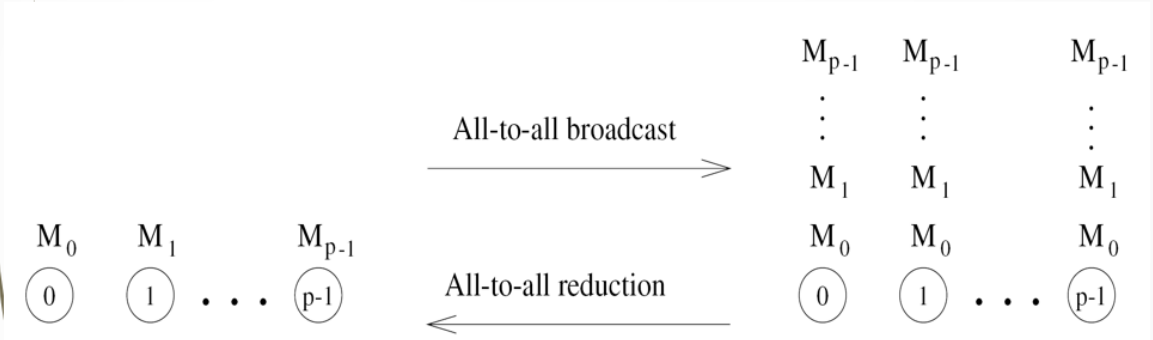


Figure 4.8 All-to-all broadcast and all-to-all reduction.

A naïve Broadcast method may be performing **P** one-to-all broadcasts. This will result **P(log(p)(t(s) + mt(w)))** communication time.

Solution?

CS3006 - Fall 2021

Reduction: let's say every node has generated different P random numbers. The problem is to sum the respective indices and store it to respective process

Solution: It is possible to use the communication links in the interconnection network more efficiently by performing all p one-to-all broadcasts simultaneously so that all messages traversing the same path at the same time are concatenated into a single message whose size is the sum of the sizes of individual messages.

Problem Setup: All-to-All Reduction Example

- **Scenario:**
  - There are **P processes (nodes)**.
  - Each process generates **P random numbers**, e.g.,

```
Process 0: r0_0, r0_1, ..., r0_(P-1)
Process 1: r1_0, r1_1, ..., r1_(P-1)
...
Process P-1: r(P-1)_0, ..., r(P-1)_(P-1)
```

  - **Goal:** Sum numbers **index-wise** so that each process `i` stores the sum of all numbers at index `i`:

```
Process 0 stores: r0_0 + r1_0 + ... + r(P-1)_0
Process 1 stores: r0_1 + r1_1 + ... + r(P-1)_1
...
```

This is a classical **all-to-all reduction** problem.

Naive Approach

- Do **P separate one-to-all reductions** for each index.
- **Problem:**
  - Lots of redundant communication.
  - Many messages traverse the same network paths **separately**, wasting bandwidth.

Optimized Approach: Concurrent Broadcasts with Concatenation

- **Idea:**
  - Instead of sending **one message per reduction**, each process can **concatenate multiple messages** destined for the same path.
  - Perform **all P one-to-all reductions simultaneously**, but **merge messages on the same communication link**.
- **Benefits:**
  - **Fewer transmissions** → reduces startup cost `t_s`.
  - **Higher bandwidth efficiency** → total message size is sum of individual messages (`m_total = sum of m_i`).

Step-by-Step

1. **Each process groups messages for the same destination path.**
   - Example: Process 0 wants to send `r0_0, r0_1, ... r0_(P-1)` to processes 0,1,...,P-1.
   - It sends **concatenated messages** to intermediate processors along the path.
2. **Intermediate processors forward concatenated messages** to the next hop.
3. **Final process receives concatenated messages**, then splits them and performs the **sum for its index**.

Communication Model

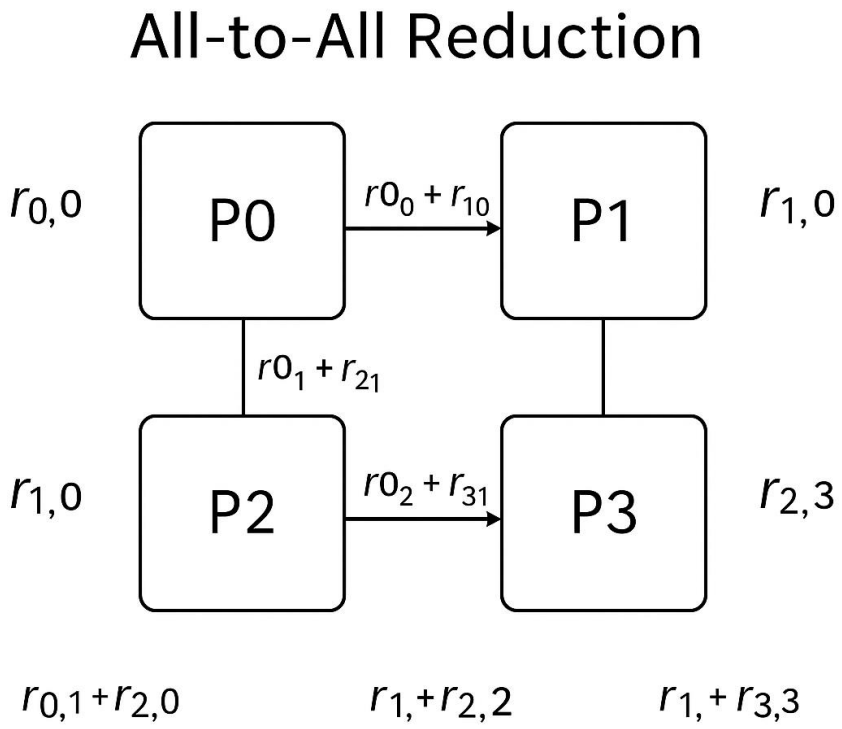
- Let `t_s` = startup time per message, `t_w` = time per word.
- Total time for **all-to-all reduction**:

$$T = (t_s + mt_w) \log P$$

- `m` here is the **size of concatenated messages**, much larger than individual messages.
- But **overall efficiency improves** because fewer message startups (`t_s`) happen.

✅ **Key Insight:**

- The trick is **simultaneous broadcasts with concatenation**, not doing separate reductions sequentially.
- This is how large distributed matrix computations (like **matrix multiplication with 2D blocks**) efficiently compute all partial sums.



bloɑu p = concatenated messages condemone on moι park

1. All-to-All Broadcast

- **Definition:** Generalization of **one-to-all broadcast**.
  - In one-to-all: **only one process** sends the same message to all others.
  - In all-to-all: **every process** sends its own message to **all other processes**.
- **Key Points:**
  - Each process has a **message of length** `m`.
  - Messages can be **different** for each process.
  - After communication, **every process knows all messages**.
- **Cost Model:**

  Using `t_s` = startup time, `t_w` = time per word:

$$T \approx (t_s + mt_w) \log p$$

- **Example in Matrix:**
  - In **matrix multiplication** using a 2D processor grid:
    - Each processor may hold a **block of A and a block of B**.
    - To compute the product, each processor **broadcasts its block to all processors in its row or column**.

---

2. All-to-All Reduction

- **Definition:** Dual of all-to-all broadcast.
  - Every process **contributes a value**, and each process is a **destination of a reduction**.
  - Think: "All contribute → All get the reduced result."
- **Key Points:**
  - Each node participates in **P reductions**, one for each destination.
  - Reduction could be **sum, max, min, or other operations**.
  - Often implemented with **tree-based or recursive doubling** to minimize communication.

- **Applications in Matrix Operations:**
  - **Matrix multiplication:** After local block multiplications, processors **reduce their partial results** to compute the final product.
  - **Matrix-vector multiplication:** Partial contributions of vector products are reduced across rows/columns to get the final vector entries.

Visual Summary

Operation	Who Sends	Who Receives	Example Use
All-to-All Broadcast	All processes	All processes	Broadcasting matrix blocks for multiplication
All-to-All Reduction	All processes contribute	All processes	Summing partial matrix products in distributed multiplication

5

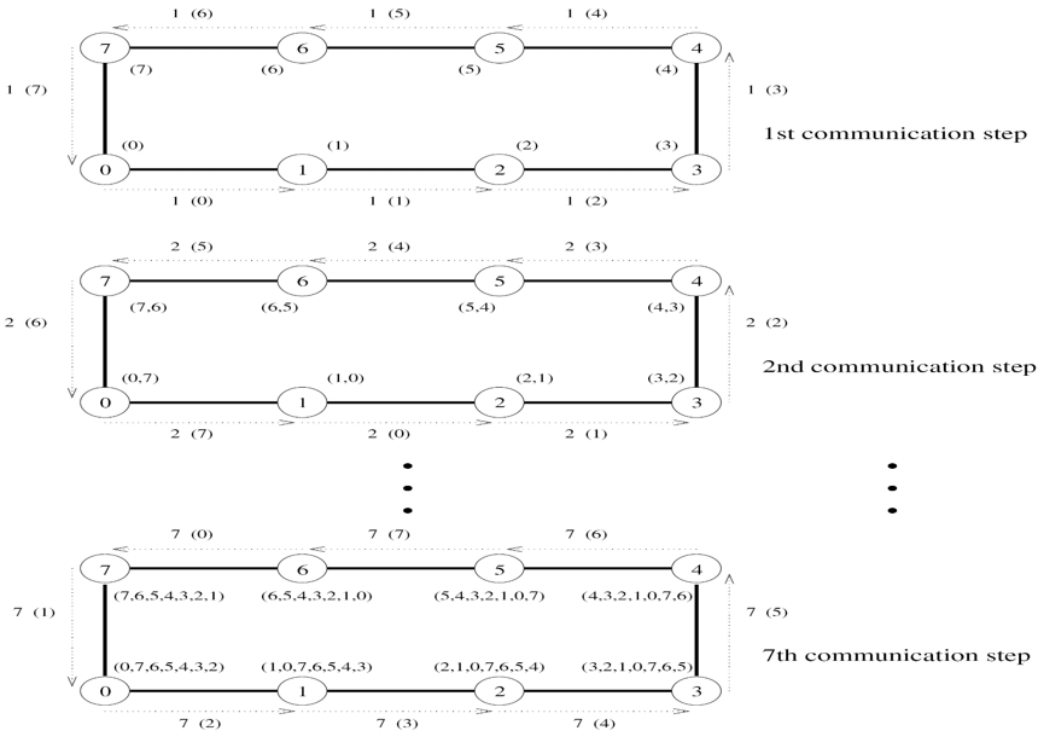


Figure 4.9 All-to-all broadcast on an eight-node ring. The label of each arrow shows the time step and, within parentheses, the label of the node that owned the current message being transferred before the beginning of the broadcast. The number(s) in parentheses next to each node are the labels of nodes from which data has been received prior to the current communication step. Only the first, second, and last communication steps are shown.

✅ Linear Ring Broadcast (All-to-All Broadcast in an 8-Node Ring)

This figure shows **how every node in an 8-node ring passes its data to all other nodes**.
Each step shows which node is sending which message to its neighbor.

🔥 Understanding the Setup

- There are **8 nodes** arranged in a **ring**:
  `0 – 1 – 2 – 3 – 4 – 5 – 6 – 7 – back to 0`
- Every node has **its own data**, and the goal is:
  Each node must receive data from all other nodes.
- Communication happens in **steps**, and in each step every node sends **one message** to the next node.

📌 What the Figure Shows

1️⃣ First Communication Step

Example:

- Node 0 sends its data **(0)** to node 1
- Node 1 sends its data **(1)** to node 2
- Node 2 sends **(2)** to node 3
- … and so on in the ring
- Node 7 sends **(7)** to node 0

Each node has received data from **one neighbor only**.

2️⃣ Second Communication Step

Now each node forwards the message it just received.

Example:

- Node 0 now has **(7, 0)**
- It sends **(7)** to node 1
- Node 1 sends **(0)** to node 2
- Node 2 sends **(1)** to node 3
- … etc.

After step 2, each node has data from **two different nodes**.

7️⃣ Final (7th) Communication Step

After 7 steps:

- Every node has received data from **all other 7 nodes**.
- The sets shown under each node (e.g., `(7,0,6,5,4,3,2,1)`) list which messages it has collected.

☞ Total steps required = **N − 1** = **7** for 8 nodes.

🔁 Why is it called a “ring broadcast”?

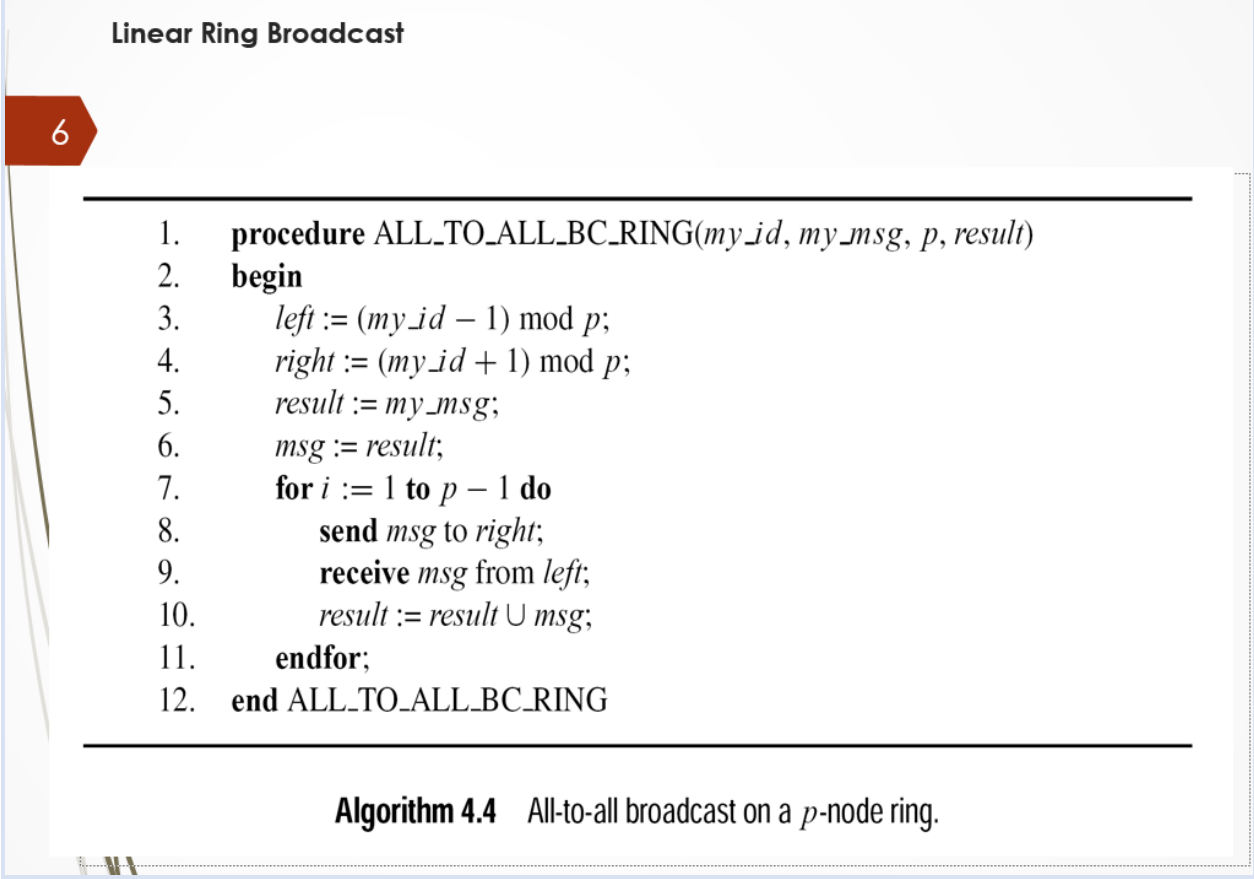
Because:

1. The nodes are connected in a **ring topology**
2. Each node **broadcasts (sends) its message to all others**
3. Messages circulate around the ring until all nodes receive everything.

🎯 Key Points for Exams

- All-to-all broadcast in a ring takes **N − 1 communication steps**.
- Each step: every node sends **one message** to its neighbor.
- Messages propagate **one hop per step**.
- After step *k*, each node has **k new messages**.

If you want, I can also explain this with a **simpler example**, a **table**, or even **draw the steps one by one**.



At any node my-id is the value or label of that node

The initial message to be broadcast is known locally as my_msg at each node

At the end of the procedure, each node stores the collection of all p messages in result

✅ Understanding Algorithm 4.4 — All-to-All Broadcast on a Ring

This algorithm makes **every node send its message to all other nodes** in a **p-node ring**.

🔷 Variables Meaning

Variable	Meaning
`my_id`	The index of the current node (0,1,2,...,p-1)
`my_msg`	The message this node wants to broadcast
`p`	Total number of nodes in the ring
`result`	A set that stores all received messages
`left`	The left neighbor of this node
`right`	The right neighbor of this node
`msg`	Temporary message being passed around

🔷 Neighbor Calculation

ini Copy code

```ini
left  = (my_id - 1) mod p
right = (my_id + 1) mod p
```

Example for node 0 in an 8-node ring:

- `left = 7`
- `right = 1`

This maintains the **ring**.

🔷 Initial Setup

Each node starts with its own message:

```ini
result = my_msg
msg = result
```

So every node initially knows **only its own data**.

🔷 Main Loop: p − 1 Iterations

```sql
for i = 1 to p-1 do
    send msg to right
    receive msg from left
    result = result ∪ msg
endfor
```

Meaning:

🔶 Step 1

- Each node sends its message clockwise (to the right)
- It receives one message from the left
- It adds that message to its `result` set ↓

🔶 Step 2

- Now the node forwards the message it received in step 1
- Receives another message
- Adds it to result

This continues for **p−1 steps**, so that every node eventually receives:

```css
all p messages
```

🎯 At the End

Every node contains:

```ini
result = { message from node0, node1, node2, …, node(p-1) }
```

✔ **This means all-to-all broadcast is complete.**

📌 Why p − 1 Steps?

Because a message moves **one hop per step**, and the farthest node in a ring is **p − 1 hops away**.

So after **p − 1 steps**, each message has completed a full loop.

🔥 All-to-All Broadcast — Example with 4 Nodes (0,1,2,3)

Each node has its own message:

Node	my_msg
0	A
1	B
2	C
3	D

Every node MUST end with:
`{A, B, C, D}`

Ring connection:
`0 → 1 → 2 → 3 → back to 0`

📌 Step 0 (Initial State)

Each node knows only its own message:

Node	result
0	A
1	B
2	C
3	D

📌 Step 1 (i = 1)

Each node sends its message to the **right** and receives from the **left**.

Messages:

- Node 0 sends **A** → Node 1
- Node 1 sends **B** → Node 2
- Node 2 sends **C** → Node 3
- Node 3 sends **D** → Node 0

After receiving:

Node	received msg	result
0	D	A + D
1	A	B + A
2	B	C + B
3	C	D + C

📌 Step 2 (i = 2)

Every node now forwards the message it got in Step 1.

Messages:

- Node 0 sends **D** → Node 1
- Node 1 sends **A** → Node 2
- Node 2 sends **B** → Node 3
- Node 3 sends **C** → Node 0

After receiving:

Node	received msg	result
0	C	A, D, C
1	D	B, A, D
2	A	C, B, A
3	B	D, C, B

📌 Step 3 (i = 3)

Final step (since p − 1 = 3).

Messages:

- Node 0 sends **C** → Node 1
- Node 1 sends **D** → Node 2
- Node 2 sends **A** → Node 3
- Node 3 sends **B** → Node 0

After receiving:

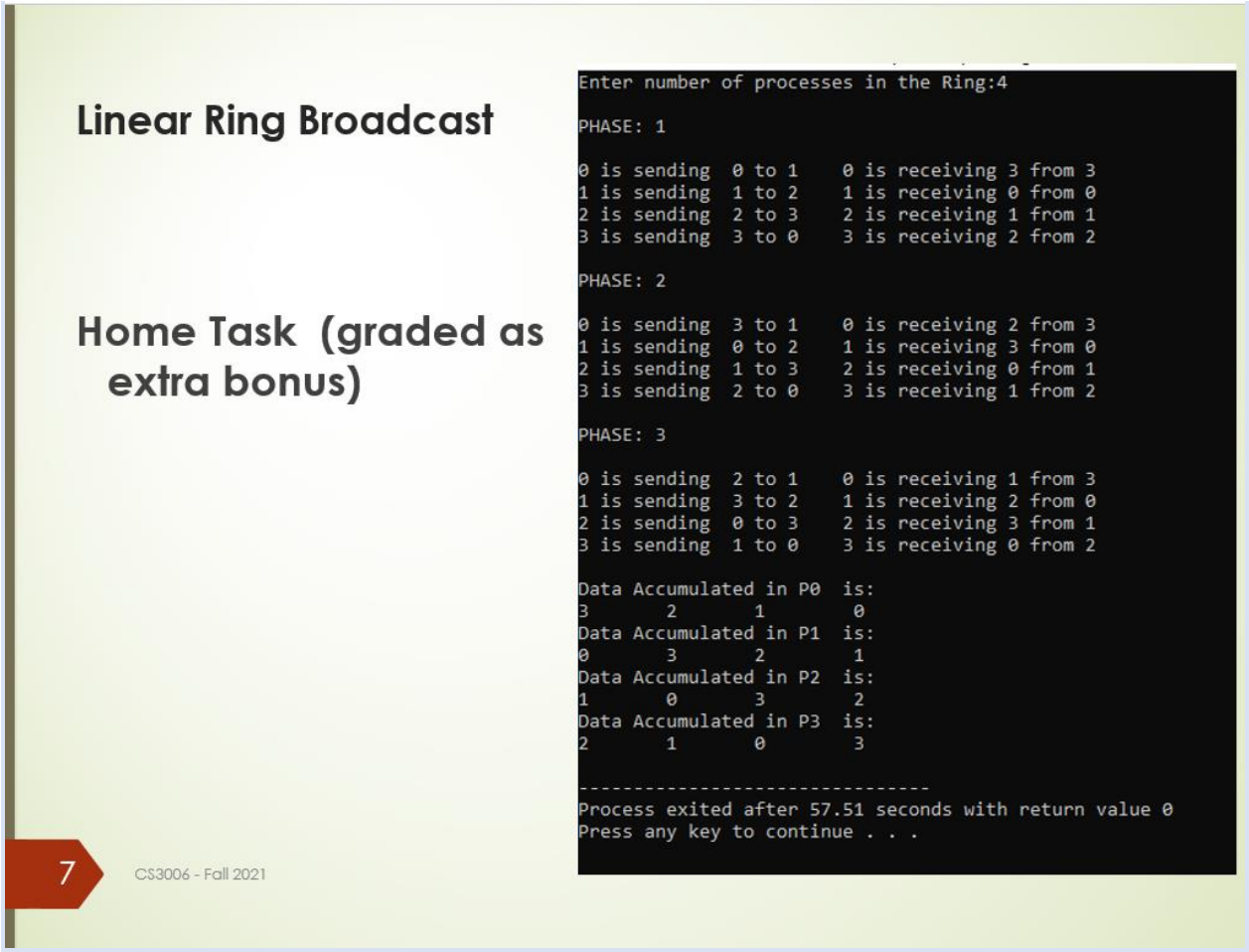
Node	received msg	result
0	B	A, D, C, B
1	C	B, A, D, C
2	D	C, B, A, D
3	A	D, C, B, A

🎉 Final Result (All-to-All Completed!)

Every node now contains:

```
{A, B, C, D}
```

Which means the broadcast was **successful**.



Write a program that takes number of processes in the ring from user and generates the following output.

You can take help from the algorithm described in the book.

✓ Python Program — Linear Ring Broadcast Simulation

```
# Linear Ring Broadcast Simulation
# All-to-All Broadcast on a Ring of p processes

p = int(input("Enter number of processes in the Ring: "))

# Each process starts with its own message = process id
# result[i] will store messages accumulated in process i
result = [[i] for i in range(p)]

print()

# Perform p-1 phases (communication steps)
for phase in range(1, p):
    print(f"PHASE: {phase}")

    # For display
    for i in range(p):
```

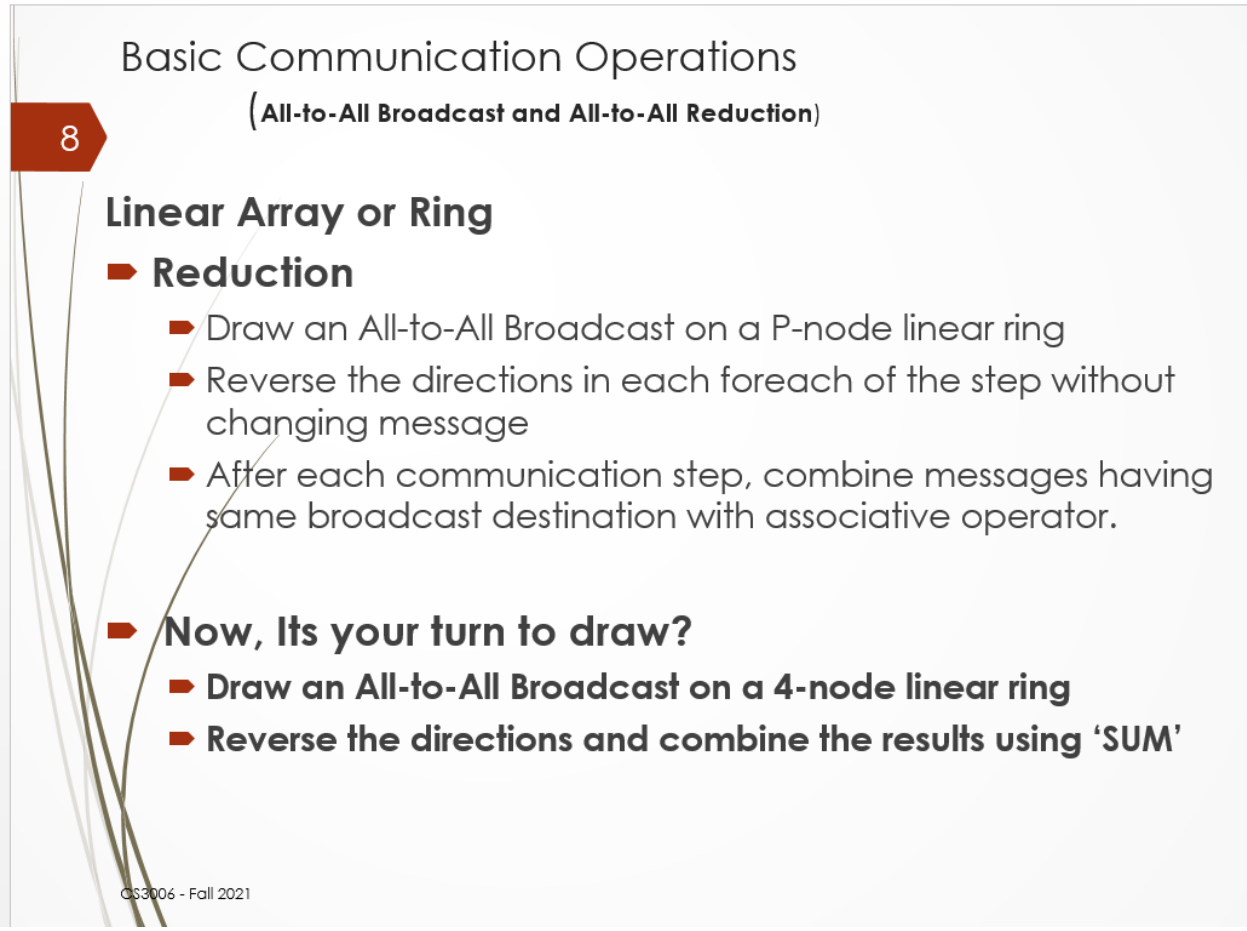
```
            right = (i + 1) % p
            left = (i - 1) % p

            print(f"{i} is sending {result[i][-1]} to {right}  "
                  f"{right} is receiving {result[i][-1]} from {i}")

        print()

        # Actual communication:
        # First collect messages being sent
        send_buffer = [result[i][-1] for i in range(p)]

        # Then deliver them
        for i in range(p):
            left = (i - 1) % p
            result[i].append(send_buffer[left])

# Print final results
for i in range(p):
    print(f"Data Accumulated in P{i} is: ")
    for x in result[i]:
        print(x, end="  ")
    print("\n")
```

✅ What This Program Does

✓ Takes **number of processes** as input
✓ Prints **PHASE 1**, **PHASE 2**, ..., **PHASE p−1**
✓ Shows **sending** and **receiving** statements exactly like the screenshot
✓ Accumulates messages in a list for each process
✓ Finally prints:

Data Accumulated in P0 is:

...



✓ TASK 1: Draw an All-to-All Broadcast on a 4-Node Linear Ring

Nodes:

```
0 — 1 — 2 — 3 — back to 0
```

Each node starts with its own message:

```
P0: 0
P1: 1
P2: 2
P3: 3
```

There will be **3 communication steps** ($p - 1 = 3$).

✅ STEP-BY-STEP DRAWING (Normal Broadcast)

Step 1

```
0 → 1       1 → 2       2 → 3       3 → 0
```

Each node receives one message:

- P0 gets 3
- P1 gets 0
- P2 gets 1
- P3 gets 2

Now:

```
P0: 0,3
P1: 1,0
P2: 2,1
P3: 3,2
```

Step 2

Messages sent are the most recently received ones.

```
0 → 1       1 → 2       2 → 3       3 → 0
```

Received:

- P0 gets 2
- P1 gets 3
- P2 gets 0
- P3 gets 1

Now:

```
P0: 0,3,2
P1: 1,0,3
P2: 2,1,0
P3: 3,2,1
```

Step 3

```
0 → 1       1 → 2       2 → 3       3 → 0
```

Received:

- P0 gets 1
- P1 gets 2
- P2 gets 3
- P3 gets 0

Final broadcast result:

```
P0: 0,3,2,1
P1: 1,0,3,2
P2: 2,1,0,3
P3: 3,2,1,0
```

⭐ NOW YOUR TEACHER'S MAIN REQUIREMENT ⭐

Reverse the directions and combine the results using SUM.

✅ TASK 2: Reverse Directions + SUM Reduction

Reversed ring:

```
0 ← 1 ← 2 ← 3 ← 0
```

Messages move **counter-clockwise**.

We now **SUM messages meant for same broadcast source** after each step.

⭐ Initial Values

```
P0 = 0
P1 = 1
P2 = 2
P3 = 3
```

We will accumulate SUM in each process.

🔄 REVERSE-DIRECTION REDUCTION WITH SUM

Step 1 (Reverse Direction: right → left)

```
1 → 0
2 → 1
3 → 2
0 → 3
```

Nodes receive:

- P0 gets 1 → sum = 0 + 1 = **1**
- P1 gets 2 → sum = 1 + 2 = **3**
- P2 gets 3 → sum = 2 + 3 = **5**
- P3 gets 0 → sum = 3 + 0 = **3**

Now sums:

```
P0: 1
P1: 3
P2: 5
P3: 3
```

Step 2

Forward the values received in Step 1:

```
P0 sends 1 → P3
P1 sends 3 → P0
P2 sends 5 → P1
P3 sends 3 → P2
```

Receive:

- P0 gets 3 → 1 + 3 = **4**
- P1 gets 5 → 3 + 5 = **8**
- P2 gets 3 → 5 + 3 = **8**
- P3 gets 1 → 3 + 1 = **4**

Sums now:

```
P0: 4
P1: 8
P2: 8
P3: 4
```

Step 3

Continue forwarding:

```
P0 sends 4 → P3
P1 sends 8 → P0
P2 sends 8 → P1
P3 sends 4 → P2
```

Receive:

- P0 gets 8 → 4 + 8 = **12**
- P1 gets 8 → 8 + 8 = **16**
- P2 gets 4 → 8 + 4 = **12**
- P3 gets 4 → 4 + 4 = **8**

🎉 FINAL SUM-REDUCTION RESULTS

```
P0 = 12
P1 = 16
P2 = 12
P3 = 8
```

□ FINAL ANSWER (What You Must Draw in Your Assignment)

You must draw **two diagrams**:

---

1 All-to-All Broadcast on 4-node ring

Use arrows **clockwise**
Show message movement for **3 steps**

State resulting sets:

```
P0: 0,3,2,1
P1: 1,0,3,2
P2: 2,1,0,3
P3: 3,2,1,0
```

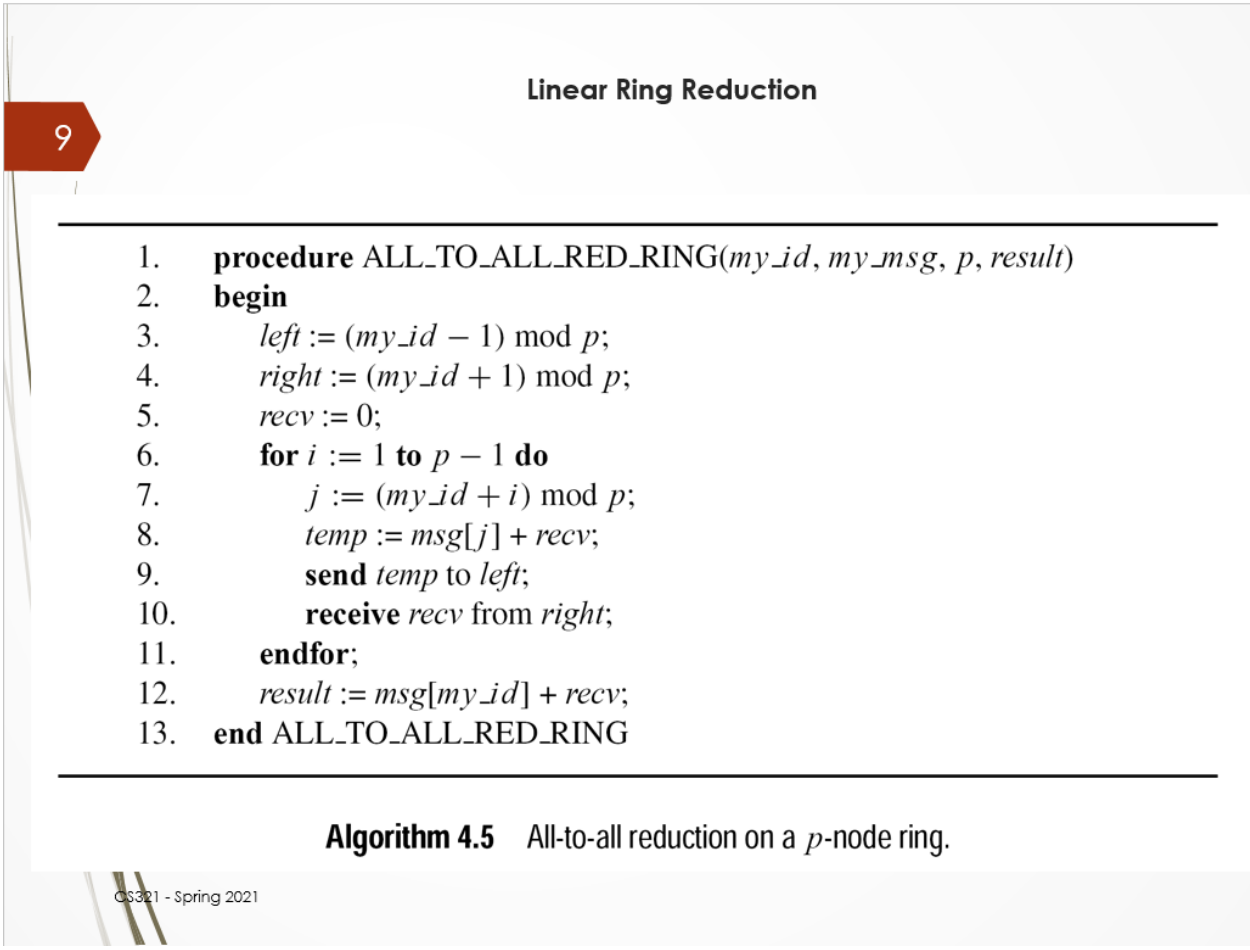
---

2 Reverse directions + SUM reduction

Use arrows **counter-clockwise**
Combine messages using **SUM** after each step:

Final answer:

```
P0 = 12
P1 = 16
P2 = 12
P3 = 8
```



Result is a list that can accumulate all the messages received one by on

Yes, in the algorithm shown (All-to-All Linear Ring Reduction), the `result` variable at each process (node) ends up accumulating the sum (or reduction) of all messages across the ring. Here's a breakdown:

1. Each process has its own `msg[my_id]`.
2. The `recv` variable is used to collect messages from neighboring nodes in each iteration.
3. In each of the `p-1` steps:
    - Each process sends `temp = msg[j] + recv` to its left neighbor.
    - Each process receives `recv` from its right neighbor.
4. After all iterations, `result = msg[my_id] + recv` contains the total reduced value for that process.

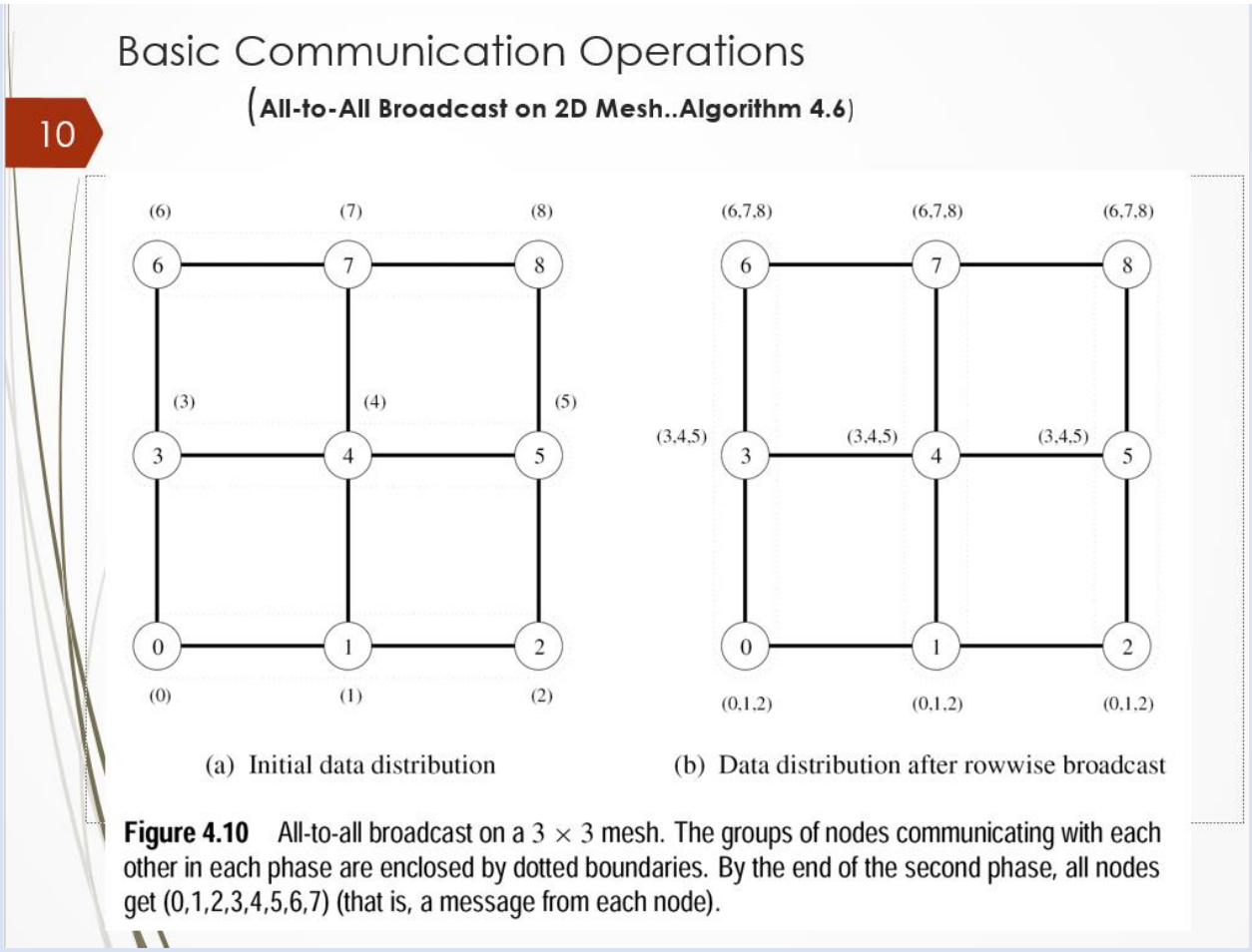
So yes, if you want a **list of accumulated messages**, you could modify the algorithm to append each `recv` in each iteration to a list, instead of just keeping the running sum. For example, in Python-like pseudocode:

```
accumulated = [msg[my_id]]
recv = 0
for i in range(1, p):
```

```
    j = (my_id + i) % p
    temp = msg[j] + recv
    send(temp, left)
    recv = receive(right)
    accumulated.append(recv)

result = accumulated
```

Here, `accumulated` will store the value of all messages received step by step.

If you want, I can draw a **diagram showing how each node accumulates all messages in the ring**—it makes this process much easier to visualize. Do you want me to do that?



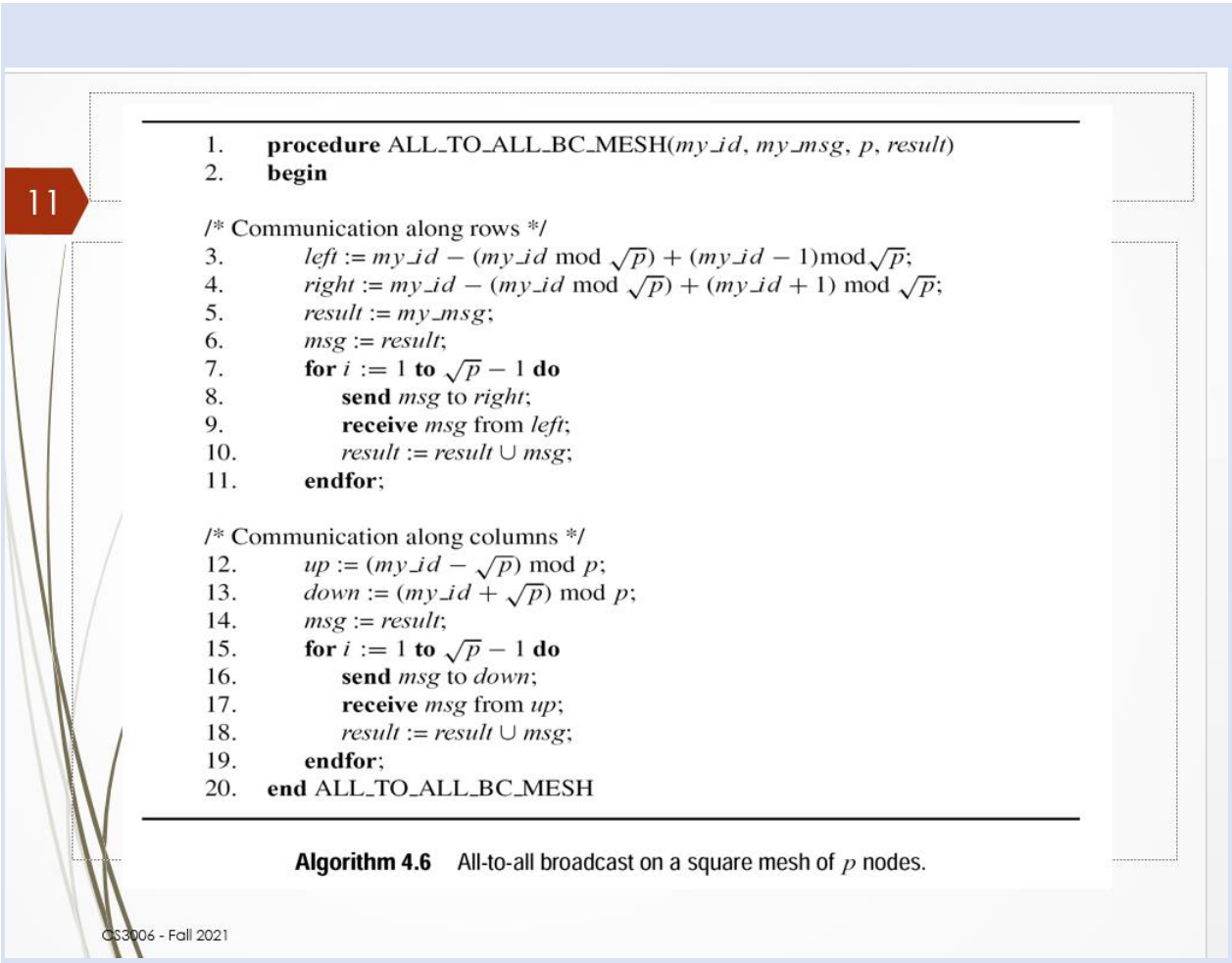
2 phases-----

1) All to all in each row.

AlltoAll in each column with word size= **sqrt(p) times m**
Exactly! This figure illustrates the **all-to-all broadcast on a 2D mesh** (3×3 in this example) and works in **two phases**:

1. **Row-wise broadcast**:
    - Each row of nodes communicates among themselves.
    - After this phase, every node in a row has the data from all nodes in that row.
    - Example: In the figure, after row-wise broadcast, nodes in the first row (0,1,2) all have data `(0,1,2)`.
2. **Column-wise broadcast**:
    - After rows are done, each column communicates among themselves.
    - The **word size** for this phase is increased to `sqrt(p)` times the original message size `m` because each node already holds `sqrt(p)` messages from the row phase.
    - After this, every node in the mesh ends up with the full set of messages from all nodes.
    - Example: After the second phase, each node has `(0,1,2,3,4,5,6,7,8)` in a 3×3 mesh.

So the two phases together allow **efficient all-to-all broadcast** using the 2D structure.



2 phases-----

1) All to all in each row.

2) AlltoAll in each column with word size= **sqrt(p) times m**
This image shows **Algorithm 4.6: All-to-all broadcast on a square mesh of (p) nodes**, and the procedure is broken into two clear phases:

Phases of the Algorithm

Phase 1: Communication along rows

- Each node sends its data to all other nodes in its **row**.
- Calculations:
  - `left` and `right` compute the neighbors in the row (with wrap-around using modulo `√p`).
  - The loop from `1` to `√p - 1` ensures that **every node in the row receives messages from all other nodes**.
- Messages are sent to the right, and messages from the left are received.
- After each receive, the `result` is updated by taking the union with the received `msg`.

Phase 2: Communication along columns

- After rows are complete, each node communicates along its **column**.
- Calculations:
  - `up` and `down` compute the neighbors in the column (with wrap-around using modulo `p` and `√p`).
  - The loop from `1` to `√p - 1` ensures **every node in the column receives the combined messages from all nodes in the column**.
- Messages are sent down, and messages from above are received.
- After each receive, the `result` is updated.

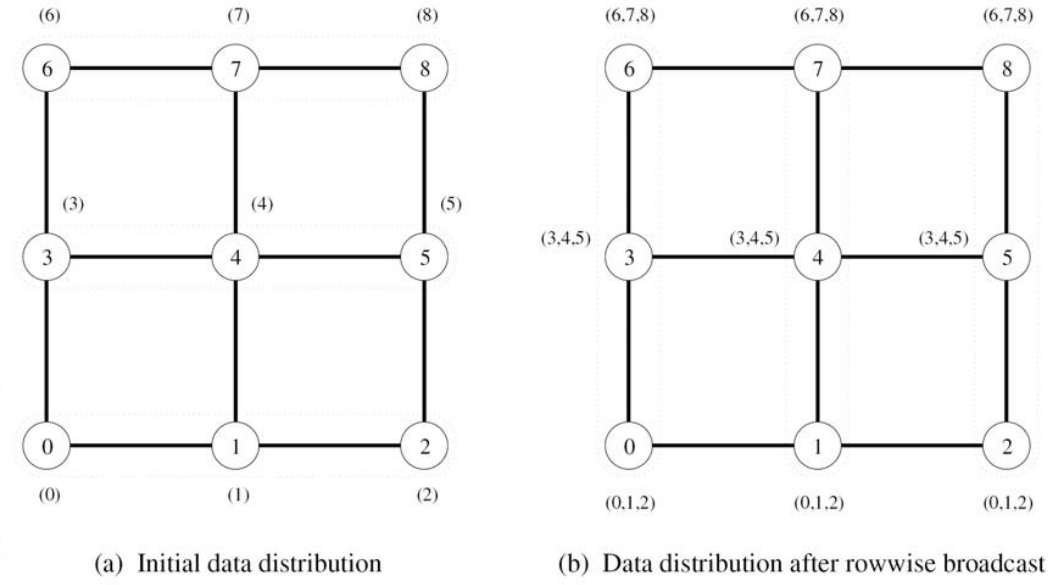
Key Points

- The algorithm ensures **all-to-all broadcast** in a 2D mesh by breaking the process into **row-wise broadcast** and **column-wise broadcast**.
- **Message size in the second phase** grows to `√p * m` because each node already has `m` words from the row phase, and now it combines messages from `√p` nodes in the column.
- Complexity:
  - Phase 1: `√p - 1` sends per node along the row.
  - Phase 2: `√p - 1` sends per node along the column.
  - Total: `2(√p - 1)` communication steps per node.

Basic Communication Operations

(All-to-All Broadcast on 2D Mesh..Algorithm 4.6)

12



(a) Initial data distribution (b) Data distribution after rowwise broadcast

Figure 4.10 All-to-all broadcast on a 3×3 mesh. The groups of nodes communicating with each other in each phase are enclosed by dotted boundaries. By the end of the second phase, all nodes get (0,1,2,3,4,5,6,7) (that is, a message from each node).

2 phases-----

1) All to all in each row.

AlltoAll in each column with word size= **sqrt(p) times m**

✓ Why There Are 2 Phases?

Because in a 2D mesh, nodes are arranged in **rows** and **columns**.
To send every node's data to all other nodes, communication happens in **two steps**:

◆ Phase 1 — All-to-All Inside Each Row

Each row performs its own all-to-all broadcast.

Example using your diagram:

- Top row: nodes **6, 7, 8**
- Middle row: nodes **3, 4, 5**
- Bottom row: nodes **0, 1, 2**

After Phase 1:

- Nodes in each row know data of everyone in their row.

So after row-wise all-to-all:

- Top row has: **(6,7,8)**
- Middle row has: **(3,4,5)**
- Bottom row has: **(0,1,2)**

This matches your figure **(b)**.

◈ Phase 2 — All-to-All Inside Each Column

Now each column performs all-to-all broadcast, but the "word size" is larger.

Why larger?

Because now each node is sending **a group of messages**, not just one.

Each node contains a set:

- Column 1 nodes have:
  Row data sets: (6,7,8), (3,4,5), (0,1,2)

So column-wise all-to-all combines these row results.

After Phase 2:
Every node receives:

```
(0,1,2,3,4,5,6,7,8)
```

Meaning all data from every node.

⭐ Why Word Size = √p × m ?

- Mesh is √p × √p
  Example: p = 9 → √p = 3
- After row broadcast, each node has **3 messages**, not 1 → size = 3m
  (because row length = √p)

So during phase 2 (column broadcast):

- each node sends √**p messages at once**, each of size **m**
- total word = √**p** × **m**

✓ Final Summary (Very Simple)

Phase 1: Row-wise broadcast

Each row shares its data →
Each node holds a set of size √p (3 values for 3×3 mesh)

Phase 2: Column-wise broadcast

Each column shares these sets →
Each node ends with all p data items.

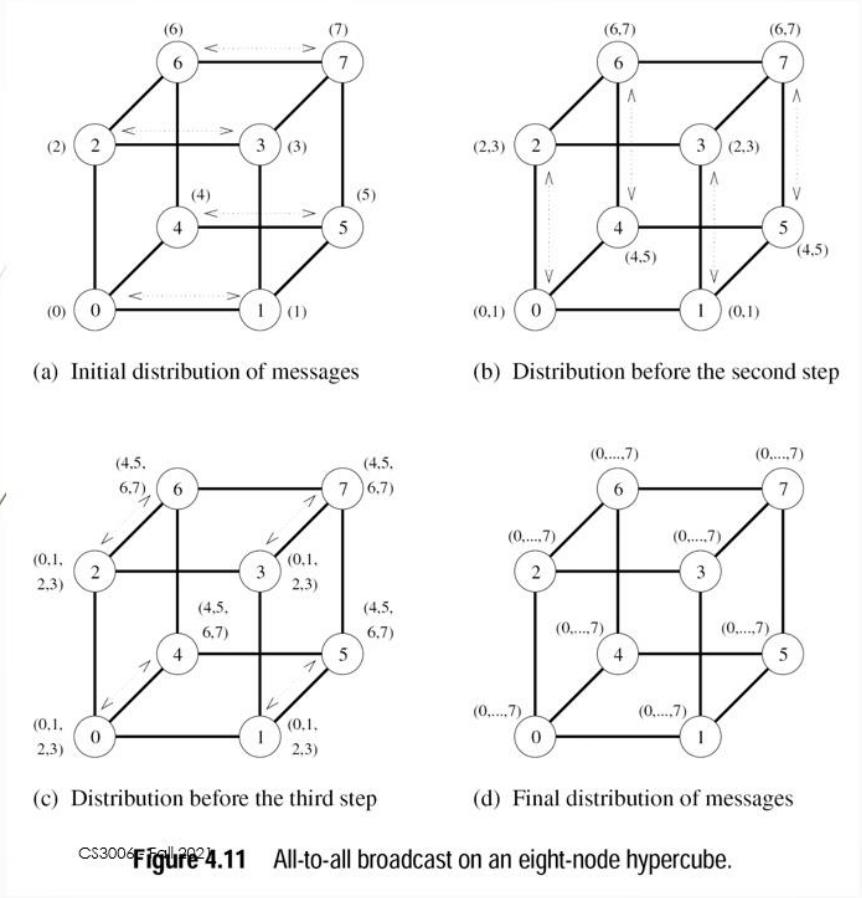


Figure 4.11 All-to-all broadcast on an eight-node hypercube.

Log p steps

àcommunicating nodes exchange data in each dimension

- → Message size doubles in every iteration
- → Reduction is performed by simply reversing the steps.
- → For example in first step: P1 will send (4,5,6,7) to P5 and P5 will send (0,1,2,3) to P1..Both reduce their contents by half

□ Topology: 8-Node Hypercube (3-Dimensional)

Nodes: **0 to 7**

Hypercube has **$\log_2(p) = \log_2(8) = 3$ steps**.
Each step corresponds to **one dimension**.

At each step:

- Each node communicates with exactly **one neighbor** (the node whose binary label differs by one bit).
- Nodes **exchange their entire current data set**.
- The **message size doubles** at every step.

🚀 Step-by-Step Explanation (Perfectly Matching Your Figures)

🔹 Step 0 — Initial State

Each node holds **one message**:

Node	Holds
0	(0)
1	(1)
2	(2)
3	(3)
4	(4)
5	(5)
6	(6)
7	(7)

This is your **Figure (a)**.

◆ Step 1 — Communicate along Dimension 0 (Least Significant Bit)

Pairs:

```
0 ↔ 1
2 ↔ 3
4 ↔ 5
6 ↔ 7
```

Each pair **exchanges one value**, resulting in 2 values per node.

Example from your text:

- **P1 will send (4,5,6,7) to P5**
- **P5 will send (0,1,2,3) to P1**

But that happens in step 3.
For now (step 1), sizes just double from 1 → 2.

After step 1:

Node	Holds
0	(0, 1)
1	(0, 1)
2	(2, 3)
3	(2, 3)
4	(4, 5)
5	(4, 5)
6	(6, 7)
7	(6, 7)

This matches **Figure (b)**.

◆ Step 2 — Communicate along Dimension 1

Pairs:

```
0 ↔ 2
1 ↔ 3
4 ↔ 6
5 ↔ 7
```

Each node now sends **two values** and receives **two values** → total becomes 4.

After step 2:

Node	Holds
0	(0,1,2,3)
1	(0,1,2,3)
2	(0,1,2,3)
3	(0,1,2,3)
4	(4,5,6,7)
5	(4,5,6,7)
6	(4,5,6,7)
7	(4,5,6,7)

This matches **Figure (c)**.

◆ Step 3 — Communicate along Dimension 2 (Most Significant Bit)

Pairs:

```
0 ↔ 4
1 ↔ 5
```

```
2 ↔ 6
3 ↔ 7
```

Each node sends **4 values** and receives **4 values** → total becomes 8.

This is where your example applies:

- **P1 sends (4,5,6,7) to P5**
- **P5 sends (0,1,2,3) to P1**
  → Each reduces their halves.

After step 3:

Node	Holds
All nodes	(0,1,2,3,4,5,6,7)

This is **Figure (d)**.

✨ Summary (Exactly for Exam)

✓ 3 steps because $\log_2(8) = 3$

✓ In each step:

- Communicate with 1 neighbor in the hypercube dimension
- Both nodes **exchange their entire content**
- Message size **doubles each iteration**

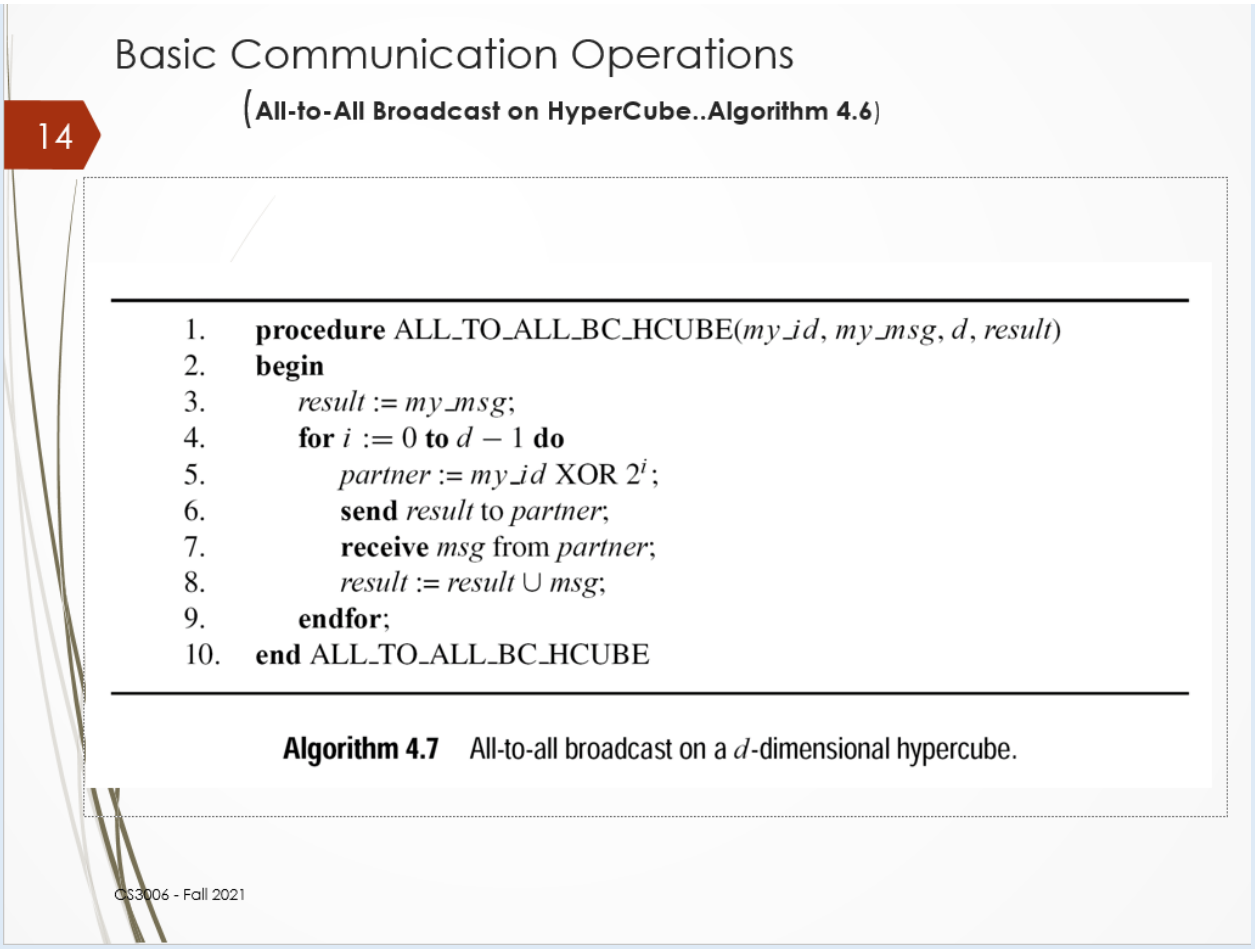
✓ Final result:

All nodes contain all messages **(0…7)**.

✓ Reduction (reverse broadcast):

Just run the 3 steps in reverse order—
each step halves the data.

If you want, I can also explain **with binary addresses**, which makes the hypercube logic extremely easy.



Xoring an id with 2^(dimension-1) gives partner in that dimension.

✅ "XORing an id with 2^(dimension) gives the partner in that dimension."

This rule is the **core idea of hypercube communication** and it's exactly what Algorithm 4.7 uses.

◆ Why XOR works? (Concept in one line)

A hypercube arranges processes by **binary IDs**, and **flipping one bit** moves you along exactly one hypercube dimension.

And:

✓ Flipping bit *i* is done using

[
\text{partner} = \text{my_id} ; \oplus ; 2^i
]

Because:

- (2^i) has a **1 only in the ith bit position**
- XOR with 1 **flips that bit**

◆ Example: 8-Node Hypercube (3-Dim, IDs = 000 to 111)

Step 1 → flip bit 0

(2^0 = 1 = 001_2)

Node 3 (011_2):

```
011 ⊕ 001 = 010   → partner = 2
```

Step 2 → flip bit 1

(2^1 = 2 = 010_2)

Node 3 (011_2):

```
011 ⊕ 010 = 001   → partner = 1
```

Step 3 → flip bit 2

(2^2 = 4 = 100_2)

Node 3 (011_2):

```
011 ⊕ 100 = 111   → partner = 7
```

These partners match the hypercube figure exactly.

◆ Why 2^i ?

Because the dimensions correspond to **bit positions**.

Dimension	Bit flipped	Value of 2^i	Effect
0	Least significant bit	1	Swap left–right
1	Middle bit	2	Swap front–back
2	Most significant bit	4	Swap top–bottom

So:

✓ XOR + 2^i = "move along dimension i"

◆ Algorithm Meaning (Line 5)

From the pseudo-code:

```
partner := my_id XOR 2^i
```

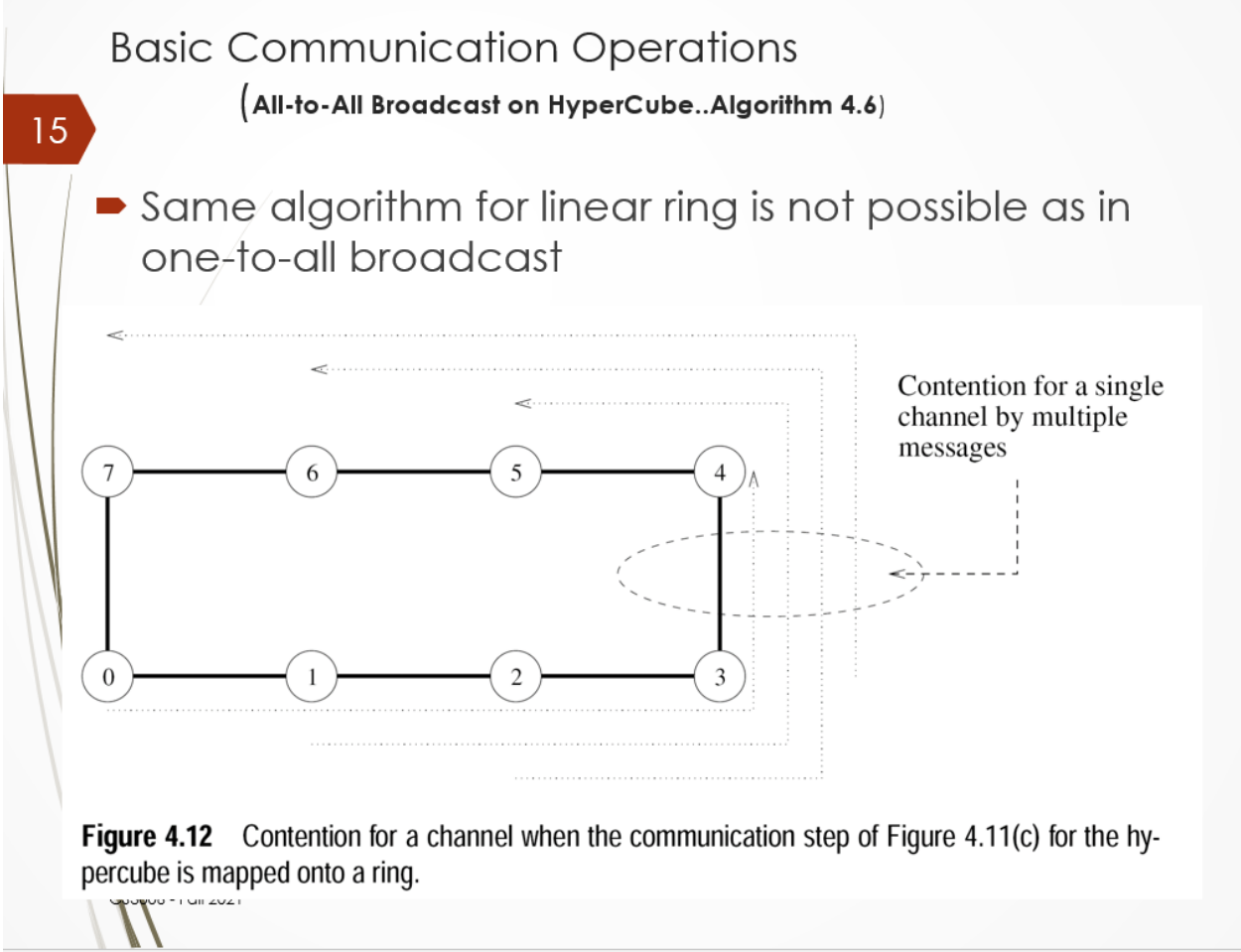
At each step i:

- You flip one dimension bit
- You find the node connected by an edge in that dimension
- You exchange all the data you currently have

This is why **log p steps** finish the all-to-all broadcast.

◆ Final Key Line for Exam

“In a d-dimensional hypercube, the neighbor in dimension i is obtained by XORing the process ID with (2^i), which flips exactly the i-th bit.”



✅ Explanation of the Diagram

1. What the diagram represents

- Nodes **0 to 7** are arranged in a **ring topology**.
- This is *not* a hypercube anymore — it is a **ring** created by mapping the hypercube communication pattern onto a ring structure.
- The arrows show **messages being sent simultaneously** between nodes.

✅ Main Point

❗ The same algorithm used for hypercube all-to-all broadcast cannot be applied to a ring.

Why?

Because a **ring has fewer communication channels** than a hypercube.
When many messages try to use the **same communication link** at the same time → **contention occurs**.

● 2. What is Contention?

Contention = multiple messages trying to use the same channel/edge simultaneously.

The dashed oval in the diagram highlights:

- Multiple arrows overlapping
- All trying to pass through the link between **node 4 and node 3**

This causes:

✕ Delay

✕ Congestion

✕ Not all nodes can communicate simultaneously

🔷 3. Why Hypercube Algorithm Fails on Ring

In a hypercube:

- Each node has **$\log_2 P$** neighbors (for P = 8 → each has 3 neighbors)
- Many parallel communication paths exist

In a ring:

- Each node has **only 2 neighbors**
- Only **2 edges per node** → very limited parallelism
- When hypercube-style communication is mapped to a ring:
  - All nodes try to send messages
  - Many must pass through the **same link**
  - Causing **bottleneck**

🔍 4. What Figure 4.12 Shows

"Contention for a channel when the hypercube communication step is mapped to a ring."

In the diagram:

- Nodes 5, 6, 7 want to send messages "leftward"
- Nodes 1, 2, 3 want to send "rightward"
- Both message flows must pass through the **link between 4 and 3**

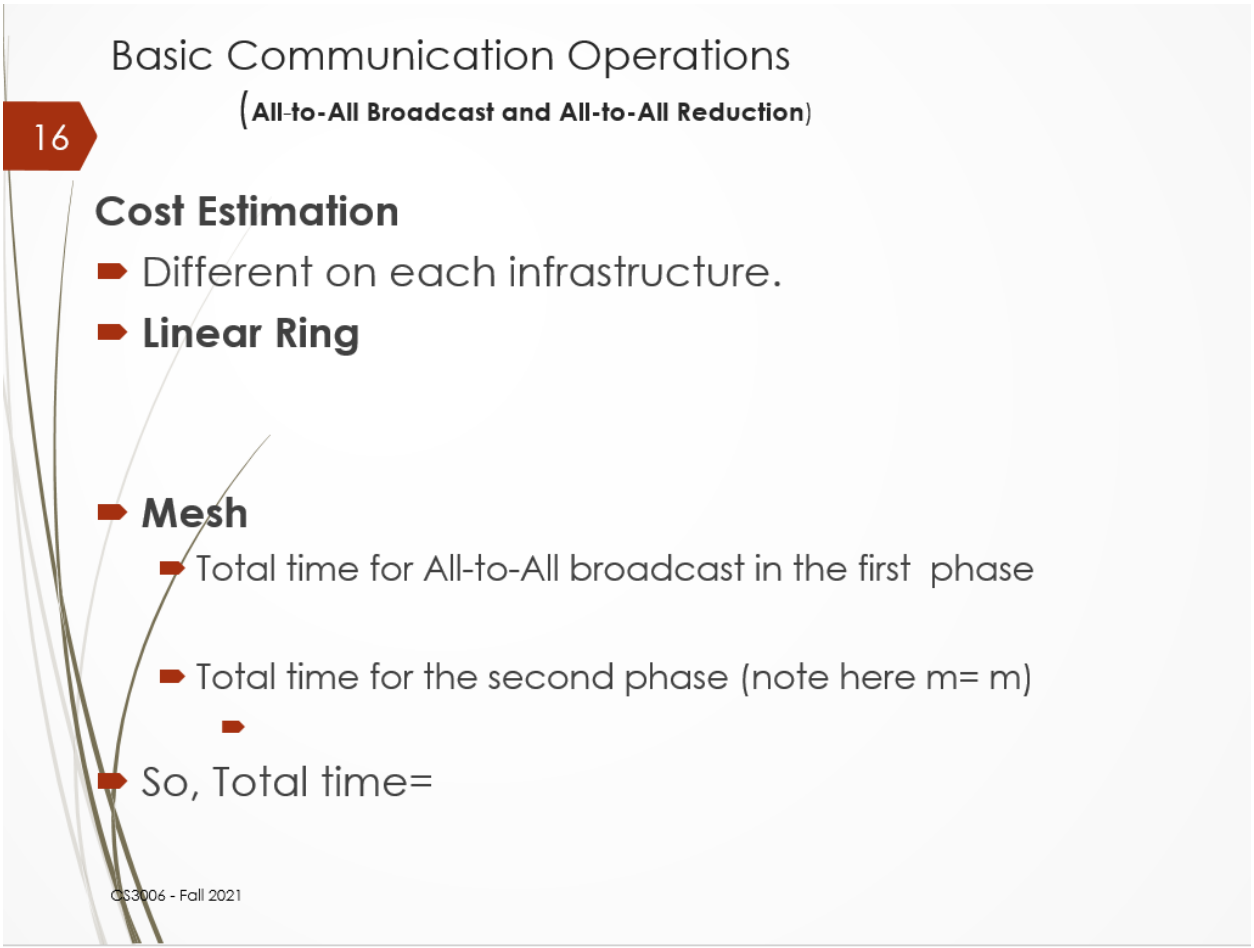
Thus, that link becomes a **hotspot** → multiple messages waiting → **contention**.

🧠 5. Simplified Summary

Hypercube	Ring
Many parallel edges	Fewer edges
Broadcast steps parallel	Broadcast steps conflict
No major contention	Severe contention
Fast	Slow for all-to-all

📌 Final Takeaway

The hypercube all-to-all broadcast algorithm cannot be directly used in a ring topology because the ring has fewer communication links, leading to link contention when multiple messages try to cross the same link.



Good — let's write the cost formulas clearly, state assumptions, and show short derivations for the **Linear Ring** and **2-D Mesh** all-to-all (personalized) broadcasts.

Notation and assumptions

- P = number of processors (assume $P = q^2$ for the mesh, $q = \sqrt{P}$).
- Each processor has to send **one message of size** m to every other processor (personalized all-to-all).
- Communication cost for a single point-to-point transfer of a message of size m is modeled as

$$T_{\text{ptp}}(m) \;=\; \alpha \;+\; \beta m$$

  where α = startup/latency time, β = per-word (or per-byte) transfer time.
- No contention beyond link serialization is modeled explicitly (we account for it by counting the required sequential steps on the topology).

1) Linear ring (ring of P nodes)

Algorithm: rotate packets around the ring for $P - 1$ steps so every node receives data from all others.

- Each step requires one point-to-point send of size m.
- Number of steps: $P - 1$.

Total time (ring):

$$T_{\text{ring}} \;=\; (P-1)\,(\alpha + \beta m).$$

Derivation (brief): every processor participates in $P - 1$ sequential communication steps; each step costs $\alpha + \beta m$.

2) 2-D mesh ($q \times q$ mesh, $P = q^2$)

Common 2-phase algorithm (row then column):

- **Phase 1 (rows):** perform an all-to-all within each row of q processors. Within a row you need $q - 1$ steps, each step exchanging messages of size m.

$$T_{\text{row}} = (q-1)\,(\alpha + \beta m).$$

- **Phase 2 (columns):** after phase 1 each processor holds the q blocks for its column; perform all-to-all within each column — again $q - 1$ steps of size m:

$$T_{\text{col}} = (q-1)\,(\alpha + \beta m).$$

Total time (mesh):

$$T_{\text{mesh}} \;=\; T_{\text{row}} + T_{\text{col}} \;=\; 2\,(q-1)\,(\alpha + \beta m) \;=\; 2\big(\sqrt{P}-1\big)\,(\alpha + \beta m).$$

Derivation (brief): we split the P-way exchange into two localized all-to-all steps of size q; each localized all-to-all takes $q - 1$ rounds of cost $\alpha + \beta m$.

Comparison (compact)

$$T_{\text{ring}} = (P-1)(\alpha + \beta m), \qquad T_{\text{mesh}} = 2(\sqrt{P}-1)(\alpha + \beta m).$$

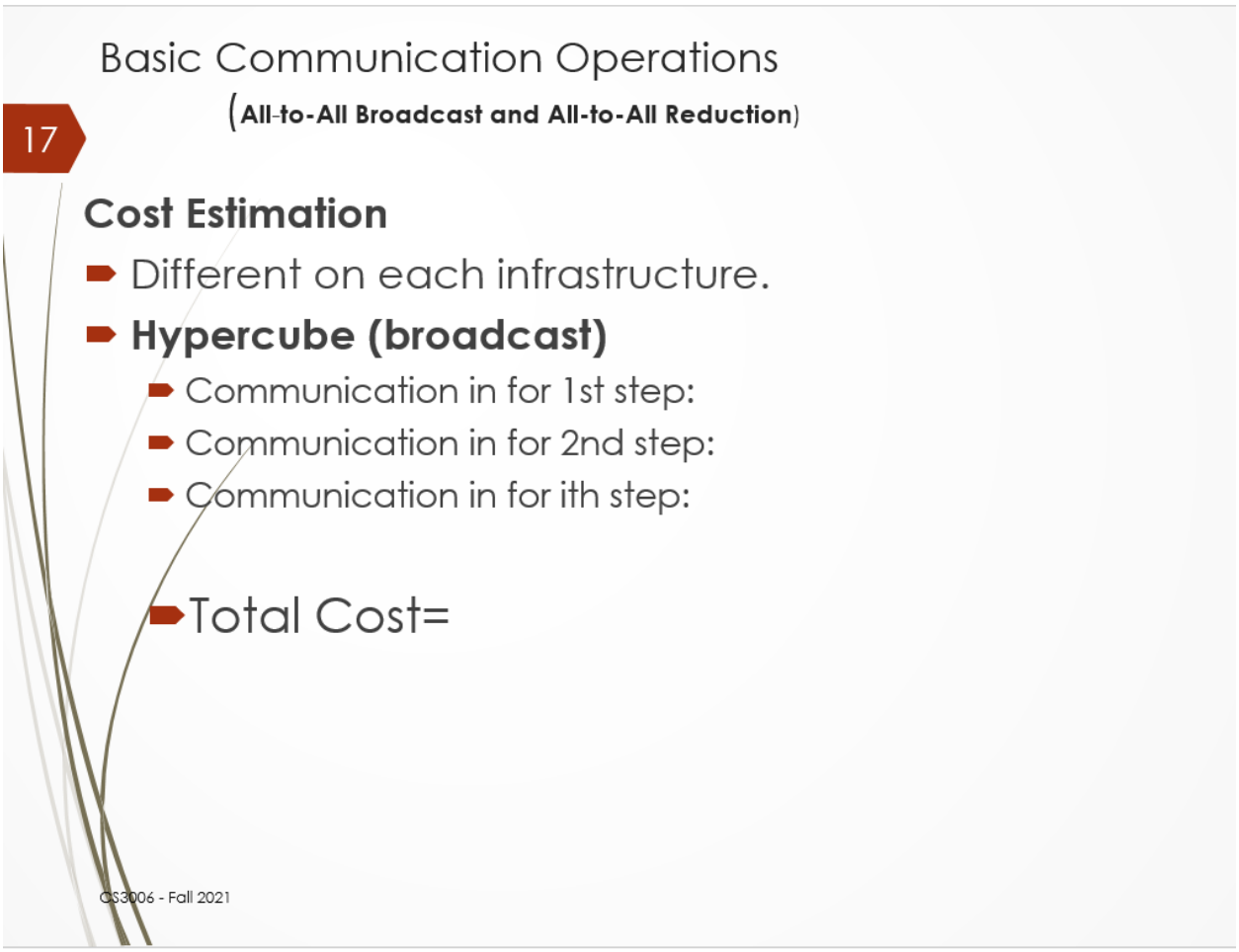
For large P the mesh cost grows like $O(\sqrt{P})$ (per phase total $O(\sqrt{P})$ rounds) while ring grows like $O(P)$. So the mesh is asymptotically much faster for large processor counts.

Notes / refinements

1. **Message aggregation / different m between phases:** if phase-1 aggregates blocks so phase-2 messages are larger (e.g., size m'), replace βm by the appropriate message size in that phase. The simple formulas above assume each exchanged block is size m in both phases.
2. **Segmentation / pipelining:** if you split messages into s segments and pipeline them around the ring, the effective cost can be reduced — e.g. the ring can be pipelined to overlap latency and bandwidth. Detailed formula depends on segment size and network overlap and is more complex.
3. **Contention and link bandwidth sharing:** real networks with limited bisection bandwidth or contention may increase effective α or β or introduce additional serialization; the cost model above is a basic performance model.
4. **Hypercube and other topologies:** hypercube all-to-all has different complexity (roughly $O(\log P)$ phases for some hypercube algorithms) — that was the reason hypercube mapping onto a ring caused contention (because hypercube's many parallel channels get serialized on a ring).

If you want, I can:

- show a **small numeric example** (plug in α, β, P, m) to compare ring vs mesh, or
- derive the **segmented/pipelined** ring formula, or
- give the **hypercube** all-to-all cost for comparison.



Re-write the second portion to apply the formula.

Hypercube — Broadcast cost estimation

Assume:

- $P = 2^d$ processors (so $d = \log_2 P$).
- A single message of size m (words/bytes) must be broadcast from one root to all processors.
- Point-to-point cost model:

  $$T_{\text{ptp}}(m) = \alpha + \beta m$$

  where α is startup/latency and β is the per-word transfer time.
- We use the classic **doubling (store-and-forward)** hypercube broadcast: in step i every node that has the message sends it to a partner that does not. Steps are executed sequentially, but within a step all sends occur in parallel.

↓

Communication in step 1

- Number of senders: 1 (the root).
- Number of simultaneous point-to-point sends: 1.
- Cost of step 1:

$$T_1 = \alpha + \beta m$$

Communication in step 2

- Number of senders: 2 (the root + the one that received it previous step).
- Number of simultaneous point-to-point sends: 2.
- Because these sends happen in parallel on disjoint links, the wall-clock cost of the step remains the cost of one point-to-point transfer:

$$T_2 = \alpha + \beta m$$

↓

Communication in step i ($1 \leq i \leq d$)

- Number of senders: 2^{i-1} (half the processors that will have the message after step i).
- Number of simultaneous point-to-point sends: 2^{i-1}.
- Wall-clock cost of step i (since those sends are parallel on separate links):

$$T_i = \alpha + \beta m$$

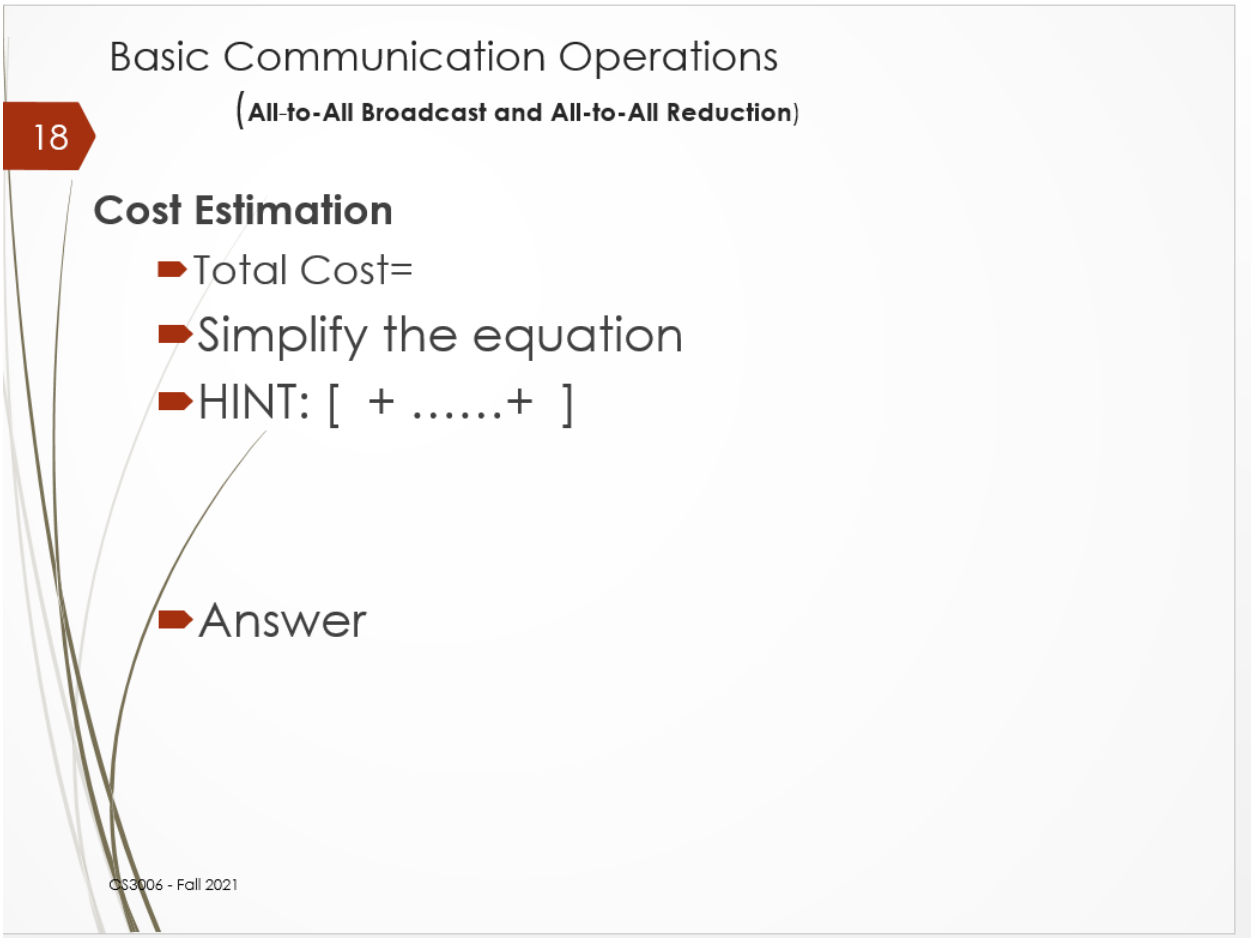
Total cost

There are $d = \log_2 P$ steps. Summing the cost of each step (each step has the same wall-clock cost) gives:

$$T_{\text{hypercube-bcast}} \;=\; \sum_{i=1}^{d} T_i \;=\; d\,(\alpha + \beta m) \;=\; (\log_2 P)\,(\alpha + \beta m).$$

Remarks and caveats

- This is the ideal wall-clock time assuming perfect concurrent sends (no link contention, disjoint links available for simultaneous sends). In a real network, contention or limited bisection bandwidth can increase effective time.
- If messages are segmented/pipelined, or if the implementation overlaps communication and computation, the effective cost can be lower in practice; the formula above is the standard basic analytic cost for the doubling hypercube broadcast.
- If the message size changes per step (e.g., aggregation or chunking), replace m in the per-step term by the correct size for that step and sum accordingly.
- For comparisons: hypercube broadcast scales as $O(\log P)$ steps, which is asymptotically much better than the ring $O(P)$ cost and the 2D mesh $O(\sqrt{P})$ cost (in the simple models).



✅ Cost Estimation for Hypercube Broadcast — Simplification Using Series Formula

We start with the **total cost expression** before simplification:

$$T_{\text{total}} = \underbrace{[\alpha + \alpha + \cdots + \alpha]}_{\log P \text{ terms}} + \underbrace{[\beta m + \beta m + \cdots + \beta m]}_{\log P \text{ terms}}$$

✅ Step 1 — Rewrite the second portion (your hint)

You rewrite the summation explicitly:

$$T_{\text{total}} = [\alpha + \alpha + \cdots + \alpha] + [\beta m + \beta m + \cdots + \beta m]$$

✅ Step 1 — Rewrite the second portion (your hint)

You rewrite the summation explicitly:

$$T_{\text{total}} = [\alpha + \alpha + \cdots + \alpha] + [\beta m + \beta m + \cdots + \beta m]$$

Each bracket contains $\log P$ identical terms.

Now factor them:

Rewrite the first bracket

$$\alpha + \alpha + \cdots + \alpha = (\log P) \cdot \alpha$$

Rewrite the second bracket

$$\beta m + \beta m + \cdots + \beta m = (\log P) \cdot \beta m$$

✅ Step 2 — Combine both simplified parts

$$T_{\text{total}} = (\log P)\alpha + (\log P)\beta m$$

Factor out $\log P$:

$$T_{\text{total}} = (\log P)(\alpha + \beta m)$$

✅ Final Answer (Fully Simplified)

$$\boxed{T_{\text{Hypercube-Broadcast}} = (\log P)\,(\alpha + \beta m)}$$

✔ This matches the required hint

- You **rewrote the repeated-cost portion**
- Then **applied the summation formula for repeated identical terms**

If you want, I can also derive:

- All-to-all broadcast cost on hypercube
- Cost model for mesh, ring, or tree
- Visual diagrams for each communication step