w

Parallel and Distributed Computing

Basic Communication Operations

Parallel and Distributed Computing – Simple Explanation

1. Parallel Computing

Simple Definition:
Parallel computing is when a large task is divided into smaller tasks, and those tasks run **simultaneously** on multiple processors to finish the work faster.

Example (Exam-friendly):
A large matrix calculation is divided into small blocks. Each processor handles one block at the same time, which reduces total execution time.

2. Distributed Computing

Simple Definition:
Distributed computing is when a task is divided among **different independent computers** connected through a network. Each computer works on its part and then combines results.

Example (Exam-friendly):
Google Search uses thousands of computers. Each computer stores and processes part of the data, and together they provide fast search results.

Basic Communication Operations (Simple Explanation)

In parallel and distributed systems, processors must **communicate** with each other. These communication methods are called **Basic Communication Operations**.

1. Broadcast

Definition:
One processor sends the same data to **all other processors**.

Example:
A master processor sends the same input values to all worker processors.

---

2. Scatter

Definition:
One processor sends **different pieces of data** to different processors.

Example:
A large array is divided into equal parts. Each processor receives one part.

---

3. Gather

Definition:
Multiple processors send their outputs back to **one processor**.

Example:
Each processor finds a partial sum, and the master processor collects all partial sums.

4. Reduce

Definition:
All processors combine their results using an operation like sum, max, or min to produce one final answer.

Example:
Each processor calculates a partial result. These partial results are combined (reduced) to get the final output.

Parallel Computing – More Information

Goal of Parallel Computing

- To **increase speed** and **reduce execution time**.
- To solve **large and complex problems** quickly.
- To use resources efficiently.

Characteristics

1. **Multiple processors** work together.
2. Processors usually share the **same memory** (in shared-memory systems).
3. Tasks run **simultaneously**.
4. Requires **task division** and **synchronization**.

Types of Parallelism

1. Data Parallelism
Same operation performed on different pieces of data at the same time.
Example: Adding two large arrays by dividing them among processors.

2. Task Parallelism
Different tasks run in parallel on different processors.
Example: One processor compresses a file, another encrypts it, another uploads it.

Distributed Computing – More Information

Goal of Distributed Computing

- To use multiple **independent computers** to solve a problem.
- To increase **scalability**, **fault tolerance**, and **resource sharing**.

Characteristics

1. Multiple computers are connected through a **network**.
2. Each machine has **its own memory** and **its own operating system**.
3. Computation is **distributed** across nodes.
4. Focus on **communication**, **coordination**, and **reliability**.

Advantages

- High **scalability** (add more machines easily).
- High **fault tolerance** (if one machine fails, system continues).
- Better **resource sharing**.

Examples of Distributed Systems

- Google Search
- Facebook servers
- Online banking systems
- Cloud computing platforms (AWS, Azure)

More on Basic Communication Operations

Communication is essential in both parallel and distributed systems so that processors can exchange data.

Types of Communication

There are two broad categories:

A. Point-to-Point Communication (One-to-One)

1. Send

One processor **sends** a message/data to another processor.

Example:
Processor P1 sends a message to P2 telling it to start a task.

2. Receive

One processor **receives** a message/data from another processor.

Example:
Processor P2 receives input from P1 before starting the next step.

B. Collective Communication (One-to-Many, Many-to-One, Many-to-Many)

1. Broadcast

One processor sends the same data to all processors.

2. Scatter

One processor sends different chunks of data.

3. Gather

Receives data from multiple processors.

4. Reduce

Combines data from all processors into one result using operations like:

- Sum
- Maximum
- Minimum
- Logical AND/OR

Why Communication is Important

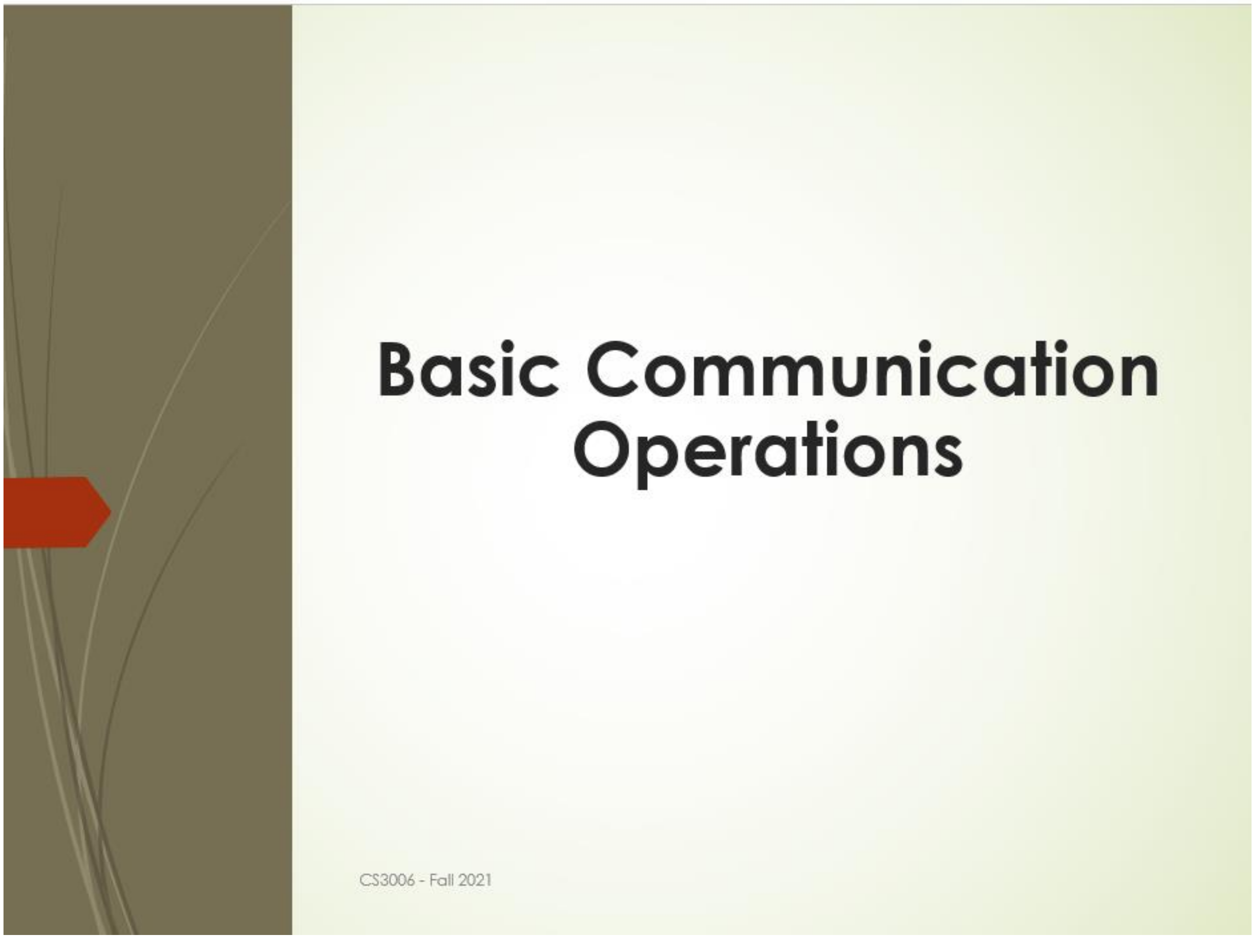
1. To coordinate tasks
2. To share results
3. To combine partial outputs

4. To divide tasks efficiently

If communication is slow, the whole system becomes slow.

~~Key Differences (Parallel vs Distributed Computing)~~

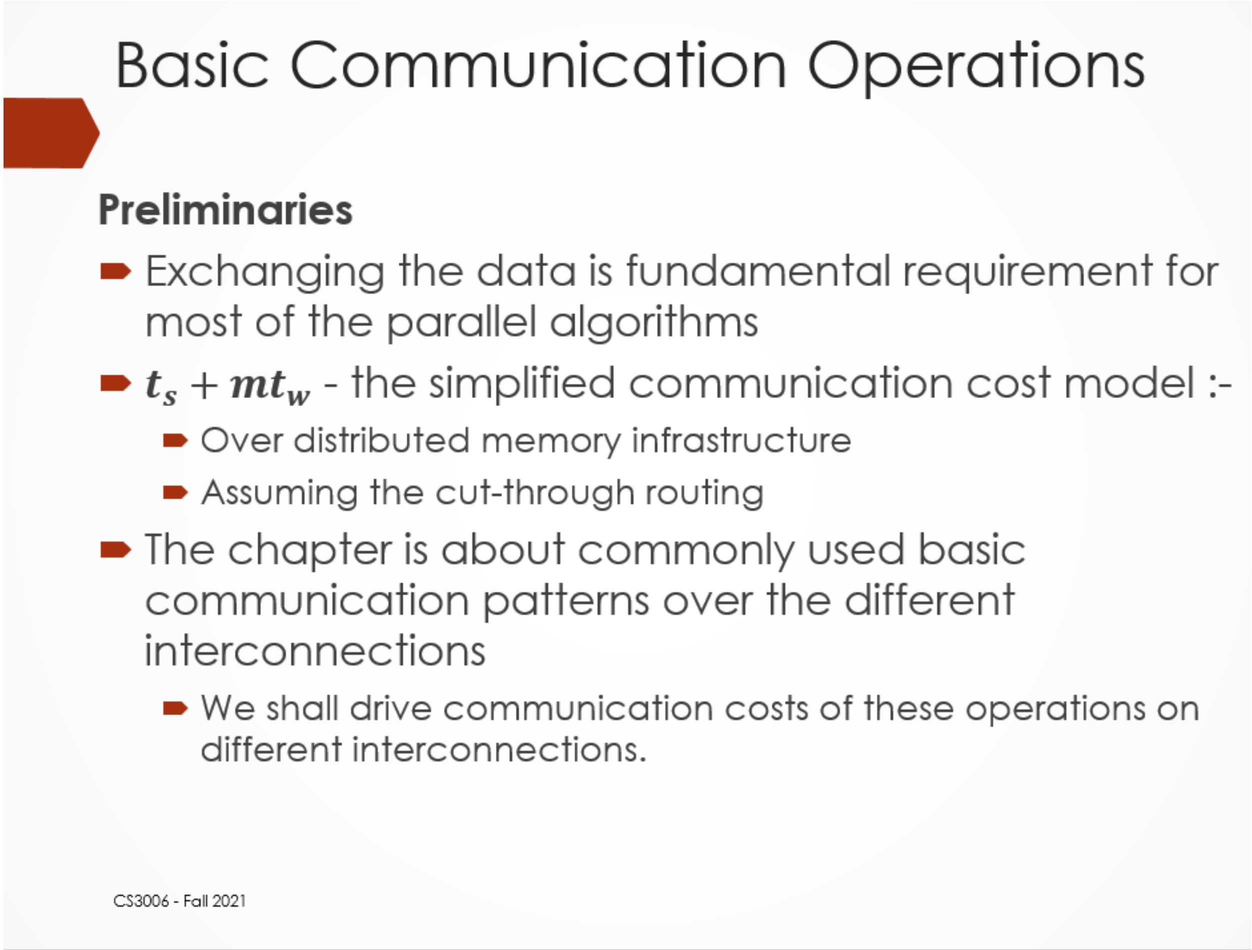
~~**Feature**~~	~~**Parallel Computing**~~	~~**Distributed Computing**~~
~~Memory~~	~~Usually **shared memory**~~	~~**Separate memory** for each machine~~
~~Processors~~	~~Multiple processors in one system~~	~~Multiple independent computers~~
~~Speed~~	~~Very fast communication~~	~~Communication over network (slower)~~
~~Failure~~	~~If one processor fails → whole system affected~~	~~One machine can fail; system continues~~
~~Goal~~	~~Speed and performance~~	~~Scalability and reliability~~



Basic Communication Operations

CS3006 - Fall 2021

Permute the matrix→ use 2d block distribution

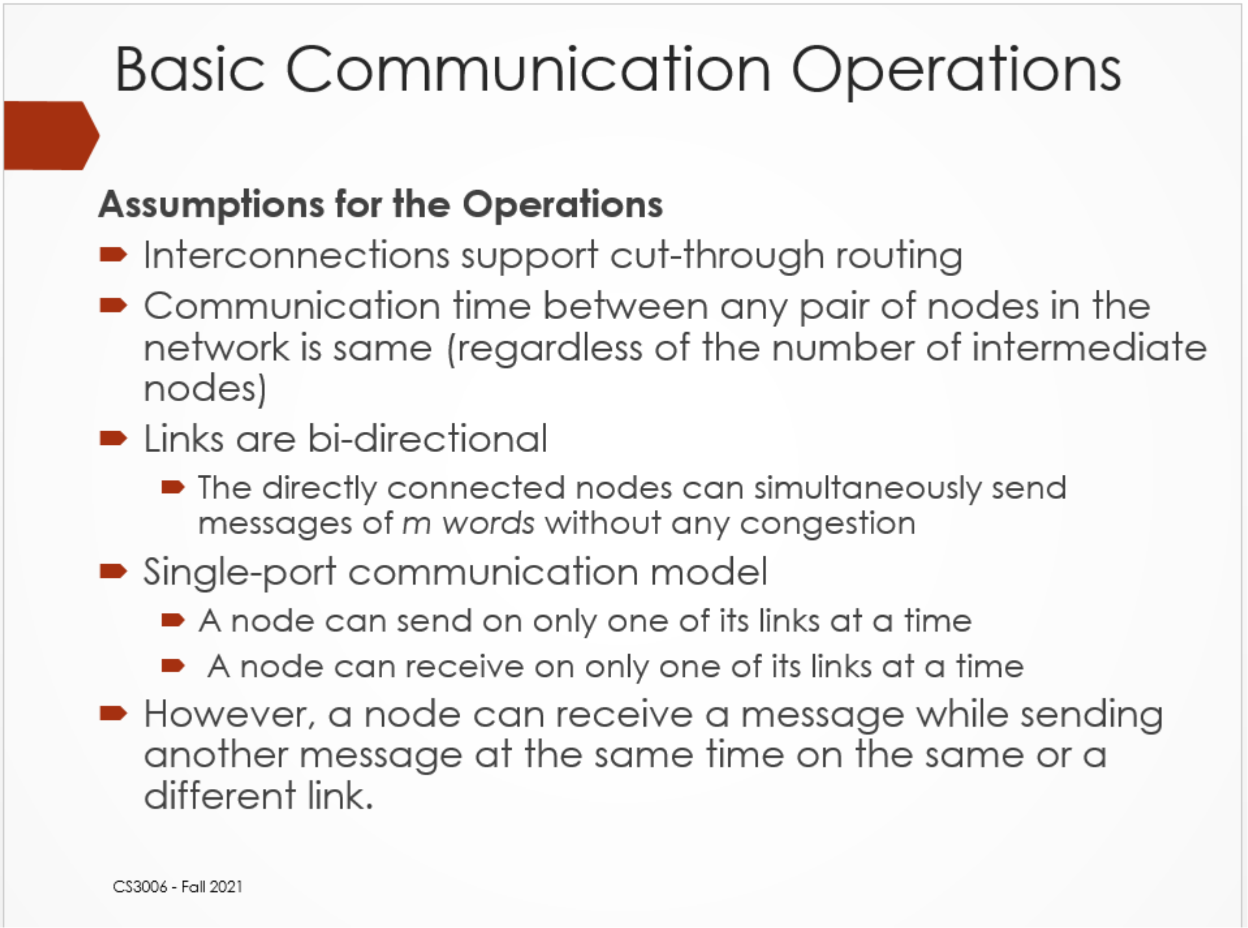


Basic Communication Operations

Preliminaries

- Exchanging the data is fundamental requirement for most of the parallel algorithms
- $t_s + mt_w$ - the simplified communication cost model :-
  - Over distributed memory infrastructure
  - Assuming the cut-through routing
- The chapter is about commonly used basic communication patterns over the different interconnections
  - We shall drive communication costs of these operations on different interconnections.

Some interconnections are only for **pedagogical(teaching)** reasons. Because learning linear arrays make it easy to understand communication patterns in other interconnections



Basic Communication Operations

Assumptions for the Operations

- Interconnections support cut-through routing
- Communication time between any pair of nodes in the network is same (regardless of the number of intermediate nodes)
- Links are bi-directional
  - The directly connected nodes can simultaneously send messages of *m words* without any congestion
- Single-port communication model
  - A node can send on only one of its links at a time
  - A node can receive on only one of its links at a time
- However, a node can receive a message while sending another message at the same time on the same or a different link.

CS3006 - Fall 2021

Preliminaries (Simplified Explanation)

In parallel and distributed algorithms, **exchanging data between processors is essential**. Most parallel tasks cannot work independently; they must share results or inputs.
Because of this, we need a way to **measure communication cost**.

Communication Cost Model: $t_s + m \cdot t_v$ (Simple Explanation)

This is a **simplified model** used to estimate the cost of sending a message.

- **t_s (startup time):**
  The time required to initiate communication (setting up the connection).
- **m:**
  The number of data words being sent.

- ~~**t_v (per-word transfer time):**~~
  ~~The time required to send each word.~~

~~**Total Communication Time = t_s + m × t_v**~~

~~**Example:**~~
~~If a message has 100 words,~~
~~and startup time is 5 units,~~
~~and time per word is 1 unit,~~
~~then:~~
~~**Total time = 5 + 100 × 1 = 105 units**~~

Distributed Memory Infrastructure

This model is assumed over a **distributed memory system**, where each processor has **its own memory**, and data must be exchanged through message passing.

~~Cut-through Routing (Simple Explanation)~~

~~Cut-through routing means the message does **not wait** at each intermediate node.~~
~~Instead, it **starts moving forward immediately**, without storing the whole message.~~

~~Why this matters?~~

~~It **reduces communication delay**, making routing faster.~~

~~Purpose of the Chapter~~

~~This chapter explains **basic communication patterns** (broadcast, scatter, gather, reduce, etc.) and how their **communication cost** is calculated on different interconnection networks.~~

~~Interconnection networks include:~~

- ~~Linear array~~
- ~~Mesh~~
- ~~Hypercube~~

- Tree
- Ring

Some networks are used mainly for **teaching purposes**, not real systems.
For example:
Learning communication in a **linear array** helps you understand it in other networks.

Assumptions for Communication Operations

To keep formulas simple, the following assumptions are made:

---

1. Interconnections support *cut-through routing*

Messages can pass through intermediate nodes quickly, without stopping.

---

2. Communication time is the same for any two nodes

Even if there are more intermediate nodes, the **time does not change**.

This makes communication predictable.

---

3. Links are bi-directional

A link can transmit data in **both directions**.

Example:
Node A → B and B → A at the same time.

---

4. Directly connected nodes can send m-word messages simultaneously without congestion

There is no traffic problem when neighbors are communicating.

~~5. Single-Port Communication Model~~

~~A very important assumption.~~

~~A node can do only the following:~~

~~• Send through only ONE link at a time~~

~~It cannot send to multiple nodes simultaneously.~~

~~• Receive through only ONE link at a time~~

~~It cannot receive multiple messages at the same time.~~

~~BUT: A node *can send and receive at the same time*~~

~~It can do:~~

- ~~Send 1 message~~
- ~~Receive 1 message~~
  ~~**simultaneously**~~

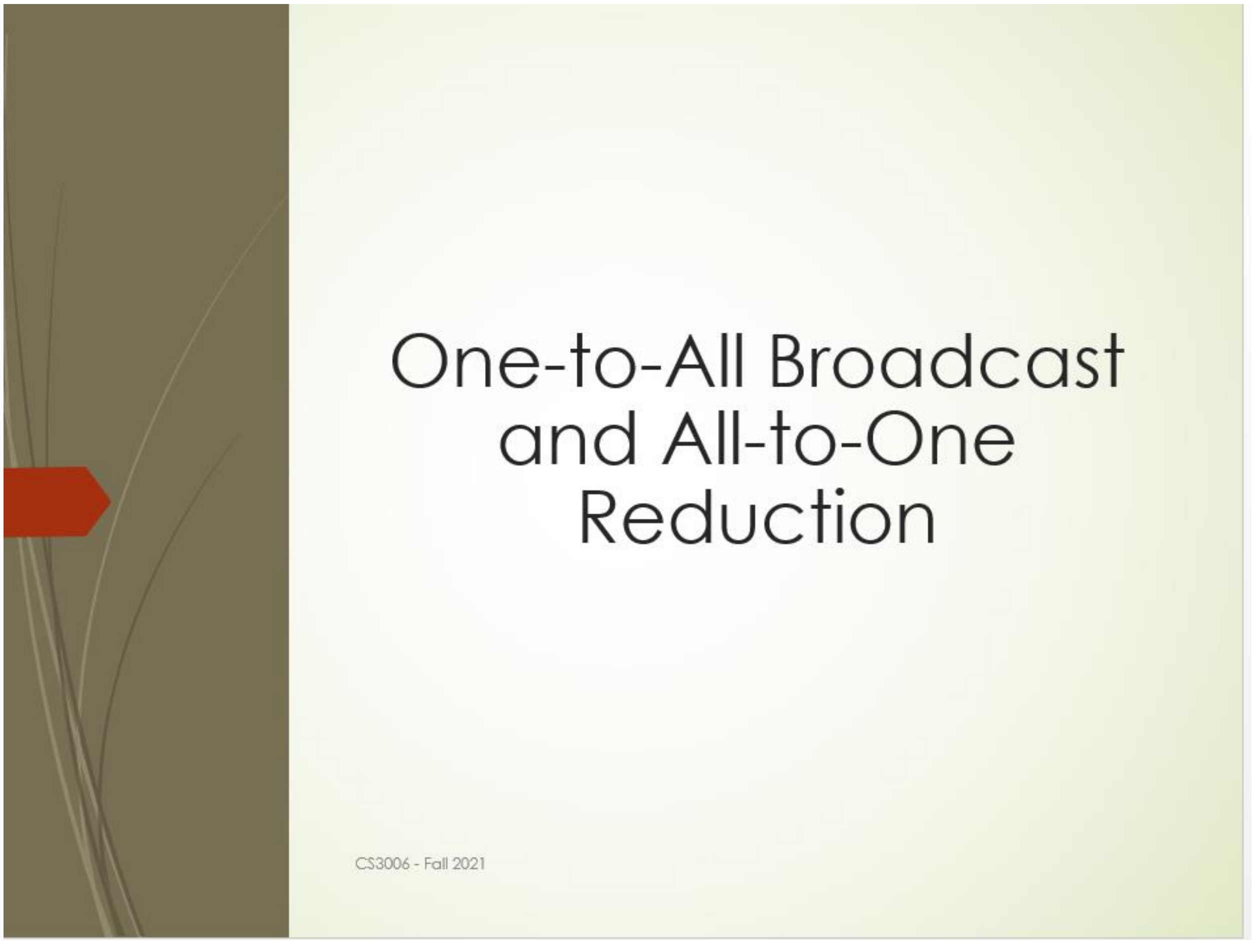
~~Same or different link — both allowed.~~

~~**Example:**~~
~~Node A sends to B~~
~~Node A receives from C~~
~~→ This is allowed at the same time.~~

Summary You Can Write in Exam

- Communication is essential for parallel algorithms.
- The simplified cost model is $\mathbf{t_s + m \cdot t_v}$.
- Cut-through routing is assumed for faster message passing.
- Costs of communication patterns (broadcast, scatter, etc.) are analyzed on various interconnection networks.
- Networks like linear arrays are included mainly for teaching.
- Communication assumptions:
  - Same communication time between any nodes

    - Bi-directional links
    - No congestion for direct neighbors
    - **Single-port model**: only one send and one receive at a time, but both can occur simultaneously.



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

One-to-All Broadcast & All-to-One Reduction for permuting a matrix with 2D block distribution

Nice — let's walk through a clear, practical recipe you can implement (MPI-style) and the costs so you know what's expensive.

Setup / assumptions

- Matrix A is split into a **2D block distribution** on a $P \times Q$ process grid.
  Each process (p, q) holds a local block $A_{p,q}$ of size $r \times c$ (rows × cols).
  So global dims = $P \cdot r$ by $Q \cdot c$.
- We want to **permute** the matrix (reorder blocks) so blocks move to new owner processes.
- Communication cost model: each message costs $t_s + m\, t_w$ where m = message words.

- Use row and column communicators (MPI_Comm_split or similar) for efficient collectives.

High-level idea

A permutation that moves blocks between processes can be built from two basic patterns:

1. **One-to-All Broadcast** along a row (or column): one process sends its block to every process in that row.
2. **All-to-One Reduction** along a column (or row): every process in a column sends some partial data which gets reduced/combined at a single root.

For a general permutation you typically:

- First redistribute data along rows so each process holds the blocks that belong to its *target columns*.
- Then redistribute along columns to place blocks into final target processes.

Using row/column collectives instead of many point-to-point ops reduces startups and matches network topology.

Concrete 2-step algorithm (row-then-column) — simple and common

Goal: move each block $A_{p,q}$ to a target process (p', q') according to a known permutation `dest(p,q) = (p',q')`.

Step A — *Row phase (one-to-all style / AlltoAll within row)*

- For each row p (processes $(p, 0.Q - 1)$):
  Each process (p, q) prepares sub-blocks destined for every target column q'.
  Then run an `MPI_Alltoallv` (or Q point-to-point sends or pipelined broadcasts) inside the row communicator so after this phase every process (p, q') has the pieces from row p that belong to column q'.

Why this is “one-to-all flavored”: a source block might be needed by many processes in

Why this is “one-to-all flavored”: a source block might be needed by many processes in the row, so a broadcast-like pattern could be used instead of many sends.

Step B — *Column phase (all-to-one / AlltoAll within column or reduction)*

- For each column q': processes $(0.P - 1, q')$ now have all pieces that belong to column q' but scattered across different row owners.
  Use `MPI_Alltoallv` in the column communicator (or `MPI_Reduce` if you must combine data rather than just move it) to place each final block to its target process (p', q').

If the permutation requires **combining** several contributions into one block (e.g., sum/accumulate), replace the last step by an `MPI_Reduce` (all-to-one reduction) with the appropriate operation.

~~If the permutation requires **combining** several contributions into one block (e.g., sum/accumulate), replace the last step by an `MPI_Reduce` (all-to-one reduction) with the appropriate operation.~~

~~Pseudocode (MPI-style)~~

```
// Each process has rank pRow, pCol, rowComm, colComm

// localBlock is A_local

// 1) Prepare send buffers: for each target column q' extract bytes to send
for each target_column q' {
    pack sendbuf_row[q'] with sub-data of localBlock destined to column q'
}
// Exchange inside the row
MPI_Alltoallv(sendbuf_row, sendcounts, sdispls,
              recvbuf_row, recvcnts, rdispls, MPI_BYTE, rowComm);

// now recvbuf_row contains pieces that belong to my column but from
different rows

// 2) Prepare send buffers for column phase
for each target_row p' {
    pack sendbuf_col[p'] with appropriate pieces from recvbuf_row destined to
process (p', myCol)
}
MPI_Alltoallv(sendbuf_col, sendcounts, sdispls,
              recvbuf_col, recvcnts, rdispls, MPI_BYTE, colComm);

// recvbuf_col now holds my final blocks
```

~~If instead you must **reduce** (sum/combine) contributions from many processes into one final block: // after packing local contribution to the final block~~

~~MPI_Reduce(local_partial_block, final_block, block_size, MPI_DOUBLE, MPI_SUM, root_rank, colComm);~~

Example: 4×4 global matrix, 2×2 process grid

- $P = 2, Q = 2$. Each process holds a 2×2 block (so global 4×4).
- Suppose permutation swaps block (0,0) ↔ (1,1) and (0,1) ↔ (1,0).
- Row phase: processes in row 0 and row 1 exchange pieces to build column groups.
- Column phase: columns send to final owners. Very few messages; collectives handle routing.

I can draw an ASCII mapping if you want — but the two-phase method stays the same.

Communication cost estimate

Let each local block be $M = r \cdot c$ words.

- Row-phase (inside a row of Q procs): each proc sends ~M total split across Q messages. Using an `Alltoall` with optimized implementation, cost ≈ $(\log Q)\ (t_s + \frac{M}{Q} t_w)$ per exchange *if* tree algorithm is used — otherwise sum of per-message startups.
- Column-phase (inside a column of P procs): similar cost with P and possibly message sizes M.

Rough total startup cost: ~$O(\log Q + \log P)\ t_s$. Bandwidth cost: $O(M\ t_w)$ (each block transferred once).

If you instead used naive point-to-point for all block pairs, startup cost could be $O(PQ)\ t_s$ — much worse for many processes.

When to use One-to-All vs All-to-One

- **One-to-All / broadcast**: good when one source's block must be sent identically to many processes (same data to many). Use `MPI_Bcast` on row/column communicator.
- **All-to-One / reduction**: use when many processes contribute partial results that must be **combined** (sum/min/etc.) at a single root. Use `MPI_Reduce`.

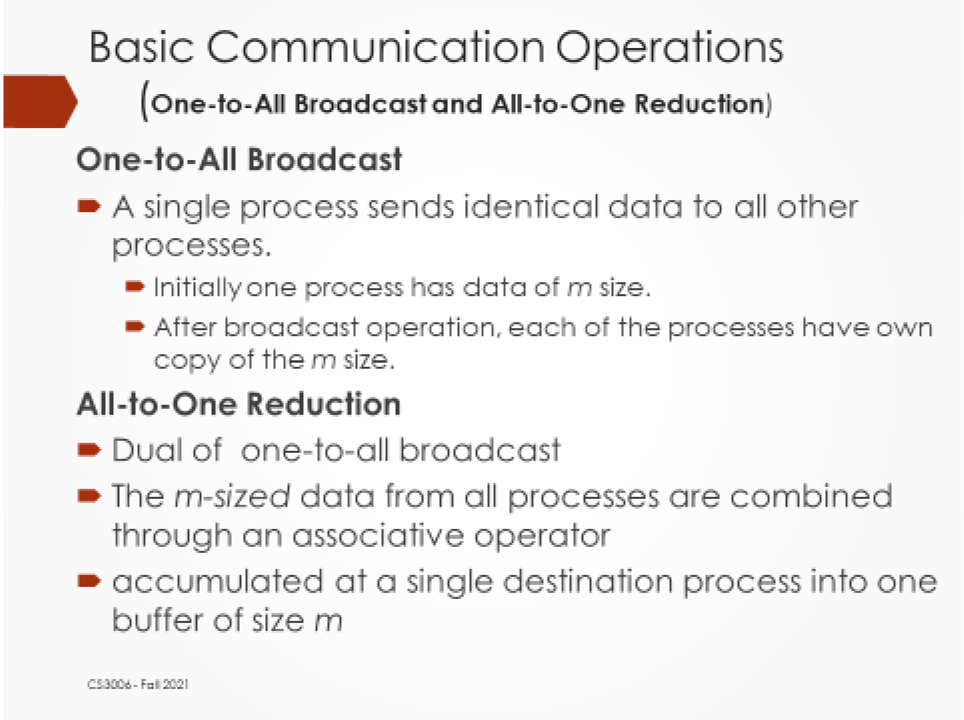
For permutations you usually need **all-to-all style exchange** (repartitioning), but some permutations can be implemented efficiently by combining broadcasts and reductions depending on the mapping.

~~Practical optimizations~~

- ~~Use `MPI_Alltoallv` inside rows/cols rather than many sends/receives.~~
- ~~Create row and column communicators to confine collective traffic to subnets.~~
- ~~Pack contiguous memory to reduce number of small messages.~~
- ~~Overlap communication & computation (nonblocking `Ialltoallv` or `Irecv` + compute).~~
- ~~If permutation is sparse (few blocks move), prefer targeted nonblocking sends.~~
- ~~If network supports hierarchy (node-local fast paths), do intranode aggregation first.~~

~~Short checklist you can implement now~~

1. ~~Build `rowComm` and `colComm` for the $P \times Q$ grid.~~
2. ~~For each local block, compute destination (p', q').~~
3. ~~Pack data per destination column and call `MPI_Alltoallv` on `rowComm`.~~
4. ~~Unpack, then pack per destination row and call `MPI_Alltoallv` on `colComm`.~~
5. ~~If you need accumulation, use `MPI_Reduce` in the final column communicator instead of an all-to-all.~~



Basic Communication Operations

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

One-to-All Broadcast

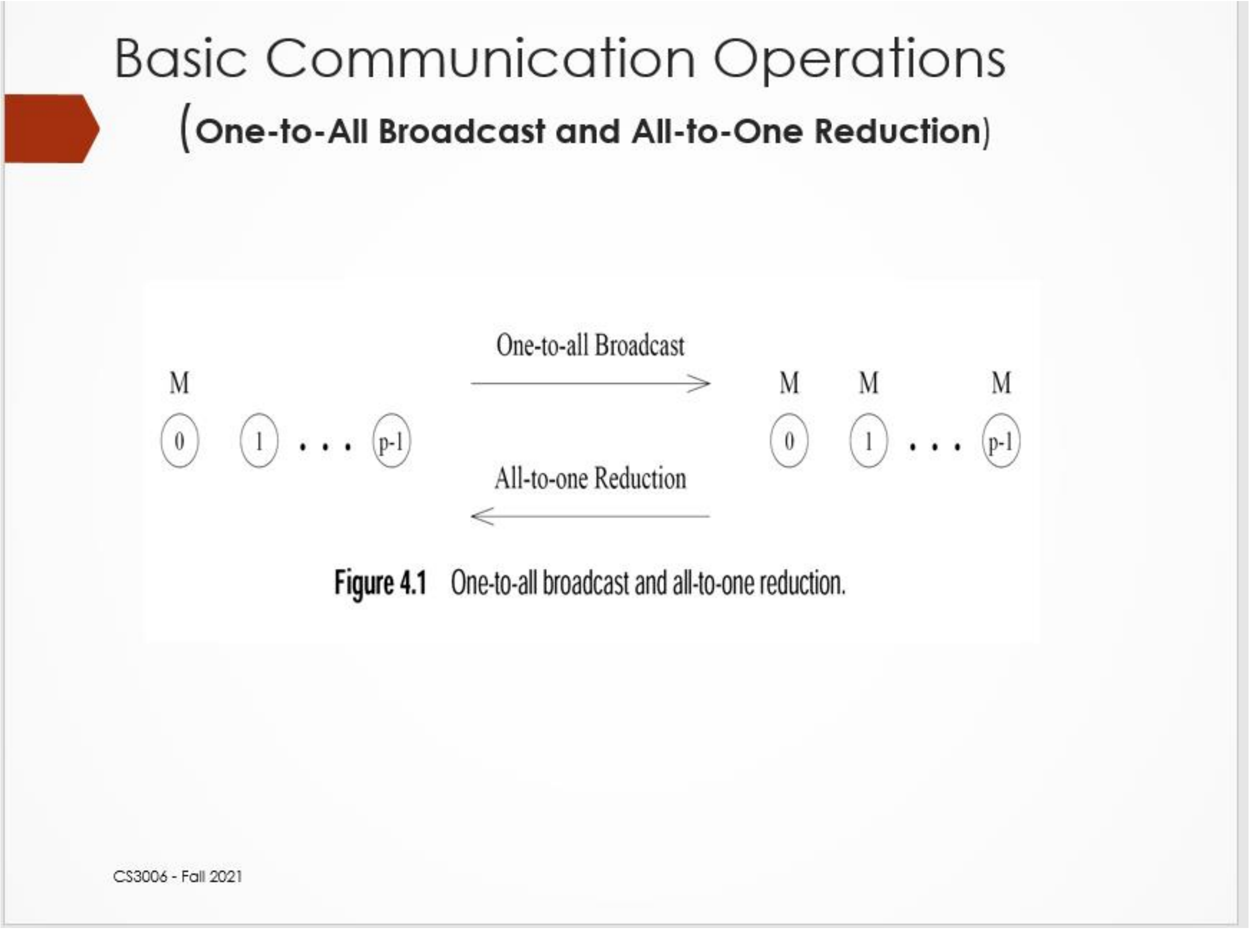
- A single process sends identical data to all other processes.
  - Initially one process has data of *m* size.
  - After broadcast operation, each of the processes have own copy of the *m* size.

All-to-One Reduction

- Dual of one-to-all broadcast
- The *m-sized* data from all processes are combined through an associative operator
- accumulated at a single destination process into one buffer of size *m*

CS3006 - Fall 2021

Applications: vector-matrix multiplication, shortest paths calculation, and inner vector product.



Basic Communication Operations

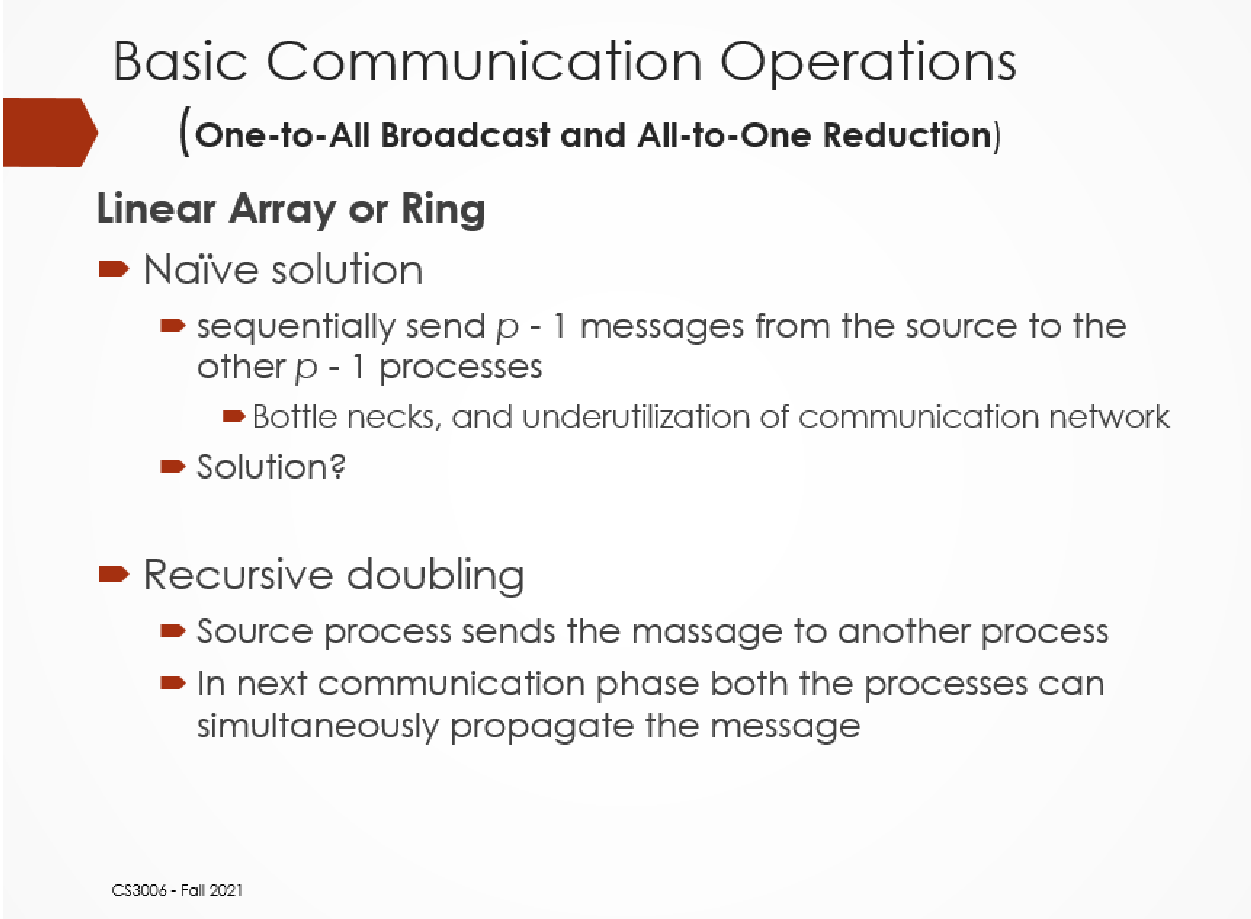
(One-to-All Broadcast and All-to-One Reduction)

Figure 4.1 One-to-all broadcast and all-to-one reduction.

CS3006 - Fall 2021

Applications: vector-matrix multiplication, shortest paths calculation, and inner vector product.

1.



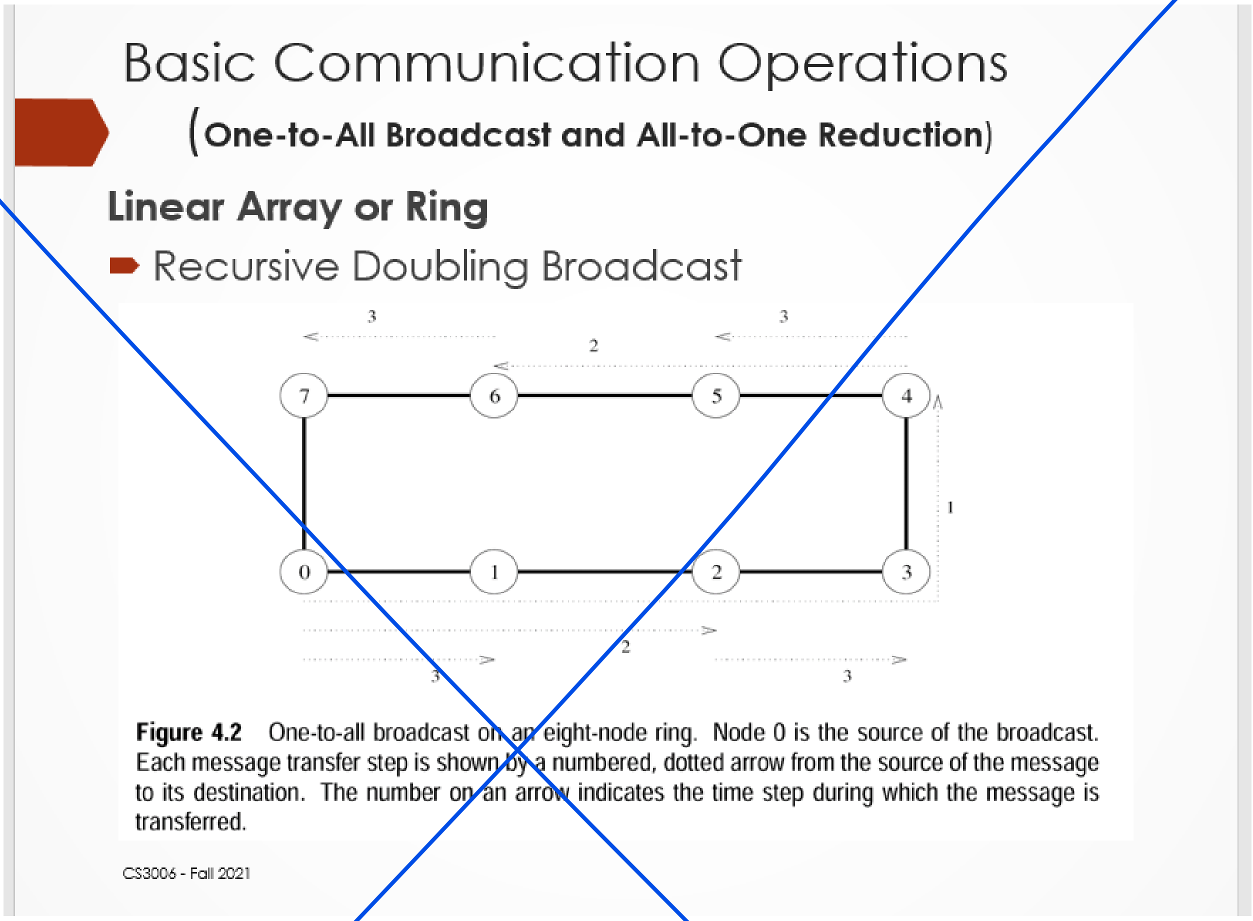
Basic Communication Operations

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

Linear Array or Ring

- Naïve solution
  - sequentially send *p* - 1 messages from the source to the other *p* - 1 processes
    - Bottle necks, and underutilization of communication network
  - Solution?

- Recursive doubling
  - Source process sends the massage to another process
  - In next communication phase both the processes can simultaneously propagate the message



Basic Communication Operations

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

Linear Array or Ring

- Recursive Doubling Broadcast

Figure 4.2 One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.

Basic Communication Operations

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

Linear Array or Ring

- Recursive Doubling Broadcast

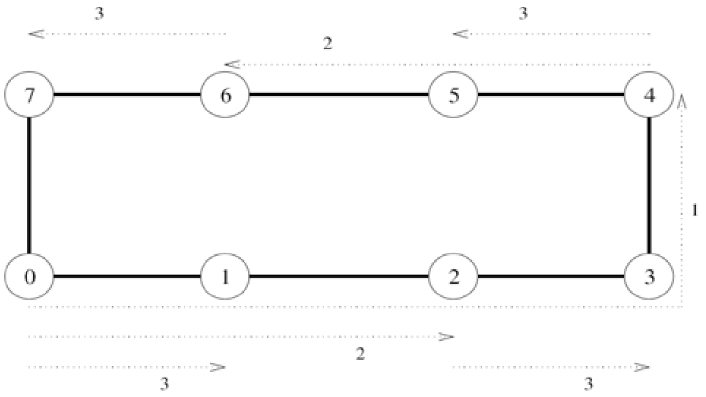
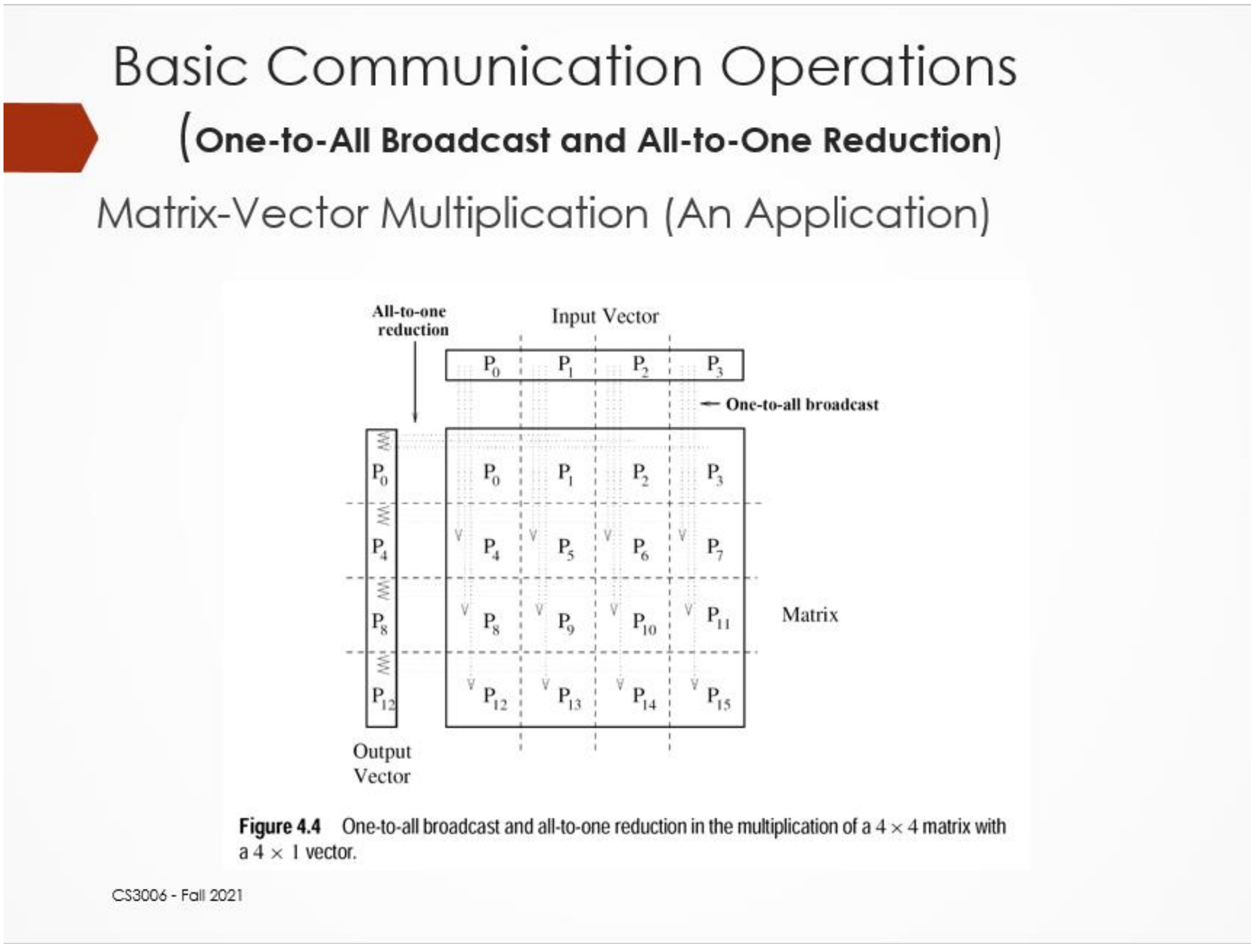


Figure 4.2 One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.



Basic Communication Operations

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

Matrix-Vector Multiplication (An Application)

Figure 4.4 One-to-all broadcast and all-to-one reduction in the multiplication of a 4 × 4 matrix with a 4 × 1 vector.

CS3006 - Fall 2021

Odd will send message to even and associative operator will combine the results

⋆ Basic Communication Operations

1) One-to-All Broadcast

- **Definition:**
  One process (source) has data of size **m** and sends the *same* data to **all other processes**.
- **End result:**
  Every process ends up with a **local copy** of the m-sized data.
- **Use cases:**
  - vector–matrix multiplication
  - shortest path computations
  - inner product

---

2) All-to-One Reduction

- **Definition:**
  Opposite of broadcast.
  All processes send **m-sized data** to a root process, and the root **combines** them using an **associative operator** (sum, min, max, etc.).
- **End result:**
  The root gets **one final m-sized result**.
- **Use cases:**
  - vector–matrix multiplication
  - shortest paths
  - inner product

⋆ Communication on Linear Array / Ring

Naïve method

- Send the message sequentially from one process to the next.
- Requires **p − 1** communication steps.
- **Problems:**
  - bottleneck at the sender
  - network under-utilized
  - slow for large p

⋆ Efficient Method: Recursive Doubling

Idea

- The source sends the message to **one process**.
- In the next step, **both** processes forward the message.
- Then 4 processes send, then 8, and so on…

This doubles the number of informed processes each step.

Result:

- Broadcast completes in **$\log_2(p)$** steps instead of **p − 1**.

⭑ Broadcast with Recursive Doubling (Linear Array / Ring)

Step pattern:

1. Step 0: Process 0 → Process 1
2. Step 1: Processes 0,1 → Processes 2,3
3. Step 2: Processes 0,1,2,3 → Processes 4,5,6,7
   ... and so on.

Benefits:

- No congestion
- Full network utilization
- Time complexity: **O(log p)**

⭑ Reduction on Linear Array / Ring

Reduction uses the **same recursive doubling pattern**, but in reverse:

1. P1 sends to P0, combine
2. P2 sends to P0 via steps, combine
3. Continue until only one process has final result

Odd processes typically send to even processes so the associative operator can be applied easily.

⭑ Matrix-Vector Multiplication (Application)

Where broadcast is used:

- A row (or column) of matrix or vector elements must be sent to all processes handling different rows/columns.

Where reduction is used:

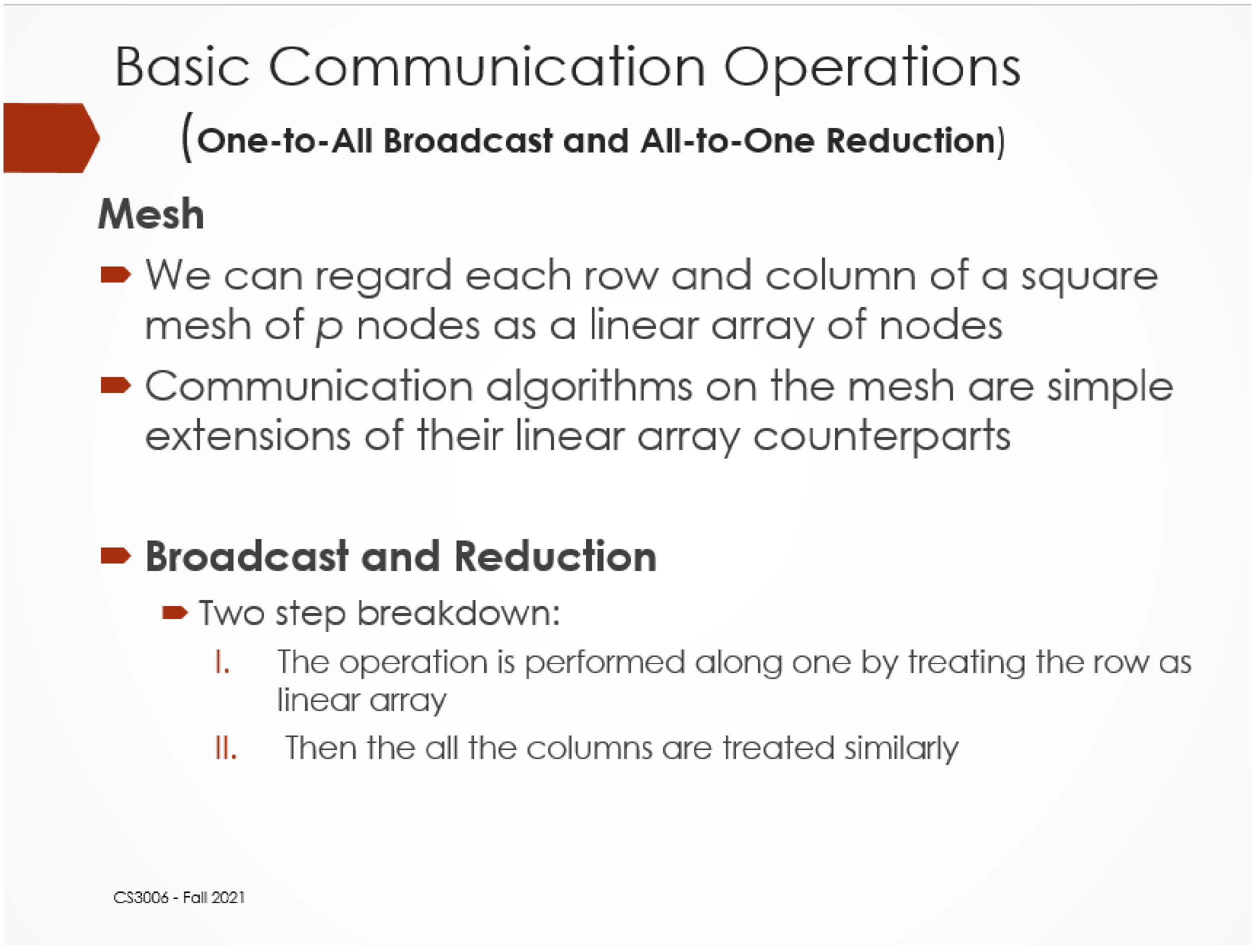
- Partial results from different processes must be **combined** into the final vector.

So:

- **Broadcast → distribute data**
- **Reduction → gather/accumulate results**

⭑ Key Takeaways (Exam-Ready)

- Broadcast = distribute **same** data to all.
- Reduction = combine data from all to **one**.
- Naïve communication = slow, $p - 1$ steps.
- Recursive doubling = fast, $\log_2(p)$ steps.
- Used in many parallel algorithms like matrix-vector multiplication, shortest paths, and inner products.



Basic Communication Operations

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

Mesh

- We can regard each row and column of a square mesh of *p* nodes as a linear array of nodes
- Communication algorithms on the mesh are simple extensions of their linear array counterparts

- **Broadcast and Reduction**
  - Two step breakdown:
    I. The operation is performed along one by treating the row as linear array
    II. Then the all the columns are treated similarly

Basic Communication Operations

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

Mesh (Broadcast and Reduction)

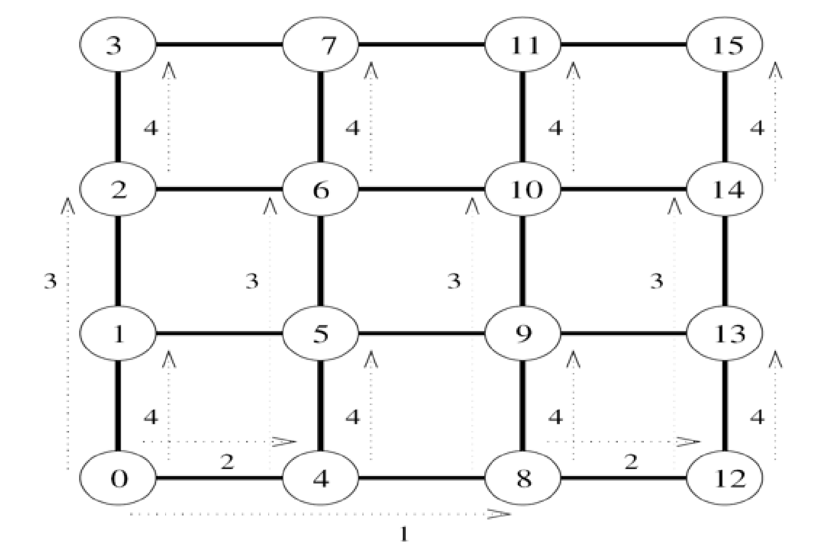
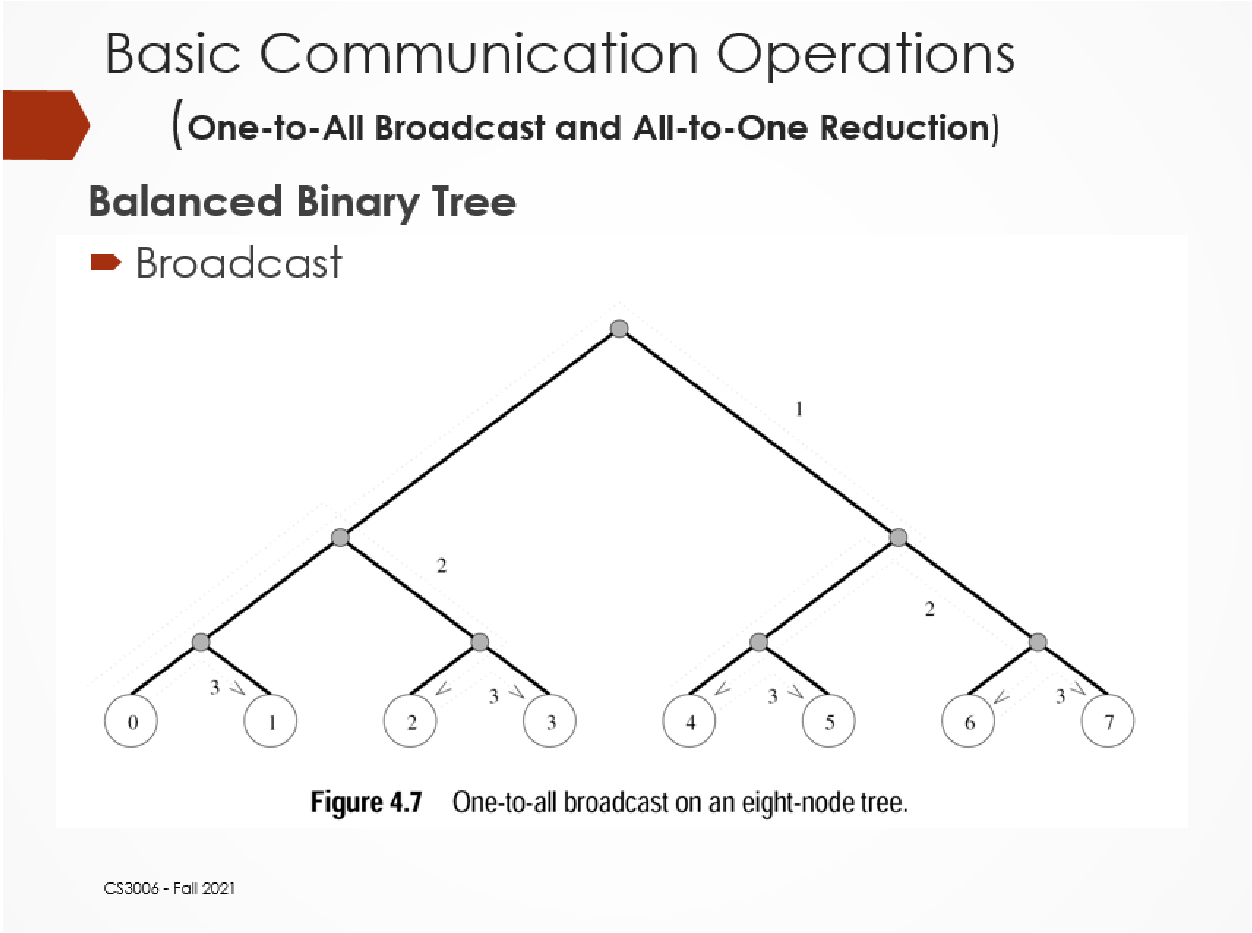


Figure 4.5 One-to-all broadcast on a 16-node mesh.



Basic Communication Operations

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

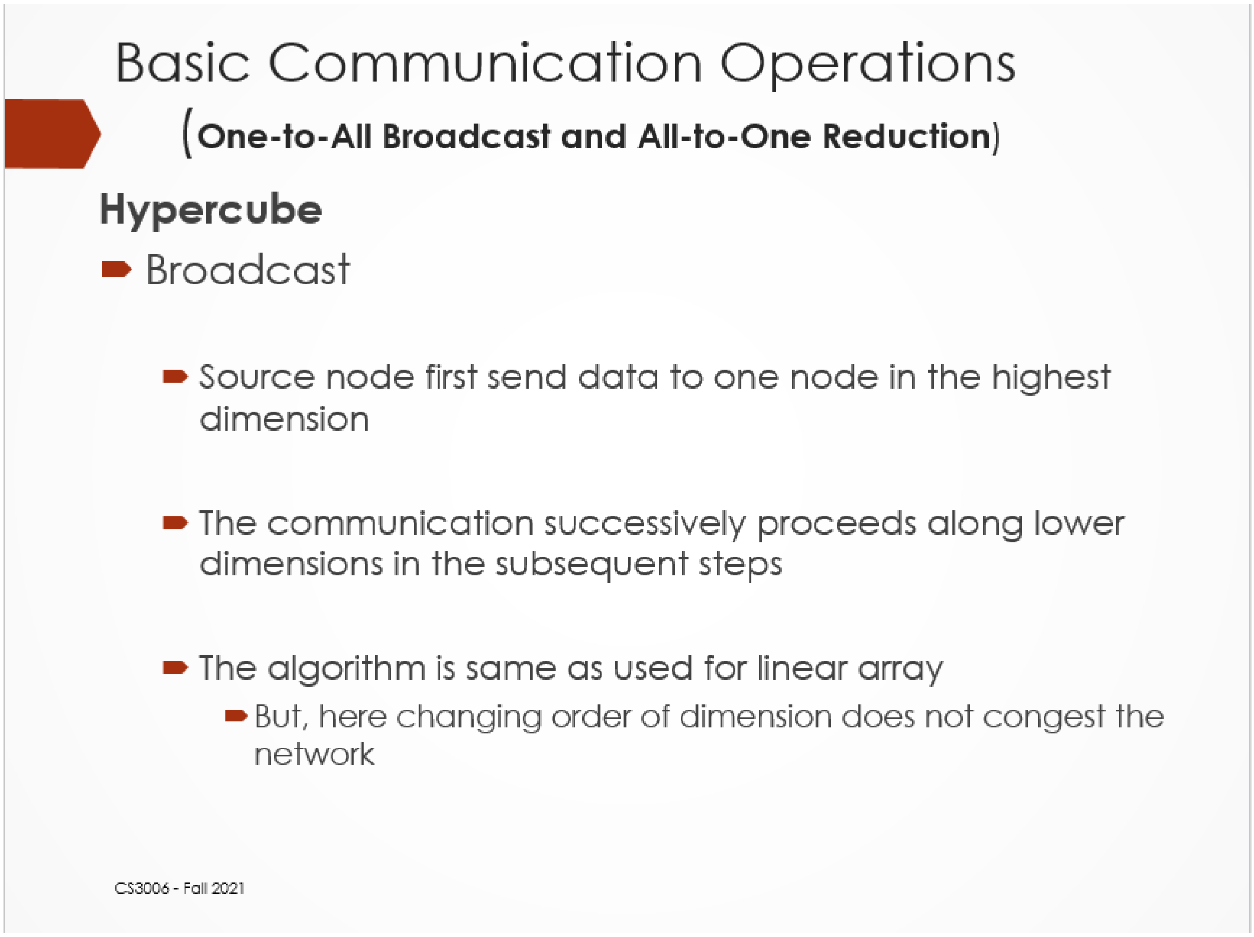
Balanced Binary Tree

- Broadcast

Figure 4.7 One-to-all broadcast on an eight-node tree.

CS3006 - Fall 2021

Hypercube algorithm without any changes.

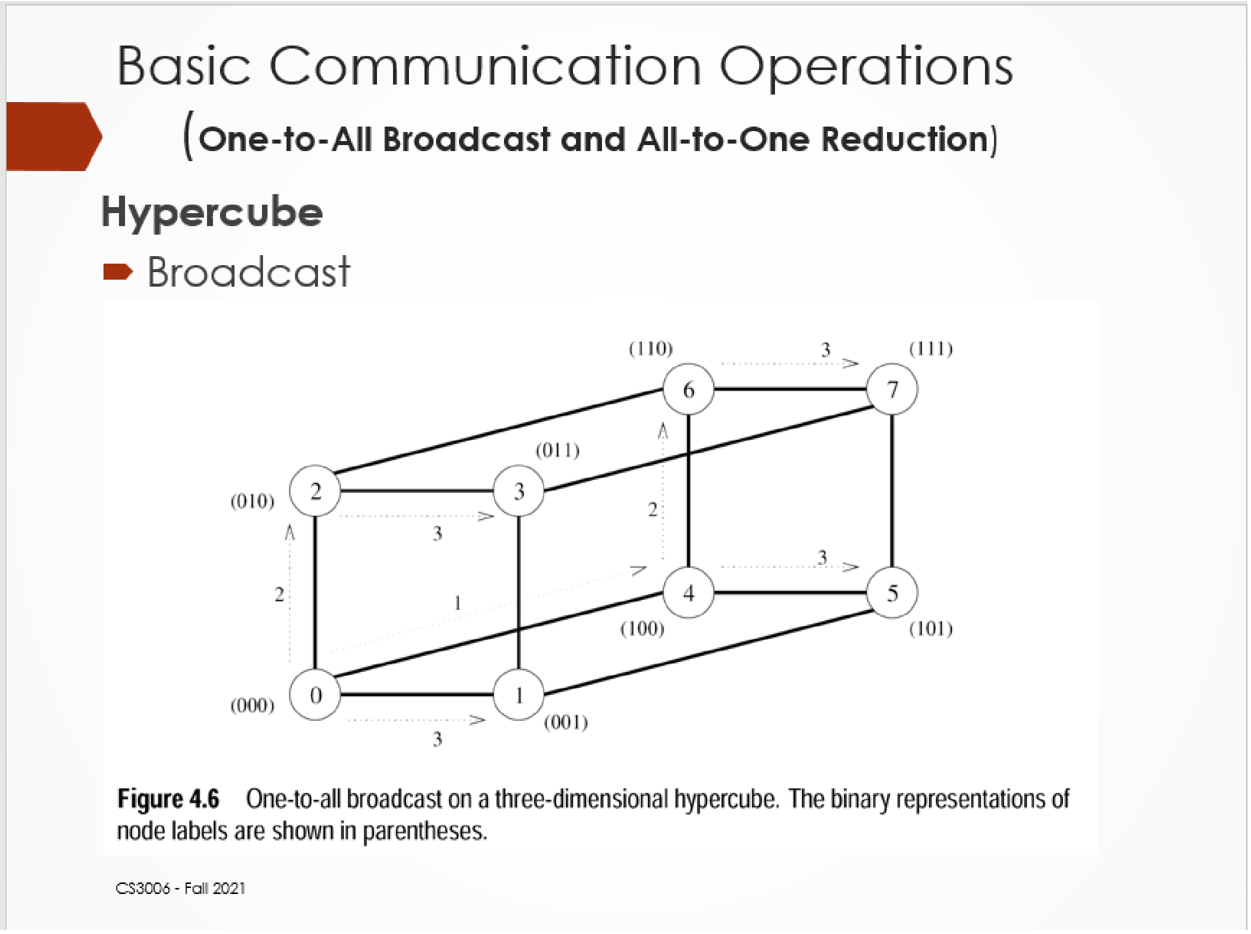


Basic Communication Operations

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

Hypercube

- Broadcast
  - Source node first send data to one node in the highest dimension
  - The communication successively proceeds along lower dimensions in the subsequent steps
  - The algorithm is same as used for linear array
    - But, here changing order of dimension does not congest the network



Basic Communication Operations

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

Hypercube

- Broadcast

Figure 4.6 One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses.

CS3006 - Fall 2021

⋆ 1. Mesh Topology

A **mesh** is a 2D grid of processors.
Example: a √p × √p mesh.

Key idea:

- A mesh can be viewed as:
  - √p **rows** → each row is a **linear array**
  - √p **columns** → each column is also a **linear array**

Therefore:

☞ **Communication algorithms for mesh = extensions of linear array algorithms.**

⭑ Broadcast on Mesh (Two-Step Method)

To broadcast data from one source to **all processors**:

Step 1: Row Broadcast

- Treat the **row of the source** as a **linear array**.
- Perform normal broadcast (recursive doubling) across the row.

Step 2: Column Broadcast

- Each node in that row now has the data.
- Every one of them acts as a source for its **column**.
- Perform broadcast down each column.

Total cost:

Broadcast along row + broadcast along columns.

⭑ Reduction on Mesh (Two-Step Method)

To reduce values from all processors to a single destination:

Step 1: Column Reduction

- Perform reduction in each column toward the top row.

Step 2: Row Reduction

- The top row now holds partial sums.
- Reduce along that row to the final destination.

Same idea as broadcast, but reversed.

⭑ 2. Balanced Binary Tree

A **balanced binary tree** can also be used for communication.

Broadcast

- The root sends to its two children.
- They forward to their children.
- Complete in **O(log p)** steps.

Reduction

- Leaves send to parent.
- Parent combines children values.
- Continue upward.

Tree-based collectives are simple, but not always optimal on 2D/3D networks.

⭑ 3. Hypercube Topology

A hypercube has **p = 2^d** nodes and **d dimensions**.

Key principle:

Each node differs from neighbors by **one bit** in its binary address.

⭑ Broadcast on Hypercube

Algorithm:

1. Start at the **source node**.
2. In the first step, send data along the **highest dimension**.
3. In each next step, send along the **next lower dimension**.
4. After **d steps**, all nodes have the message.

Why is this efficient?

- In each step, the number of nodes with the message **doubles** (same as recursive doubling).
- Network is symmetric → **changing the order of dimensions never causes congestion**.

Time complexity: O(log p)

⭑ Reduction on Hypercube

Same as broadcast, but reverse direction:

1. All nodes send partial data along dimension 0
2. Combine
3. Send along dimension 1
4. Continue until reaching the destination

Total: **log p steps**

⭑ Summary Table

Topology	Broadcast Strategy	Reduction Strategy	Time
Linear Array	Sequential or Recursive Doubling	Reverse of doubling	O(p) or O(log p)
2D Mesh	Row broadcast → column broadcast	Column reduction → row reduction	O(√p)
Binary Tree	Root → children recursively	Leaves → root	O(log p)
Hypercube	Send on each dimension	Opposite direction	O(log p)

⭑ Ultra-Short Exam Points (ready to memorize)

Mesh

- Treat rows/columns as linear arrays.
- Broadcast: row broadcast → column broadcast.
- Reduction: column reduction → row reduction.

Binary Tree

- Broadcast down the tree, reduction up the tree.

Hypercube

- Broadcast along highest dimension first, then lower ones.
- No congestion; each step doubles the number of informed nodes.

Great — here are **clear ASCII diagrams** for **Mesh**, **Balanced Binary Tree**, and **Hypercube**. These are **exam-ready visuals** you can draw exactly in your paper.

★ 1. Mesh Topology (Broadcast & Reduction) — Diagram

2D Mesh (4×4 example)

```
(0,0) -- (0,1) -- (0,2) -- (0,3)
  |        |        |        |
(1,0) -- (1,1) -- (1,2) -- (1,3)
  |        |        |        |
(2,0) -- (2,1) -- (2,2) -- (2,3)
  |        |        |        |
(3,0) -- (3,1) -- (3,2) -- (3,3)
```

Broadcast on Mesh (Two-Step Method)

Step 1: Broadcast along source row (row 0)

```
(0,0) → (0,1) → (0,2) → (0,3)
```

Step 2: Each row node broadcasts down its column

```
(0,0)
  ↓
(1,0)
  ↓
(2,0)
  ↓
(3,0)
```

```
(0,1)
  ↓
(1,1)
  ↓
(2,1)
  ↓
(3,1)
```

```
...
```

Result:

All nodes receive the message.

Reduction on Mesh (Reverse)

Step 1: Reduce up the columns

```
(3,0)
  ↑
(2,0)
  ↑
(1,0)
  ↑
(0,0)
```

(All columns do this simultaneously)

Step 2: Reduce along the top row

```
(0,3) → (0,2) → (0,1) → (0,0)
```

⭑ 2. Balanced Binary Tree — Diagram

Perfect for showing recursive broadcast/reduction.

Broadcast: Root at the top

```
        (0)
       /   \
     (1)   (2)
     / \   / \
   (3) (4)(5) (6)
```

Broadcast steps:

```
Step 1: 0 → 1,2
Step 2: 1 → 3,4 and 2 → 5,6
```

Reduction (reverse):

```
3,4 → 1
```

```
5,6 → 2
1,2 → 0
```

Time complexity: **O(log p)**

⭐ 3. Hypercube (4-D example)

A hypercube connects nodes whose binary IDs differ in 1 bit.

Example: 3-D Hypercube (8 nodes)

```
000 ----- 001
 |         |
010 ----- 011
 |         |
100 ----- 101
 |         |
110 ----- 111
```

Each edge is a flip in **one bit**.

⭐ Broadcast on Hypercube — Dimension Order

Step 1: highest dimension (bit 2)

```
000 → 100
```

Step 2: next dimension (bit 1)

```
000 → 010
100 → 110
```

Step 3: lowest dimension (bit 0)

```
000 → 001
010 → 011
100 → 101
110 → 111
```

After 3 steps ($\log_2 8 = 3$), all nodes have the data.

⭑ Reduction on Hypercube (reverse)

Each dimension folds inward:

```
000 ← 001
010 ← 011
100 ← 101
110 ← 111
```

Then:

```
000 ← 010
100 ← 110
```

Then final:

```
000 ← 100
```

Done in **log p** steps.

⭑ Super Short Exam Notes (write exactly this)

Mesh

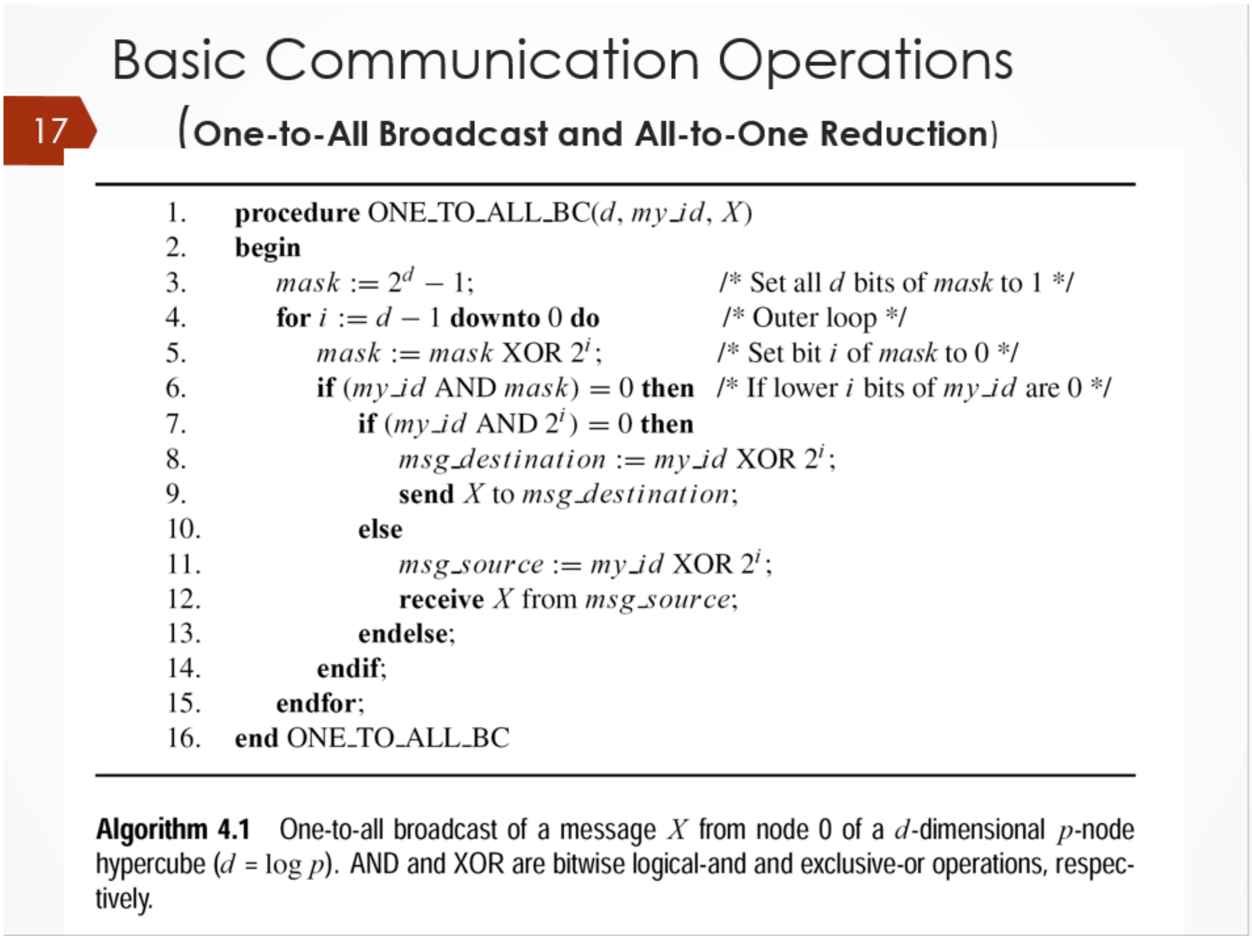
- Treat each row/column as linear array.
- Broadcast = row broadcast → column broadcast.
- Reduction = column reduction → row reduction.

Balanced Tree

- Broadcast propagates from root to children.
- Reduction accumulates from leaves to root.

Hypercube

- Each step communicates along one dimension.
- After log p steps all nodes have the data.
- No congestion due to symmetric topology.



Basic Communication Operations

17

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

```
1.  procedure ONE_TO_ALL_BC($d$, $my\_id$, $X$)
2.  begin
3.      $mask := 2^d - 1$;                      /* Set all $d$ bits of $mask$ to 1 */
4.      for $i := d - 1$ downto 0 do            /* Outer loop */
5.          $mask := mask$ XOR $2^i$;           /* Set bit $i$ of $mask$ to 0 */
6.          if ($my\_id$ AND $mask$) = 0 then   /* If lower $i$ bits of $my\_id$ are 0 */
7.              if ($my\_id$ AND $2^i$) = 0 then
8.                  $msg\_destination := my\_id$ XOR $2^i$;
9.                  send $X$ to $msg\_destination$;
10.             else
11.                 $msg\_source := my\_id$ XOR $2^i$;
12.                 receive $X$ from $msg\_source$;
13.             endelse;
14.         endif;
15.     endfor;
16. end ONE_TO_ALL_BC
```

Algorithm 4.1 One-to-all broadcast of a message X from node 0 of a d-dimensional p-node hypercube ($d = \log p$). AND and XOR are bitwise logical-and and exclusive-or operations, respectively.

6) id AND mask==0 → if I will be in communication in this step (if my i bits are zero)

7) Id AND 2i ==0 → if I'm the sender of message (if my i+1th bit is 0)

8) Xoring id with 2i gives my partner in ith phase of the algorithm

⭑ Slide 17 — ONE_TO_ALL_BC (Hypercube Broadcast from node 0)

Goal:

Node **0** sends message **X** to all other nodes of a **d-dimensional hypercube**.

Key Idea:

A hypercube uses **bit positions** to decide which neighbor to send to.

Process ID = binary number

Example (d = 3):

```
000
001
010
011
100
101
110
111
```

Each ID differs by **1 bit** = they are neighbors.

⭑ How the algorithm works

The mask controls which bit positions are checked.

Step 1:

```
mask = 2^d - 1
```

This means:

```
d = 3 → mask = 111
d = 4 → mask = 1111
```

Step 2:
Loop from the **highest bit** to the lowest bit.

For each bit position **i**:

```
mask = mask XOR 2^i
```

This sets bit **i** to 0.

Now mask looks like:

i	new mask (example d=3)
2	011
1	001

i	new mask (example d=3)
0	000

★ Decision Rule

```
if (my_id AND mask) == 0
```

This condition selects **only nodes whose lower i bits are zero**.

These nodes are the “senders”.

Then:

```
msg_destination = my_id XOR 2^i
send X to msg_destination
```

This means:

- Flip bit **i**
- Send message to the node in that dimension

★ Visual Example (d = 3)

Step 1 (i=2): flip bit 2

Node 0 (000) → 100
Node 1 (001) → no send
Node 2 (010) → no send
Node 3 (011) → no send

```
000 → 100
```

Step 2 (i=1): flip bit 1

Nodes with lower bits = 0: 000 and 100

So both send:

```
000 → 010
100 → 110
```

Step 3 (i=0): flip bit 0

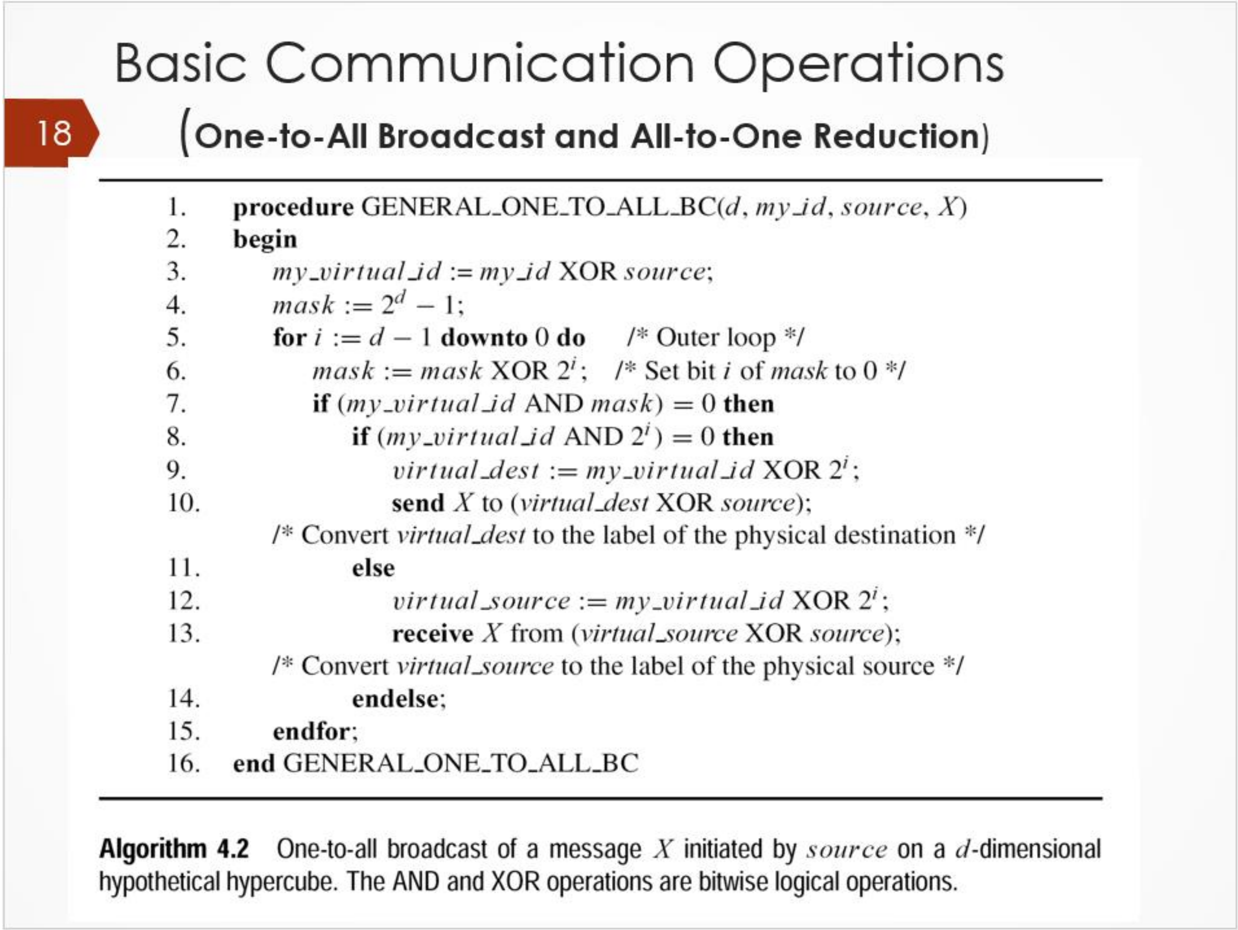
Nodes: 000, 010, 100, 110

Each sends:

```
000 → 001
010 → 011
100 → 101
110 → 111
```

Now **all nodes have X**.

➞ **Broadcast complete in log(p) steps**.



Basic Communication Operations

18

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

```
procedure GENERAL_ONE_TO_ALL_BC($d$, $my\_id$, $source$, $X$)
begin
    $my\_virtual\_id := my\_id$ XOR $source$;
    $mask := 2^d - 1$;
    for $i := d - 1$ downto 0 do      /* Outer loop */
        $mask := mask$ XOR $2^i$;   /* Set bit $i$ of $mask$ to 0 */
        if ($my\_virtual\_id$ AND $mask$) = 0 then
            if ($my\_virtual\_id$ AND $2^i$) = 0 then
                $virtual\_dest := my\_virtual\_id$ XOR $2^i$;
                send $X$ to ($virtual\_dest$ XOR $source$);
    /* Convert $virtual\_dest$ to the label of the physical destination */
            else
                $virtual\_source := my\_virtual\_id$ XOR $2^i$;
                receive $X$ from ($virtual\_source$ XOR $source$);
    /* Convert $virtual\_source$ to the label of the physical source */
            endelse;
    endfor;
end GENERAL_ONE_TO_ALL_BC
```

Algorithm 4.2 One-to-all broadcast of a message X initiated by $source$ on a d-dimensional hypothetical hypercube. The AND and XOR operations are bitwise logical operations.

source XOR source makes source→ zero and similarly maps other according to the source

⭑ Slide 18 — GENERAL_ONE_TO_ALL_BC

(**Broadcast from any arbitrary source node**)

This is the same algorithm as Slide 17, but with an **extra trick: virtual IDs**.

Why?
Because the original algorithm assumes **source is 0**.

But if the source is not 0, hypercube structure breaks.

Solution:

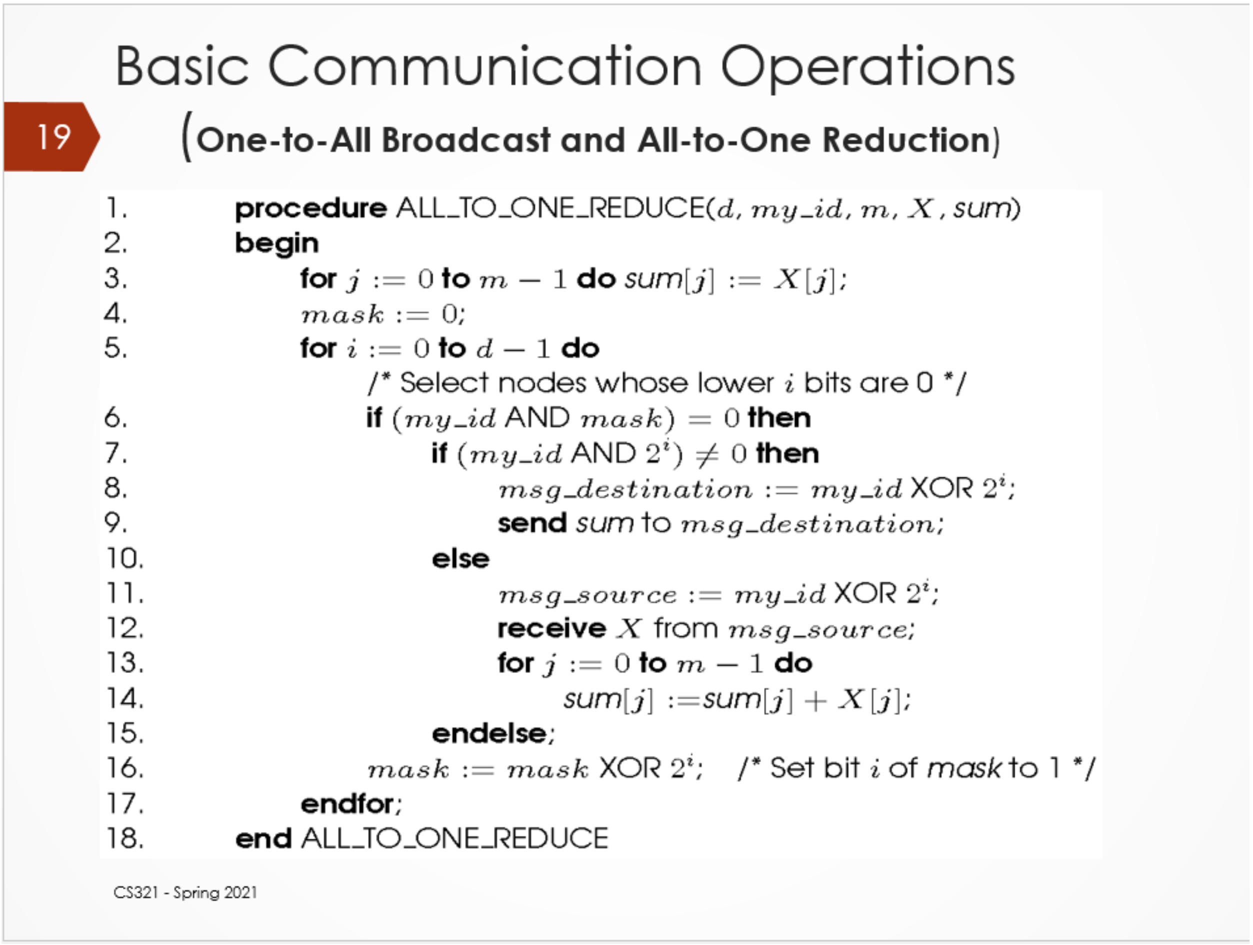
```
my_virtual_id = my_id XOR source
```

This maps the **source node** → **0** in virtual space.

Then the same broadcast logic is applied using virtual IDs.

Finally messages are mapped back using XOR with source.

➡ This ensures broadcast works even when source ≠ 0.



19

Basic Communication Operations

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

```
procedure ALL_TO_ONE_REDUCE($d$, $my\_id$, $m$, $X$, sum)
begin
    for $j := 0$ to $m - 1$ do sum$[j] := X[j]$;
    $mask := 0$;
    for $i := 0$ to $d - 1$ do
        /* Select nodes whose lower $i$ bits are 0 */
        if ($my\_id$ AND $mask$) $= 0$ then
            if ($my\_id$ AND $2^i$) $\neq 0$ then
                $msg\_destination := my\_id$ XOR $2^i$;
                send sum to $msg\_destination$;
            else
                $msg\_source := my\_id$ XOR $2^i$;
                receive $X$ from $msg\_source$;
                for $j := 0$ to $m - 1$ do
                    sum$[j]$ :=sum$[j] + X[j]$;
            endelse;
        $mask := mask$ XOR $2^i$;    /* Set bit $i$ of mask to 1 */
    endfor;
end ALL_TO_ONE_REDUCE
```

CS321 - Spring 2021

Assigned reading

⋆ Slide 19 — ALL_TO_ONE_REDUCE

(**Opposite of broadcast**)

Goal:

All nodes send data to **node 0** and combine results (sum, min, max, etc.)

Logic:

For each dimension i from **0 to d−1**:

- Nodes **whose lower i bits are 0** participate.
- Nodes with bit i = 1 → **send** to the neighbor lower-dimension node.

- Nodes with bit i = 0 → **receive** and **combine**.

Combination step:

```
sum[j] = sum[j] + X[j]
```

Works for vectors, not just single numbers.

⭐ Visual Example (d = 3)

Nodes:

```
000,001,010,011,100,101,110,111
```

Step 1 (i=0): combine bit 0

```
001 → 000
011 → 010
101 → 100
111 → 110
```

Step 2 (i=1): combine bit 1

```
010 → 000
110 → 100
```

Step 3 (i=2): combine bit 2

```
100 → 000
```

Finally:

```
000 has the full reduction result
```

➡ Reduce is also **O(log p)**.

Basic Communication Operations

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

Cost Estimation

- Broad cast needs **log(p)** point-to-point simple message transfer steps.
- Message size of each transfer is **m**
- Time foreach of the transfers is: $\boldsymbol{t_s + mt_w}$

Hence cost for log(p)transfers=T= $(\boldsymbol{t_s}+\boldsymbol{mt_w})\log \boldsymbol{p}$

Cost Estimation of One-to-All Broadcast

A broadcast sends the same message from **one process** to **all p processes**.

How many steps?

Using **recursive doubling**, the message reaches everyone in

$$\log(p)$$

steps.

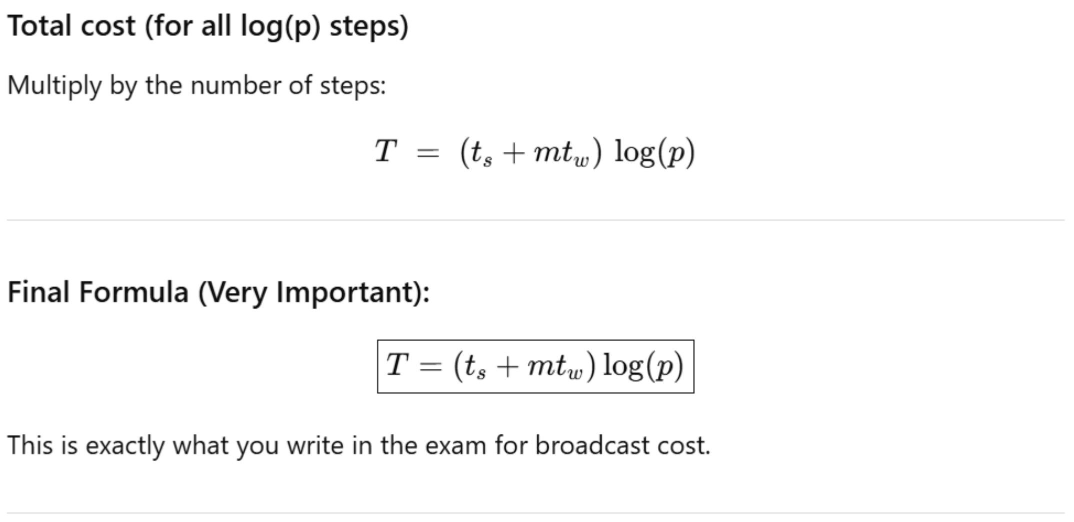
Cost per step

Each communication step sends:

- **Message size:** m
- **Startup time:** t_s
- **Per-word transfer time:** t_w

So, **time for one point-to-point transfer**:

$$t_s + mt_w$$



Total cost (for all log(p) steps)

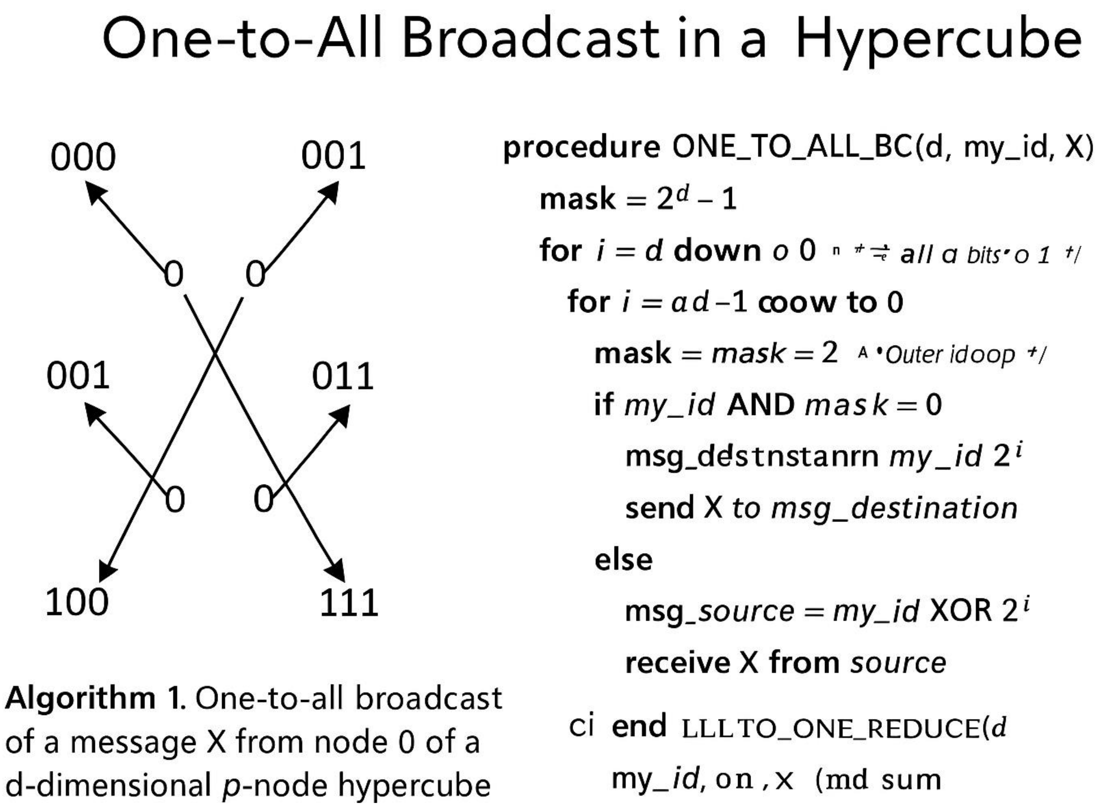
Multiply by the number of steps:

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

Final Formula (Very Important):

$$\boxed{T = (t_s + mt_w) \log(p)}$$

This is exactly what you write in the exam for broadcast cost.



One-to-All Broadcast in a Hypercube

```
procedure ONE_TO_ALL_BC(d, my_id, X)
  mask = $2^d - 1$
  for $i = d$ down o 0 ⁿ ⁺⥱ all a bits'o 1 ⁺/
    for $i = ad - 1$ coow to 0
      mask = mask = 2 ᴬ •Outer idoop ⁺/
      if my_id AND mask = 0
        msg_dedstnstanrn my_id $2^i$
        send X to msg_destination
      else
        msg_source = my_id XOR $2^i$
        receive X from source
  ci end LLLTO_ONE_REDUCE(d
     my_id, on , x (md sum
```

Algorithm 1. One-to-all broadcast of a message X from node 0 of a d-dimensional *p*-node hypercube