

MPI Tutorial 2: Collective Communication

From AllReduce to Parallel GEMM

What Will You Learn Today?

1

Understand MPI collective communication

4

Observe collective communication through MiniMPI

2

Explain why AllReduce matters in model training

5

See native MPI collective APIs in C

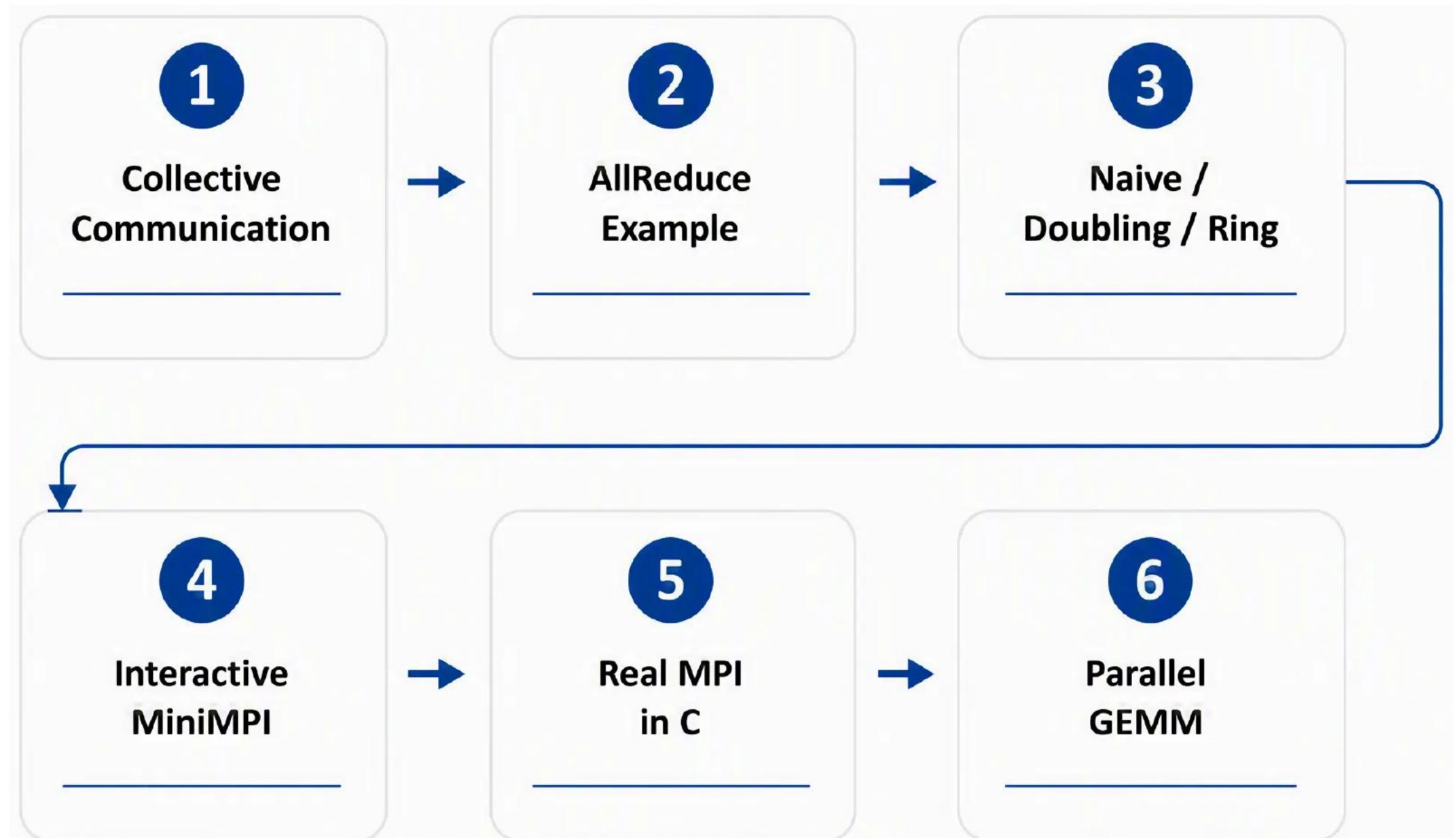
3

Compare intuitive, recursive doubling, and ring AllReduce

6

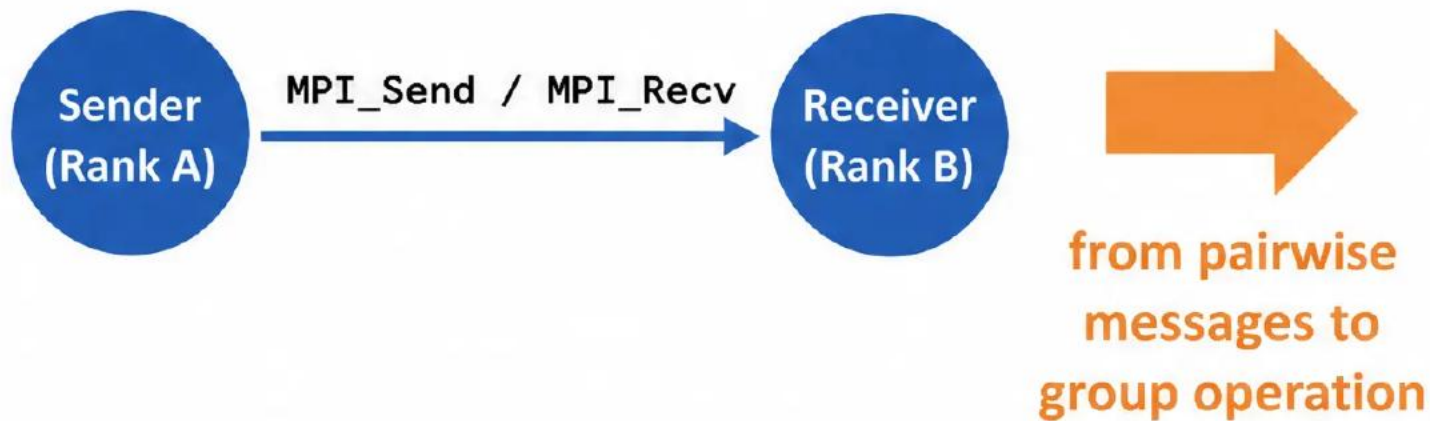
Preview Assignment 1: MPI Parallel GEMM

Our Learning Roadmap



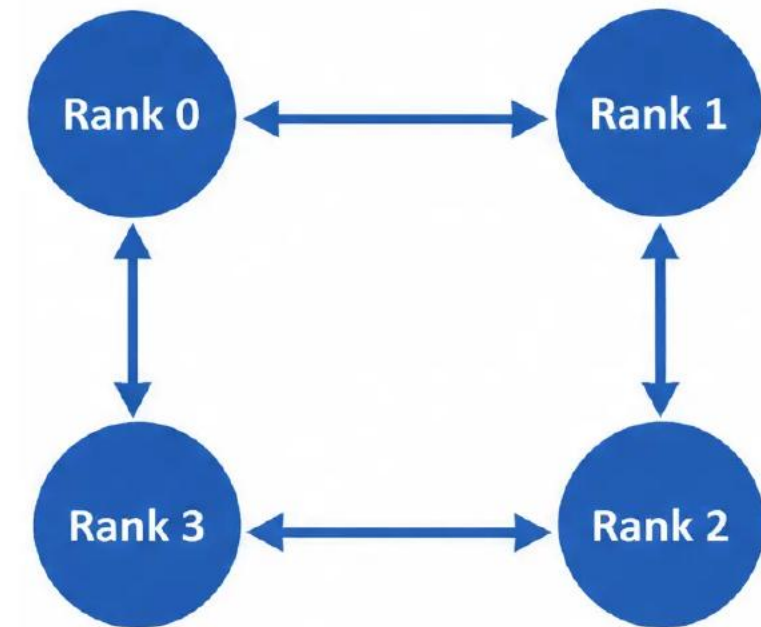
From Send/Recv to Collective

Tutorial 1: Point-to-Point



- One sender
- One receiver

Tutorial 2: Collective



- Many ranks participate together
- The group moves or combines data

Collective communication lets a group of ranks move or combine data together.

Data Movement Collectives

1) MPI_Bcast (root = R0)

BEFORE

Rank	Value
R0	7
R1	—
R2	—
R3	—

AFTER

Rank	Value
R0	7
R1	7
R2	7
R3	7

2) MPI_Scatter (root = R0)

BEFORE

Rank	Value
R0	[1, 2, 3, 4]
R1	—
R2	—
R3	—

MPI_Scatter
• root R0



AFTER

Rank	Value
R0	1
R1	2
R2	3
R3	4

3) MPI_Gather (root = R0)

BEFORE

Rank	Value
R0	1
R1	2
R2	3
R3	4



AFTER

Rank	Value
R0	[1, 2, 3, 4]
R1	—
R2	—
R3	—

4) MPI_Allgather

BEFORE

Rank	Value
R0	1
R1	2
R2	3
R3	4

MPI_Allgather



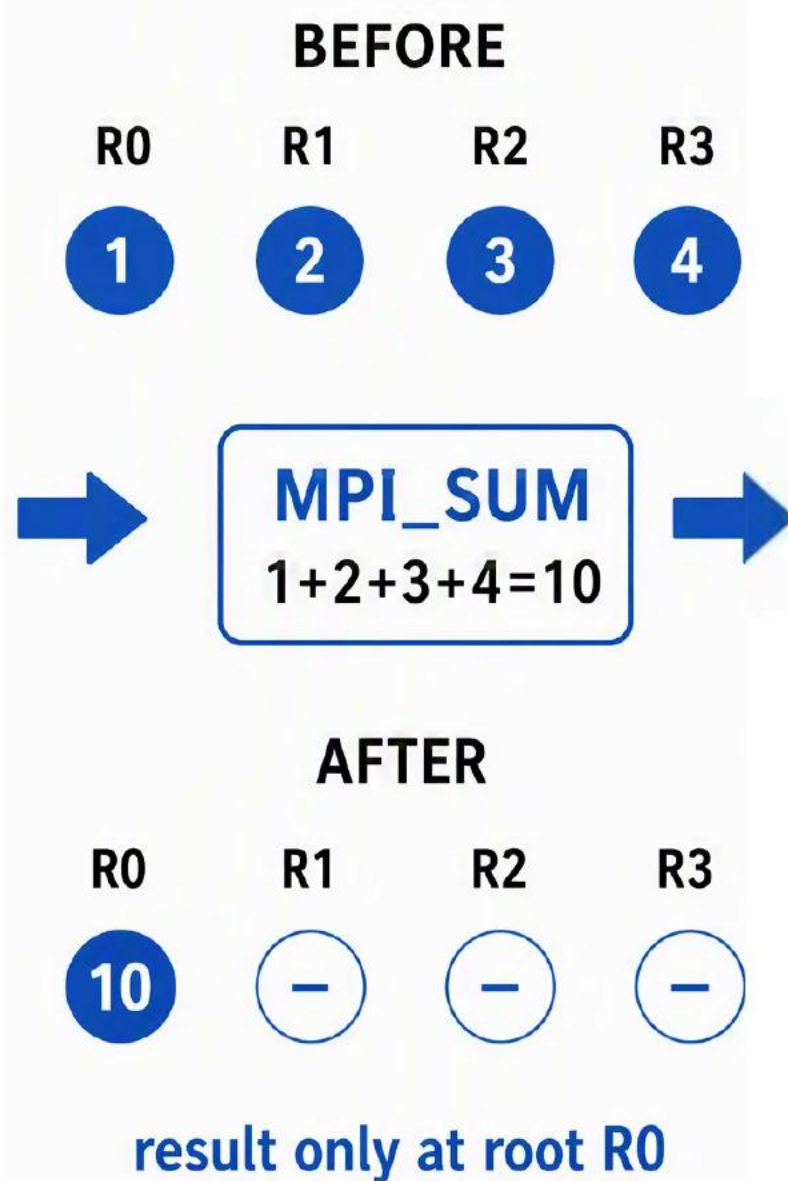
AFTER

Rank	Value
R0	[1, 2, 3, 4]
R1	[1, 2, 3, 4]
R2	[1, 2, 3, 4]
R3	[1, 2, 3, 4]

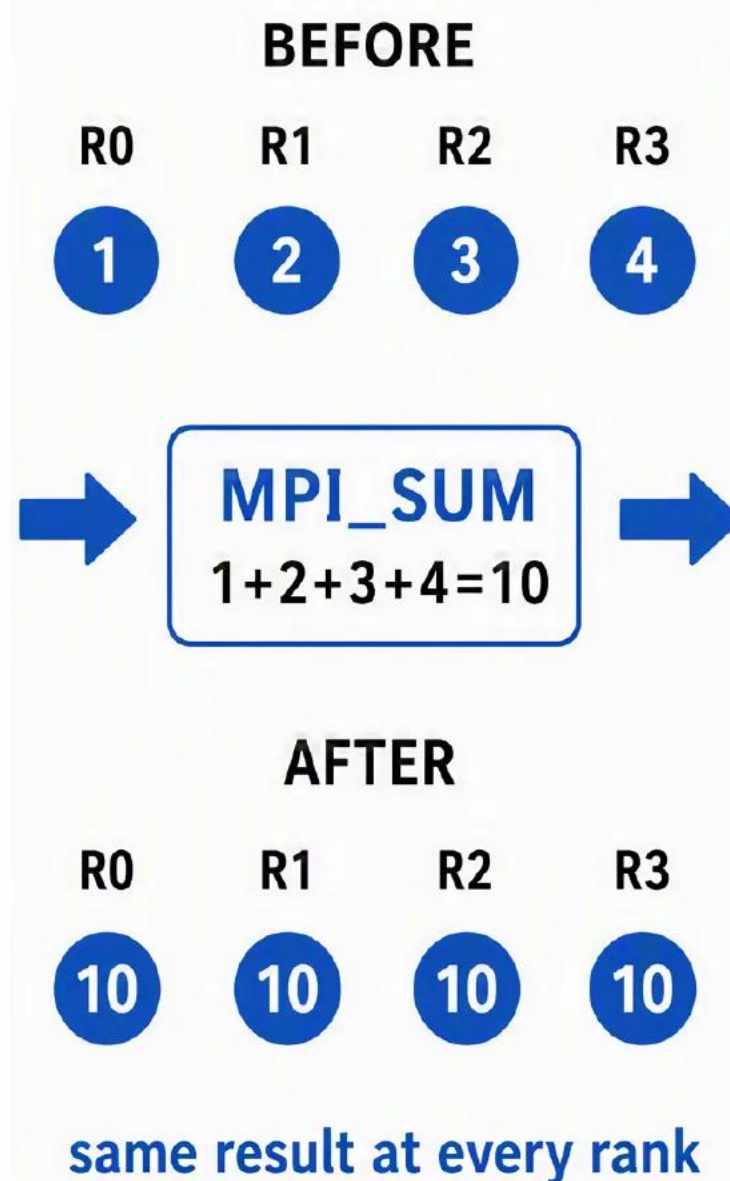
These collectives distribute or collect data.

Reduction and Synchronization Collectives

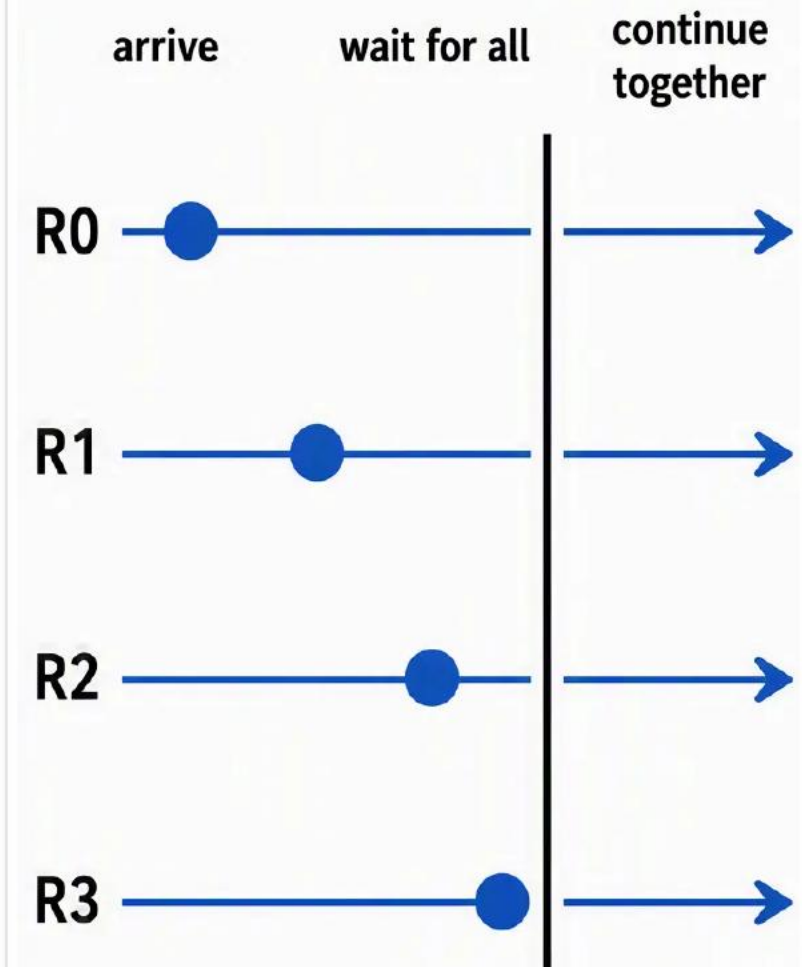
1) MPI_Reduce (SUM)



2) MPI_Allreduce (SUM)



3) MPI_Barrier



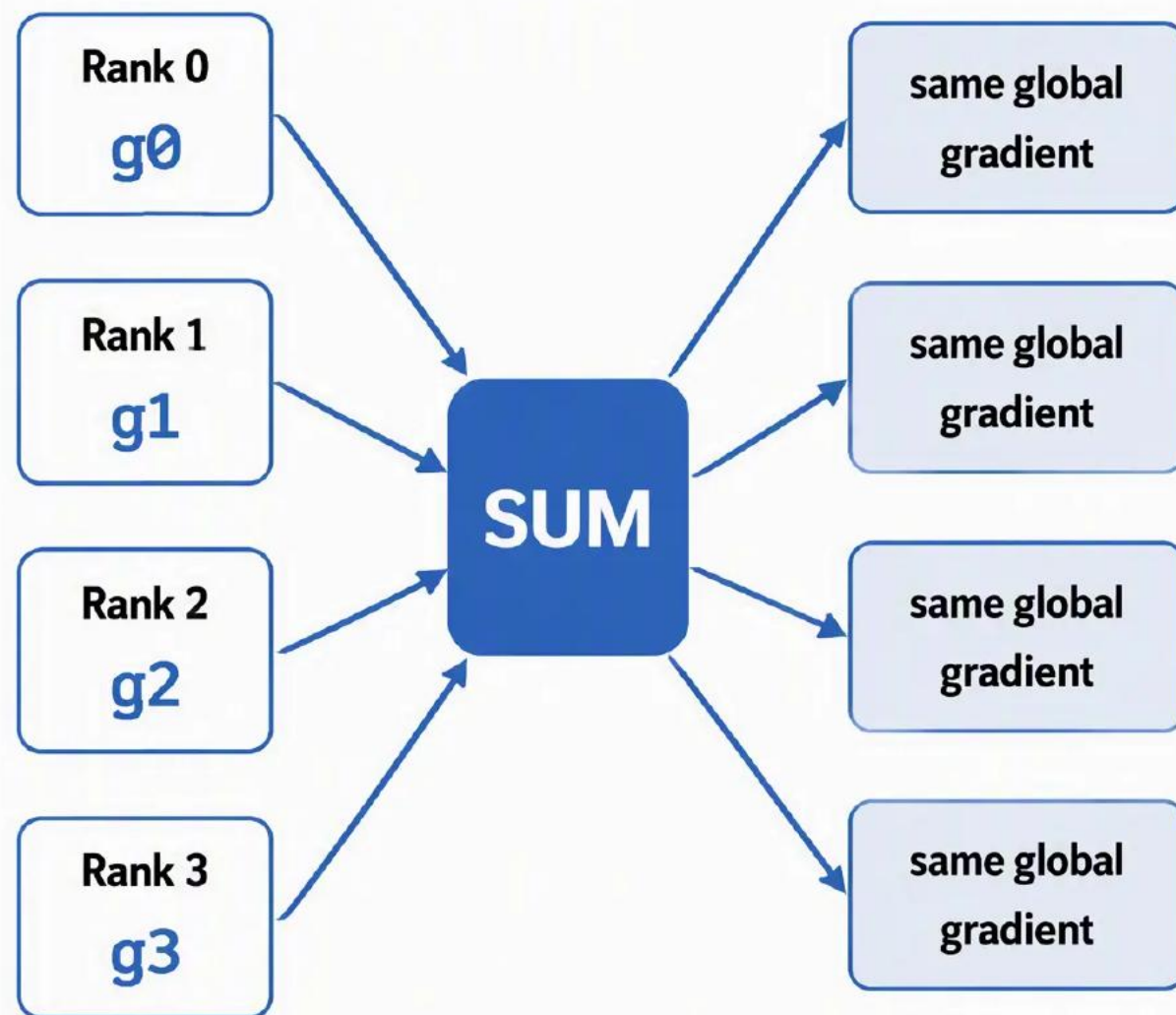
Reduce combines values · Barrier synchronizes execution

Why Focus on AllReduce?

- Rank 0 computes local gradient g_0
- Rank 1 computes local gradient g_1
- Rank 2 computes local gradient g_2
- Rank 3 computes local gradient g_3

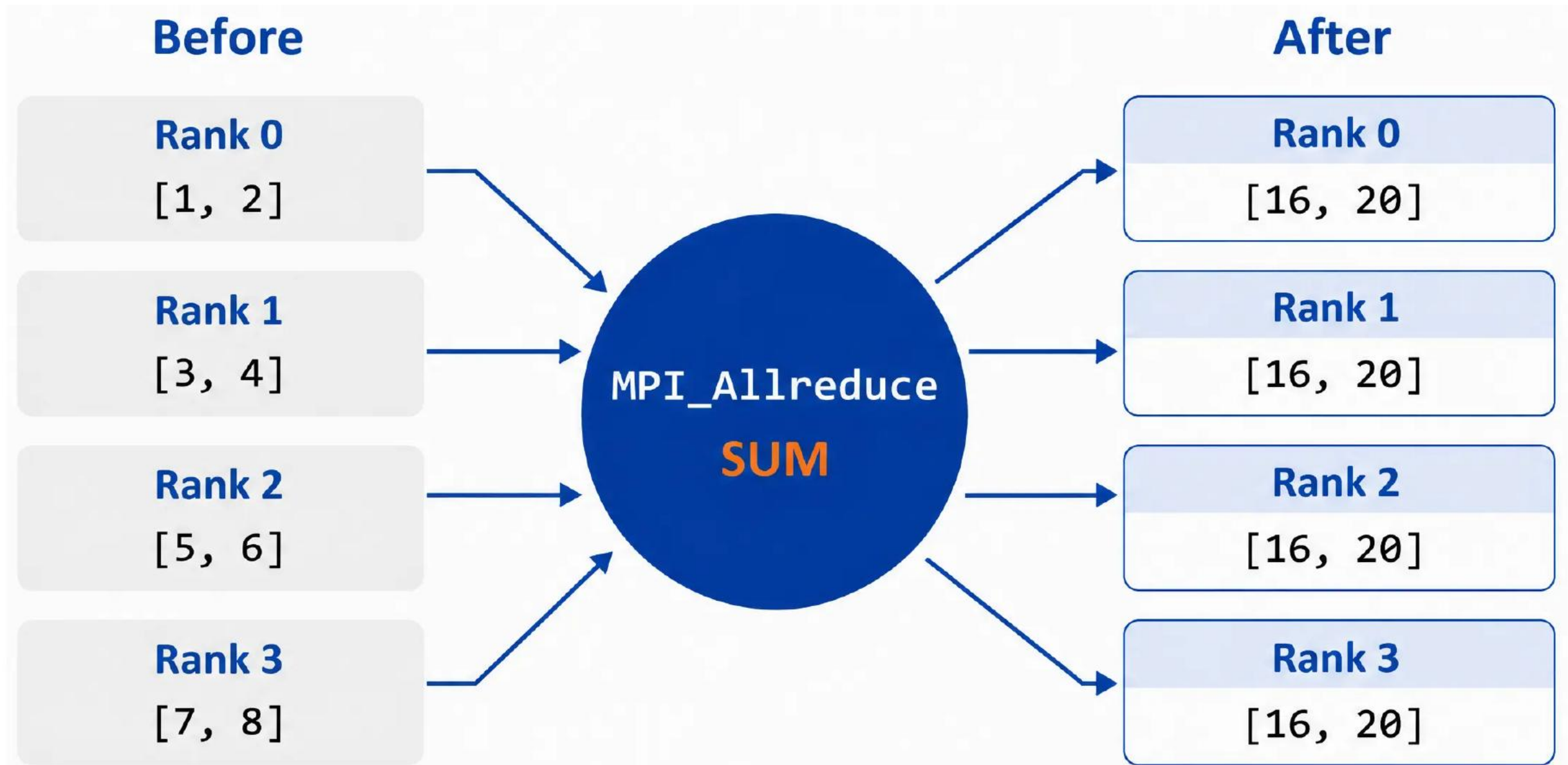
$$\text{global gradient} = g_0 + g_1 + g_2 + g_3$$

Every rank needs the same global gradient to continue training.



Gradient synchronization is a classic AllReduce workload.

What Does AllReduce Do?

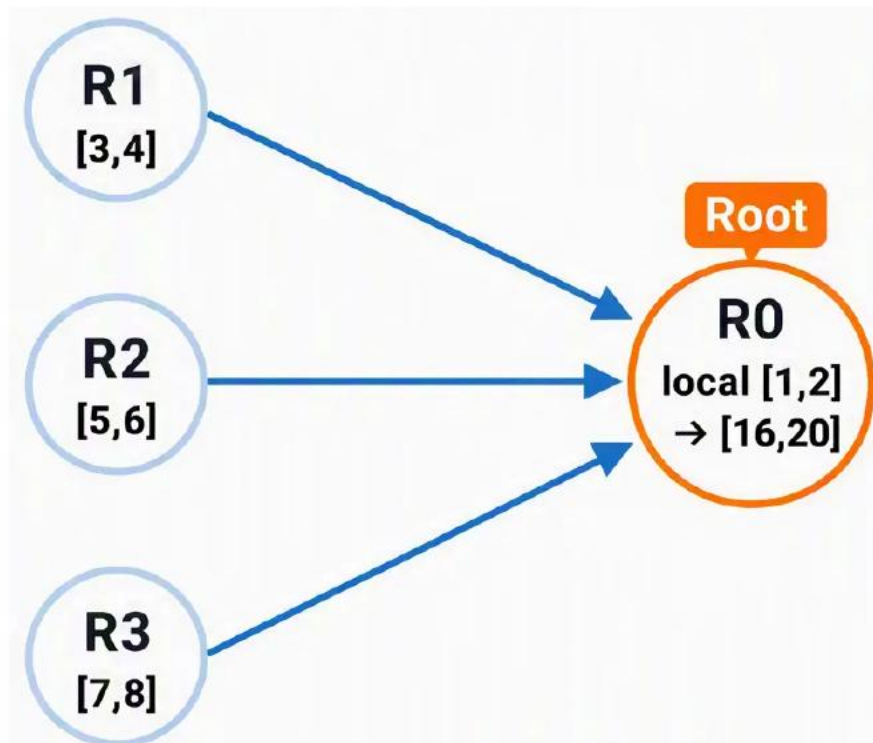


Same reduced result on every rank.

The Most Intuitive AllReduce

Step 1: Reduce to Rank 0

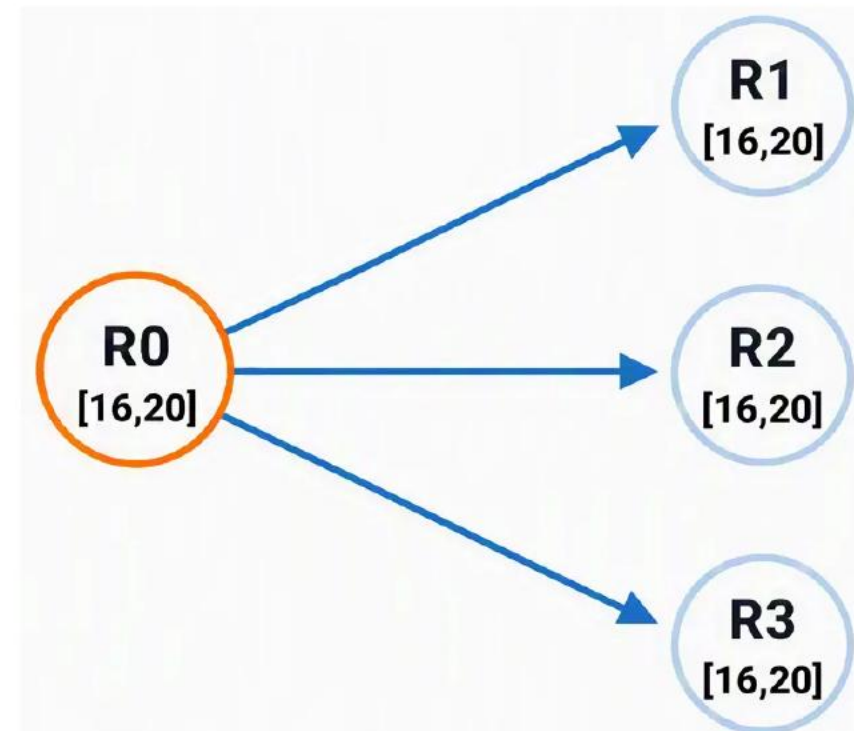
Easy to understand



$$[1,2] + [3,4] + [5,6] + [7,8] = [16,20]$$

Step 2: Broadcast from Rank 0

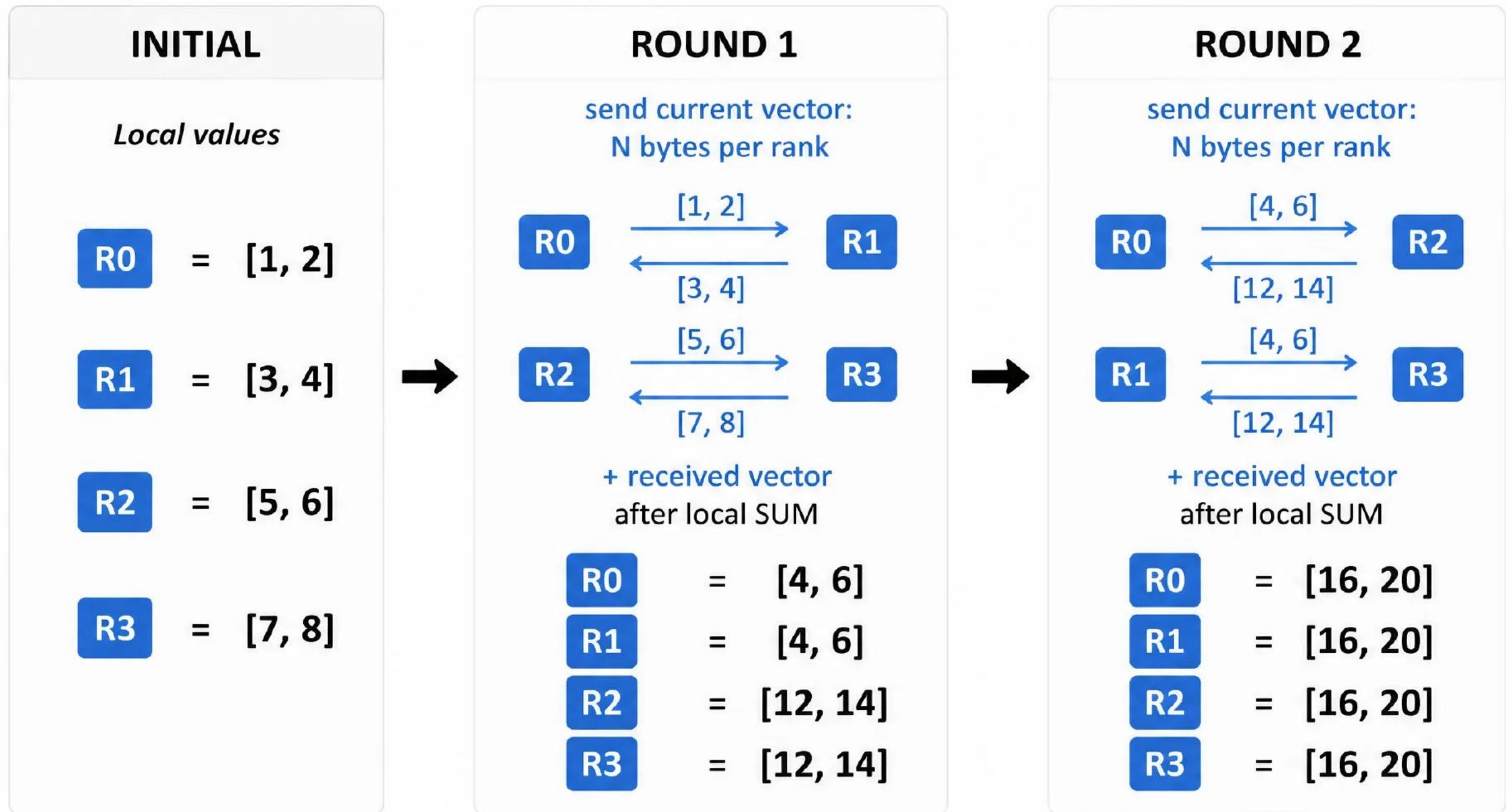
Root bottleneck
Unbalanced communication



Can all ranks participate more evenly?

Recursive Doubling AllReduce

P = 4 ranks



2 rounds = $\log_2(4)$ · Every rank finishes with [16, 20]

Ring AllReduce: Initial State

Four ranks, four chunks per local vector



R0	
C0	1
C1	2
C2	3
C3	4

R1	
C0	5
C1	6
C2	7
C3	8

R2	
C0	9
C1	10
C2	11
C3	12

R3	
C0	13
C1	14
C2	15
C3	16

C0 = 28

C1 = 32

C2 = 36

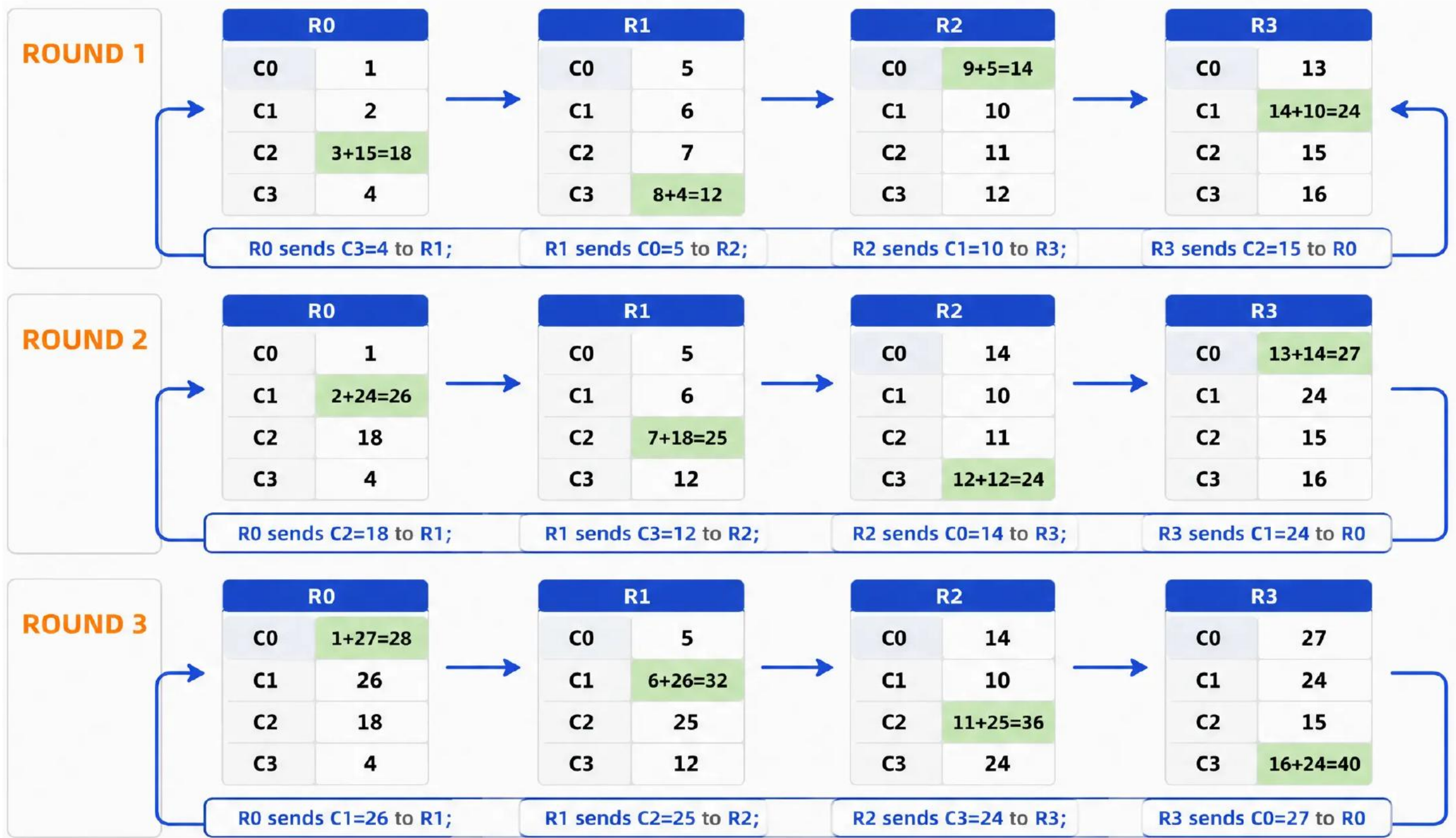
C3 = 40

Target result on every rank: [28, 32, 36, 40]

Corresponding chunks are reduced across ranks.

Ring Phase 1: Reduce-Scatter

Each round sends one chunk clockwise and adds the received value.



After Reduce-Scatter: R0 owns C0=28 · R1 owns C1=32 · R2 owns C2=36 · R3 owns C3=40

Green = updated this round

Ring Phase 2: AllGather

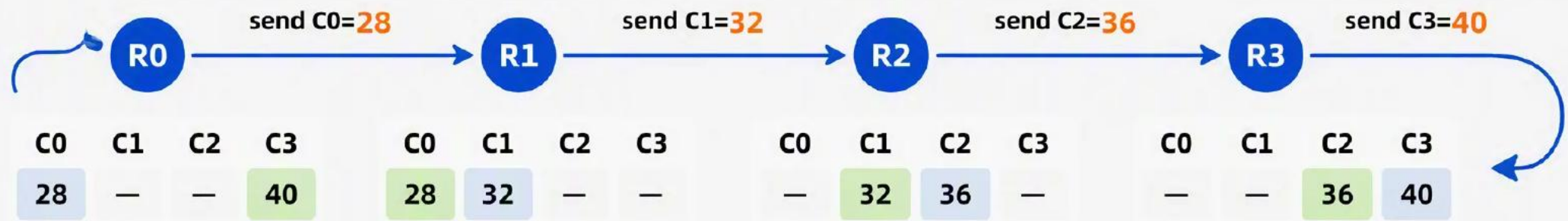
The reduced chunks circulate—no further SUM.

R0

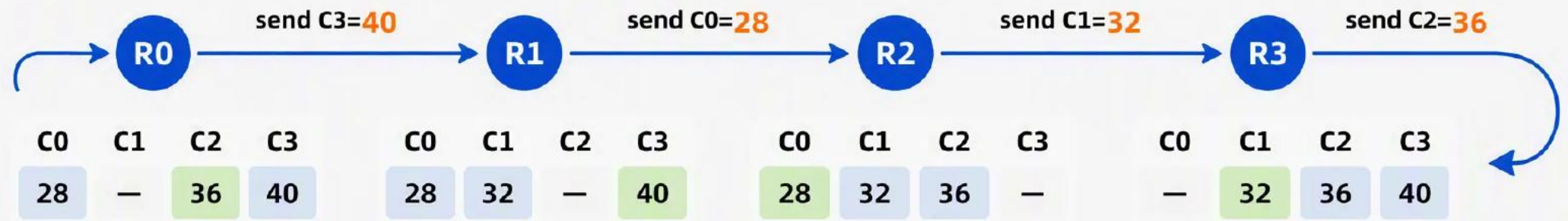
C0 C1 C2 C3

28 — — —

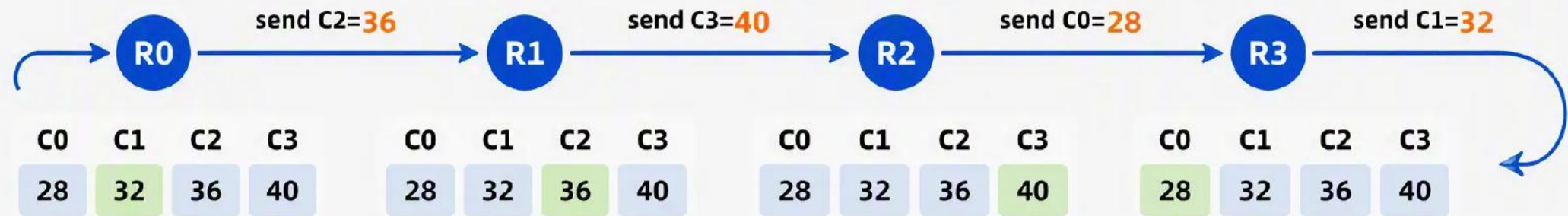
ROUND 1



ROUND 2



ROUND 3



After AllGather, every rank has all four reduced chunks.

Green = received this round · Blue = already held

Recursive Doubling: Data Volume

- $P = 4$
- Local vector size = N bytes
- Rounds = $\log_2(4) = 2$
- Sent volume per rank (outgoing only)

Sent volume per rank (outgoing only)

Round	What is sent	Bytes sent
1	Full vector (N bytes) 	N
2	Full vector (N bytes) 	N
Total sent per rank		$N + N = 2N$

All ranks send the same volume in every round

	R0	R1	R2	R3
Round 1	N	N	N	N
Round 2	N	N	N	N
Total sent per rank	$2N$	$2N$	$2N$	$2N$

Numerical example

$N = 4$ MB

Round 1: sends $N = 4$ MB

Round 2: sends $N = 4$ MB

Total sent per rank = $2N = 8$ MB

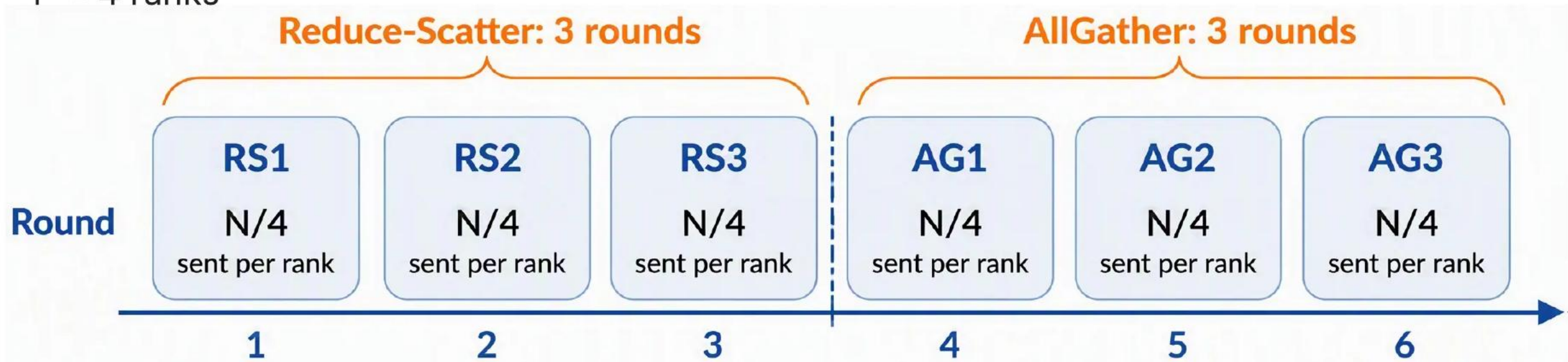


Takeaway

Few rounds, but the full vector is sent in every round.

Ring AllReduce: Data Volume

P = 4 ranks



Sent volume per rank (outgoing only)

N/4	N/4	N/4	N/4	N/4	N/4
-----	-----	-----	-----	-----	-----

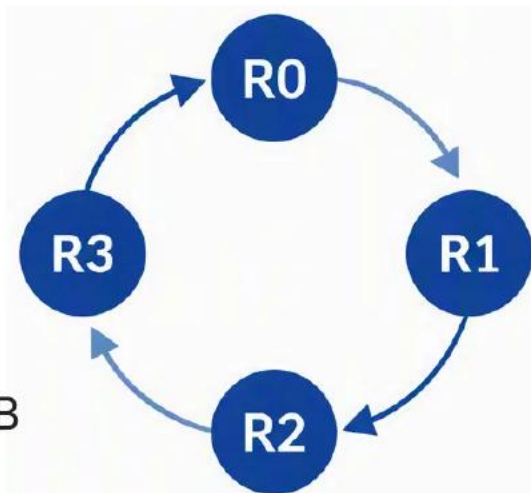
$$N/4 + N/4 + N/4 + N/4 + N/4 + N/4 = 6N/4 = 1.5N$$

Numerical example

Total data to reduce: $N = 4$ MB

Data sent per rank per round: $N/4 = 1$ MB

Total sent per rank (6 rounds): $6 \times 1 \text{ MB} = 6 \text{ MB}$



Key takeaway

**More rounds,
less data per round.**

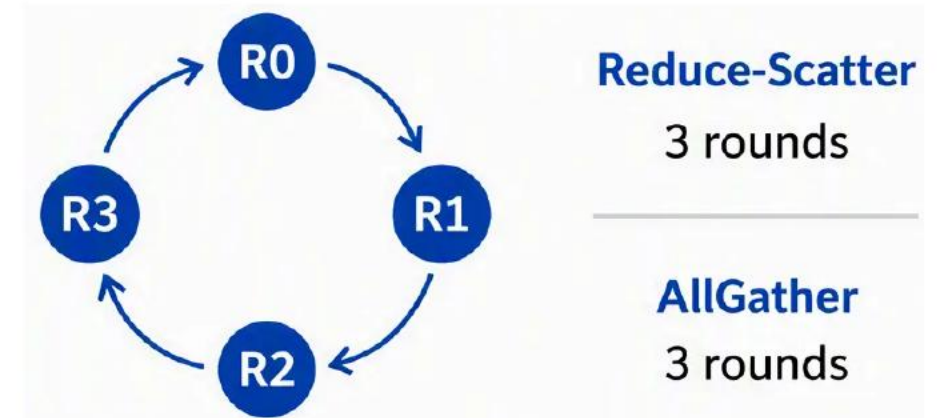
AllReduce: Recursive Doubling vs Ring

Recursive Doubling



full vector N per round

Ring (AllReduce)



Reduce-Scatter
3 rounds

AllGather
3 rounds

one chunk $N/4$ per round

Metric ($P = 4$)	Recursive Doubling	Ring (AllReduce)
Rounds	2	6
Data per round	N	$N/4$
Sent volume per rank (outgoing only)	$2N$	$1.5N$
For $N = 4$ MB vector (sent volume per rank, outgoing only)	8 MB	6 MB
Main strength	fewer rounds	less data movement

Recursive Doubling saves latency. Ring saves bandwidth.

Which One Is Faster?

$$T \approx \alpha \times \text{rounds} + \beta \times \text{bytes}$$

α : startup / latency cost

β : cost per byte transferred



Small message

latency dominates

fewer rounds may win



Large message

bandwidth dominates

less data movement may win

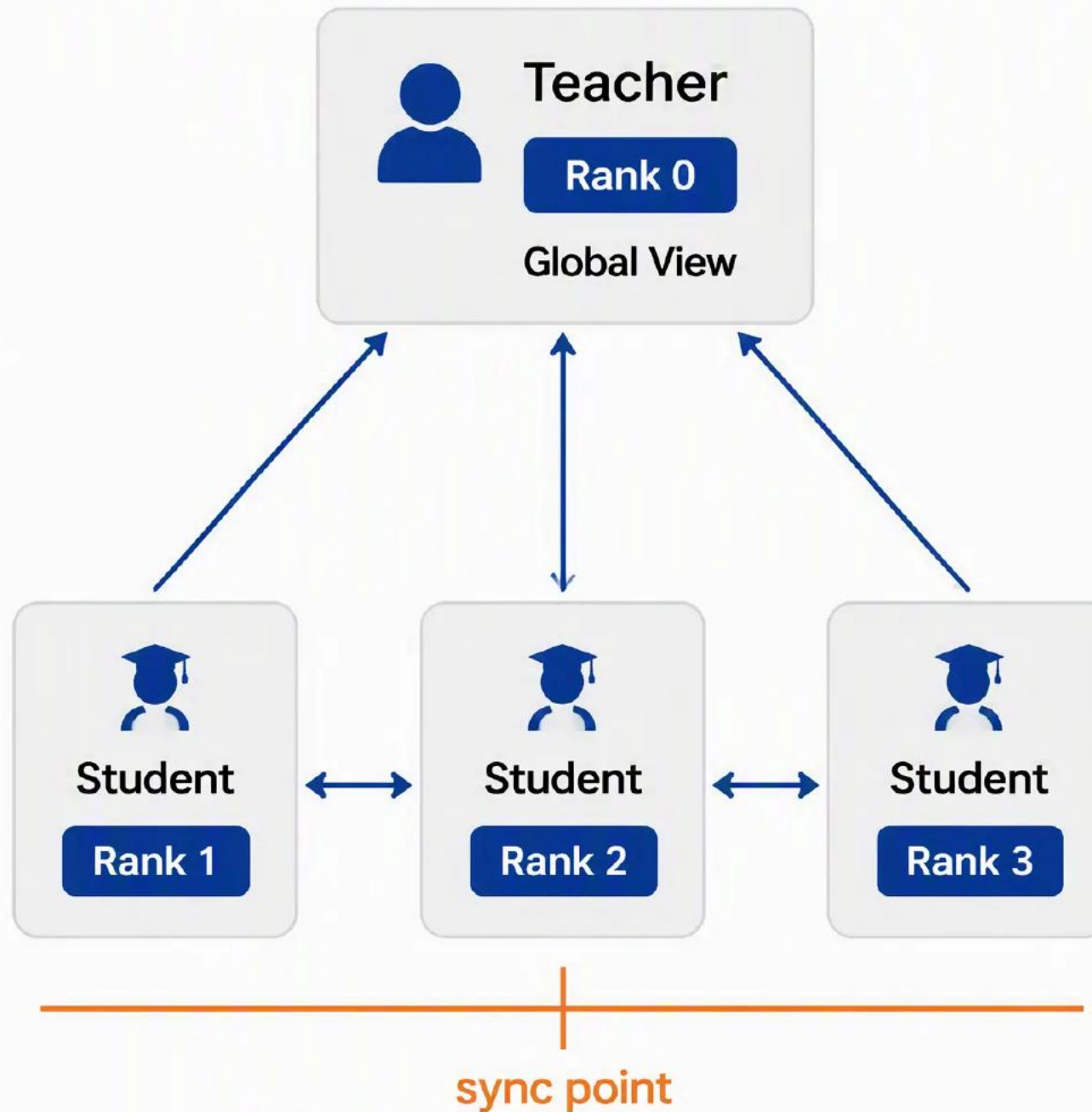
Next: observe it interactively.

Interactive MiniMPI Demo

Roles

- Teacher = Rank 0 + Global View

- Students = Rank 1 / Rank 2 / Rank 3



Observe

- ✓ Who sends to whom?
- ✓ What data is sent?
- ✓ What data is received?
- ✓ How does local data change?
- ✓ When does synchronization happen?

Demo Guide

Recommended demonstration order

1

Recursive Doubling AllReduce

- $P = 4$
- 2 rounds
- Full-vector exchange
- Every rank sends and receives



2

Ring AllReduce

- $P = 4$
- 6 rounds
- Chunk movement
- Reduce-Scatter + AllGather



3

Benchmark

- Compare small vs large data sizes
- Relate results to latency and bandwidth

small	
large	

From MiniMPI to Real MPI

MiniMPI exposes:

1. Round
2. Send / Receive
3. SUM
4. Synchronization

abstracts into

Native MPI collective API

```
MPI_Allreduce(  
    sendbuf,  
    recvbuf,  
    count,  
    MPI_INT,  
    MPI_SUM,  
    MPI_COMM_WORLD  
);
```



**MiniMPI shows possible implementations;
real MPI hides them behind collective APIs.**

Assignment 1 — Parallel GEMM with MPI

Parallel GEMM with MPI is our first assignment.



The challenge

Turn matrix multiplication into a correct, coordinated program that runs across multiple MPI ranks.



Decomposition

Divide work across ranks



Collectives

Move inputs and results



Correctness

Check against a trusted result



Performance

Reason about computation, communication, and scaling



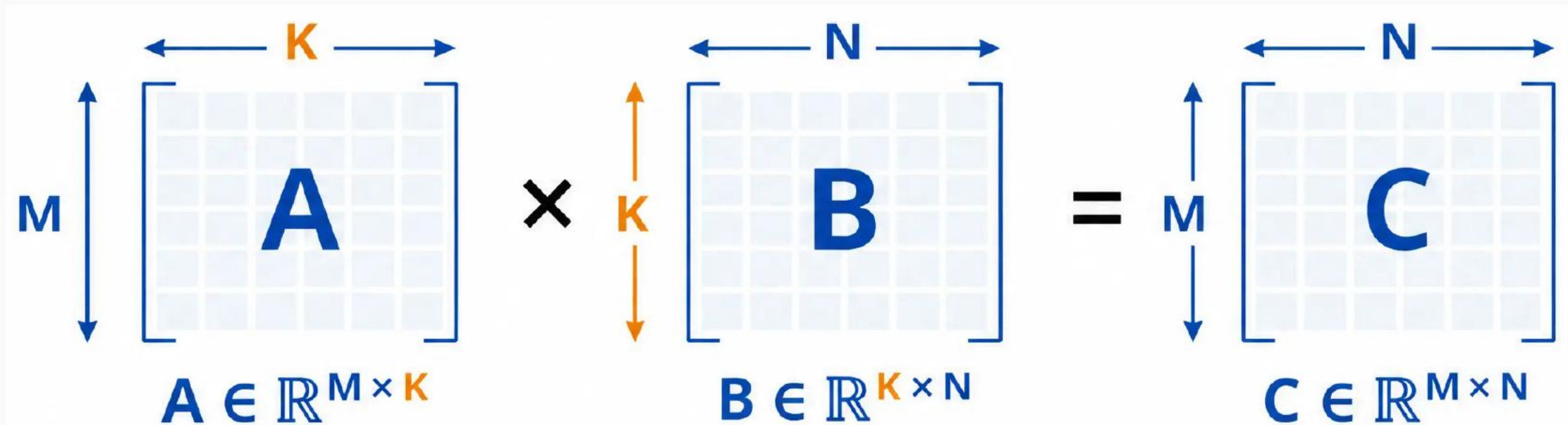
A common systems/HPC interview-style problem: explain your data layout and communication choices, then validate them.



Detailed input/output format, checker, and grading rubric will be released later.

GEMM: General Matrix–Matrix Multiplication

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$



$$(M \times K) \cdot (K \times N) \rightarrow (M \times N)$$

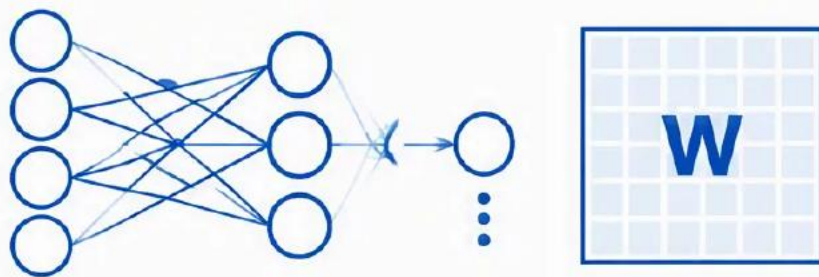
$$C_{ij} = \sum_{k=1}^K A_{ik} B_{kj}$$

Each C_{ij} is the dot product of row i of $\mathbf{A}$ and column j of $\mathbf{B}$.

Where Is GEMM Used?

$Y = XW$ — many dot products computed together

1. LLMs & Deep Learning



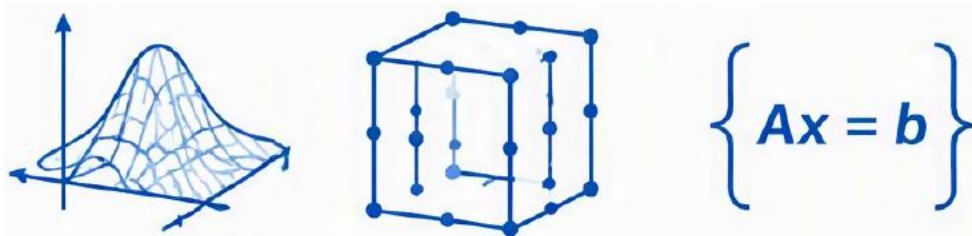
Linear layers · Attention projections ·
MLP layers · Training and inference

2. Computer Vision



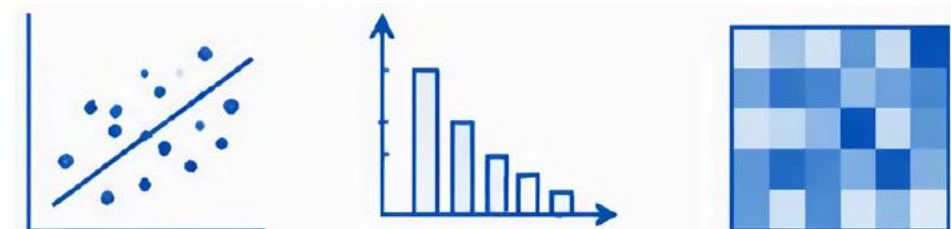
Convolutions are often transformed
into matrix multiplication.

3. Scientific Computing



Numerical linear algebra ·
Simulations · Solvers

4. Data Analysis



Least squares · PCA ·
Dense transformations

GEMM is a core building block behind many compute-intensive workloads.

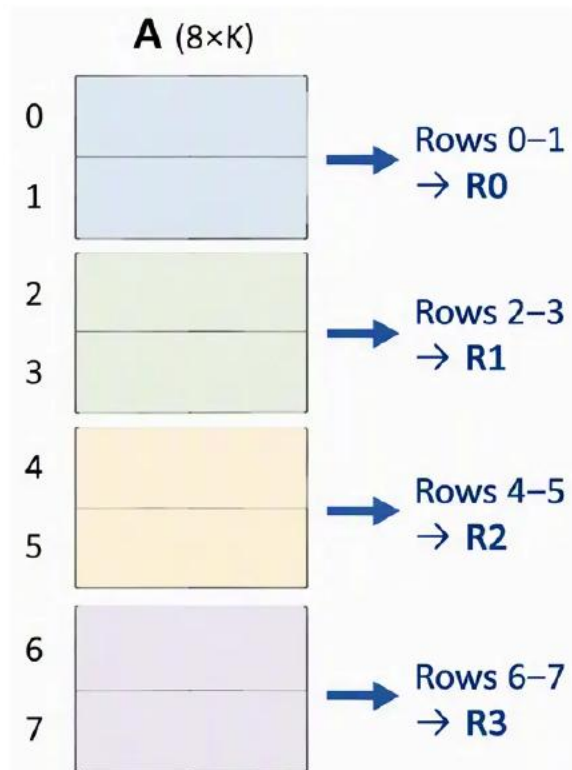
Parallel GEMM: Row Partitioning

Example: 8 rows of A across 4 MPI ranks

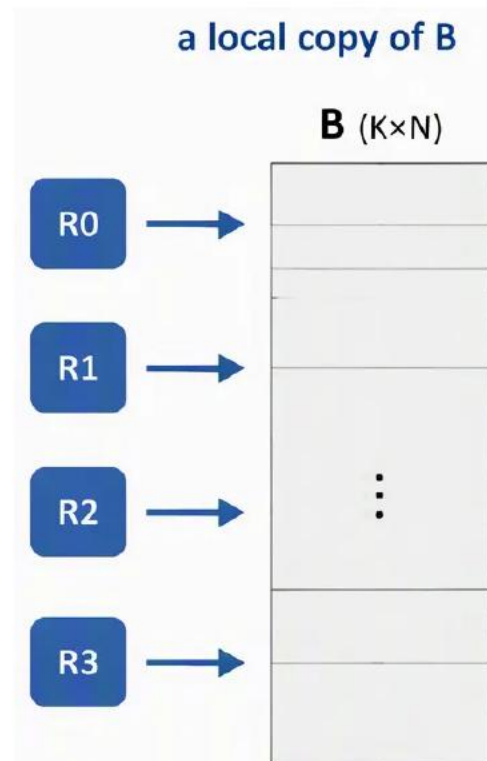
$$C = A \times B$$

Dimensions: $A(8 \times K) \times B(K \times N) = C(8 \times N) \rightarrow$ Each rank computes a $(2 \times N)$ block.

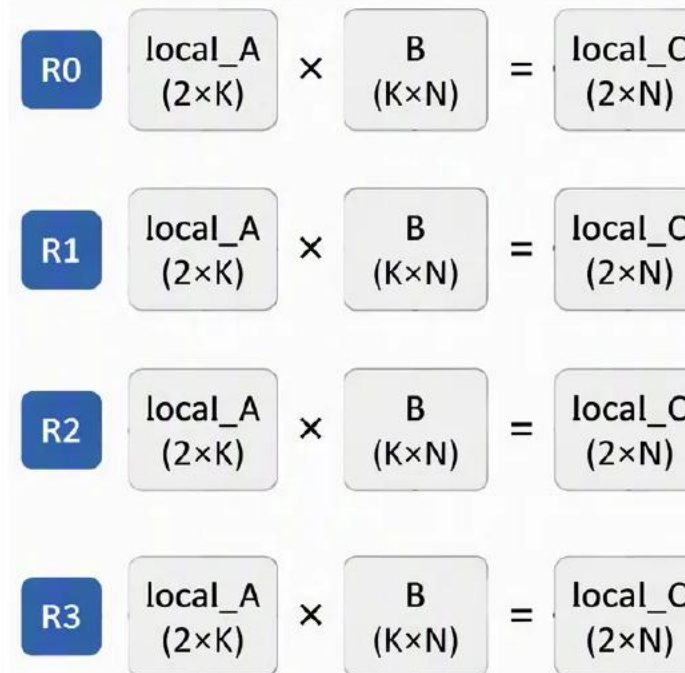
1 Partition A by rows



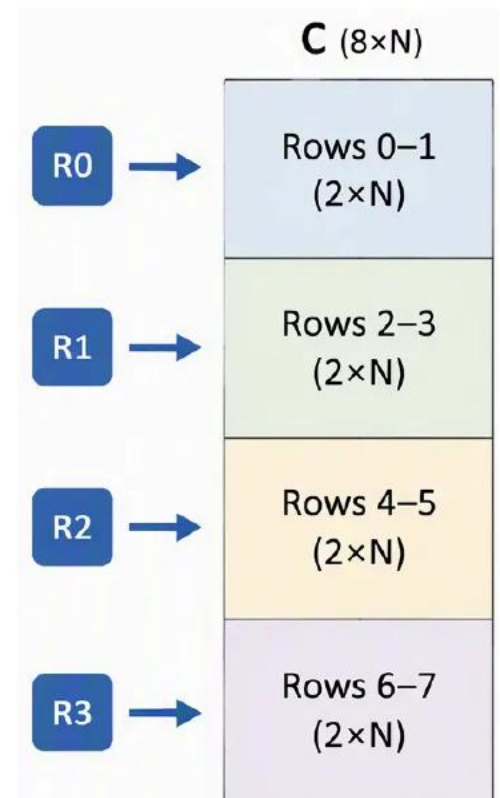
2 Every rank uses the full B



3 Compute independently



4 Row blocks form C

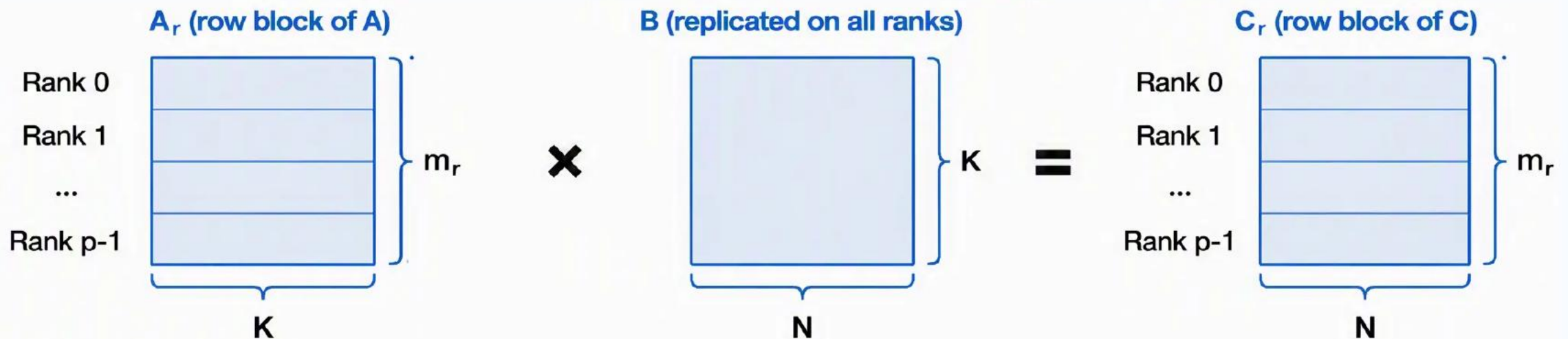
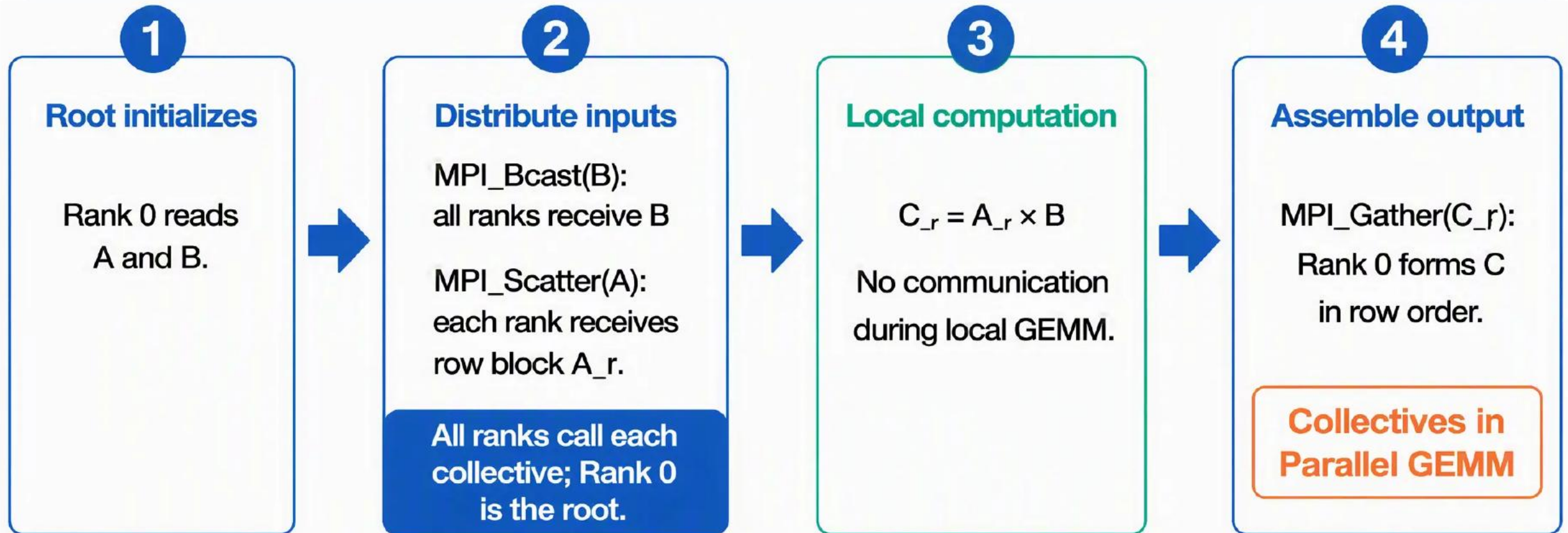


Assembled in row order

Once **A** is partitioned and **B** is available, the four local GEMMs can run at the same time.

Collectives in Parallel GEMM

B is replicated; rows of A and C are distributed.



This example assumes equal row blocks; use Scatterv/Gatherv for uneven rows.