LLM Sys

11868/11968 LLM Systems
Distributed Training

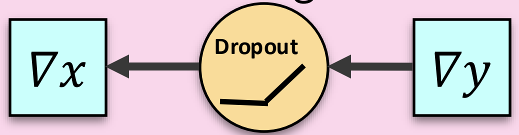
Lei Li

Carnegie Mellon University
Language Technologies Institute

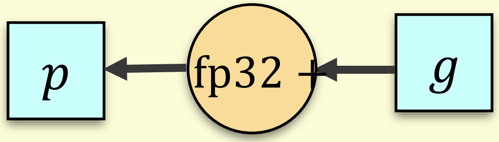
Recap: Accelerating Transformer Layers

We optimize the training process from 3 aspects

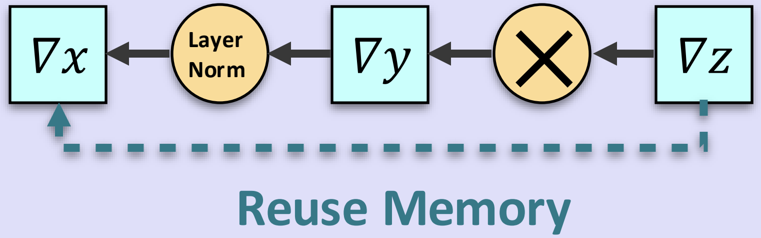
High performance Kernels for Forward and Backward via Fusion & Algebra trick



Efficient Parameter Update via mixed precision



Memory Management



Operators that can be reused in Other Networks:
Dropout, LayerNorm, Softmax, Cross Entropy

Accelerating decoding

hierachical auto-regressive search for large vocabulary converting sorting to parallel operations (max, filter, re-rank)

Deepseek opensource libraries

- FlashMLA (released 2/24/2025)
  - FlashMLA is an efficient MLA decoding kernel for Hopper GPUs, optimized for variable-length sequences serving.

- DeepEP (released 2/25/2025)
  - a communication library tailored for Mixture-of-Experts (MoE) and expert parallelism (EP). It provides high-throughput and low-latency all-to-all GPU kernels, which are also as known as MoE dispatch and combine.

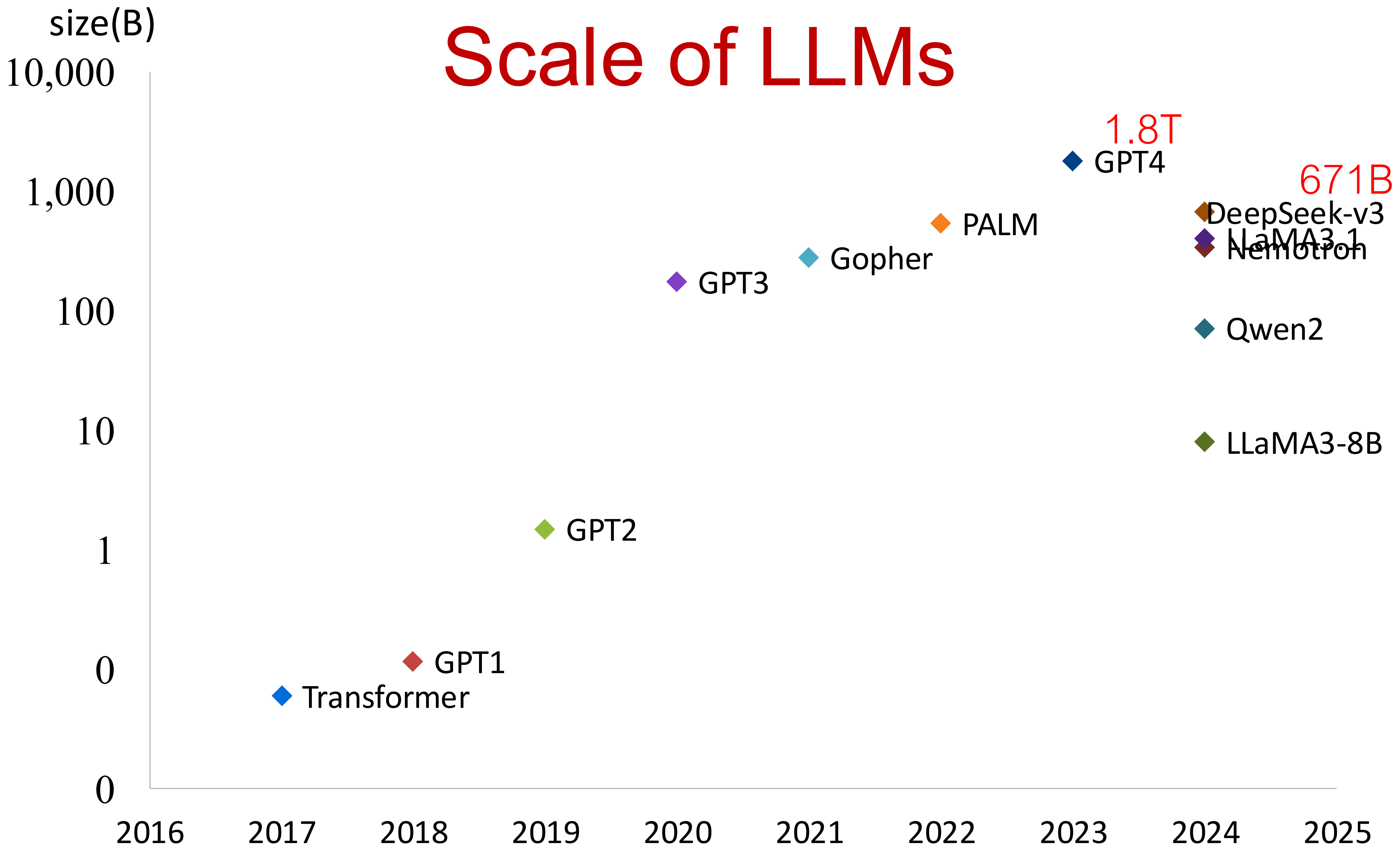
- DeepGEMM (released 2/26/2025)
  - DeepGEMM is a library designed for clean and efficient FP8 General Matrix Multiplications (GEMMs) with fine-grained scaling

https://github.com/deepseek-ai/

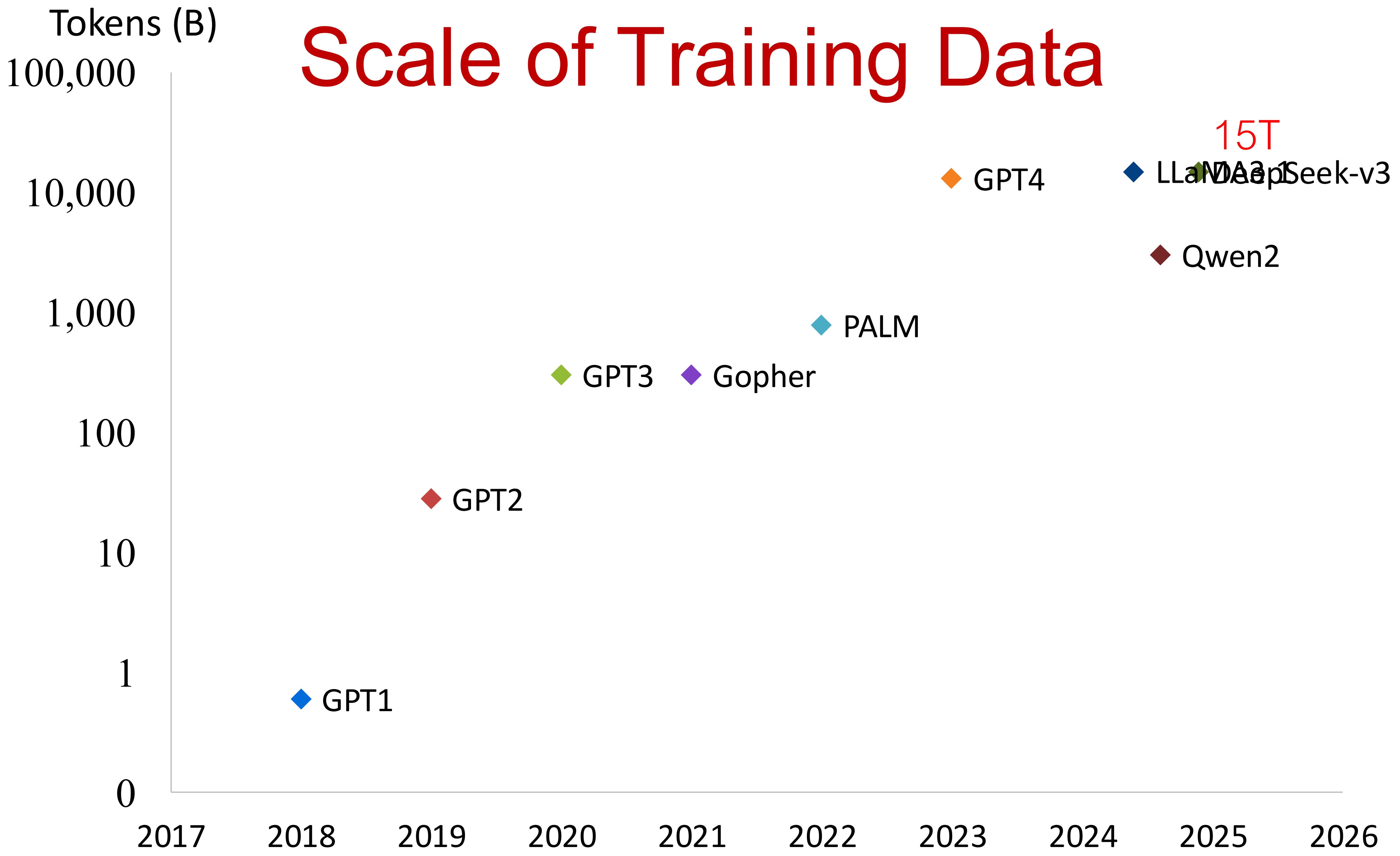
Outline

- Overview of large-scale model training
- Multi-GPU communication
- Data Parallel Training via AllReduce

Scale of LLMs



Scale of Training Data



Large-scale Distribution Training

- Pretraining for Deepseek V3 (671B)
  - 2,048 H800 GPUs
  - trained for 2 months
  - a total of 2.664 million H800 GPU hours
- LLaMA 3.1(405B)
  - using 16,000 H100 GPUs
  - a total of 30.84 million GPU hours

Strategies for Scalable Training

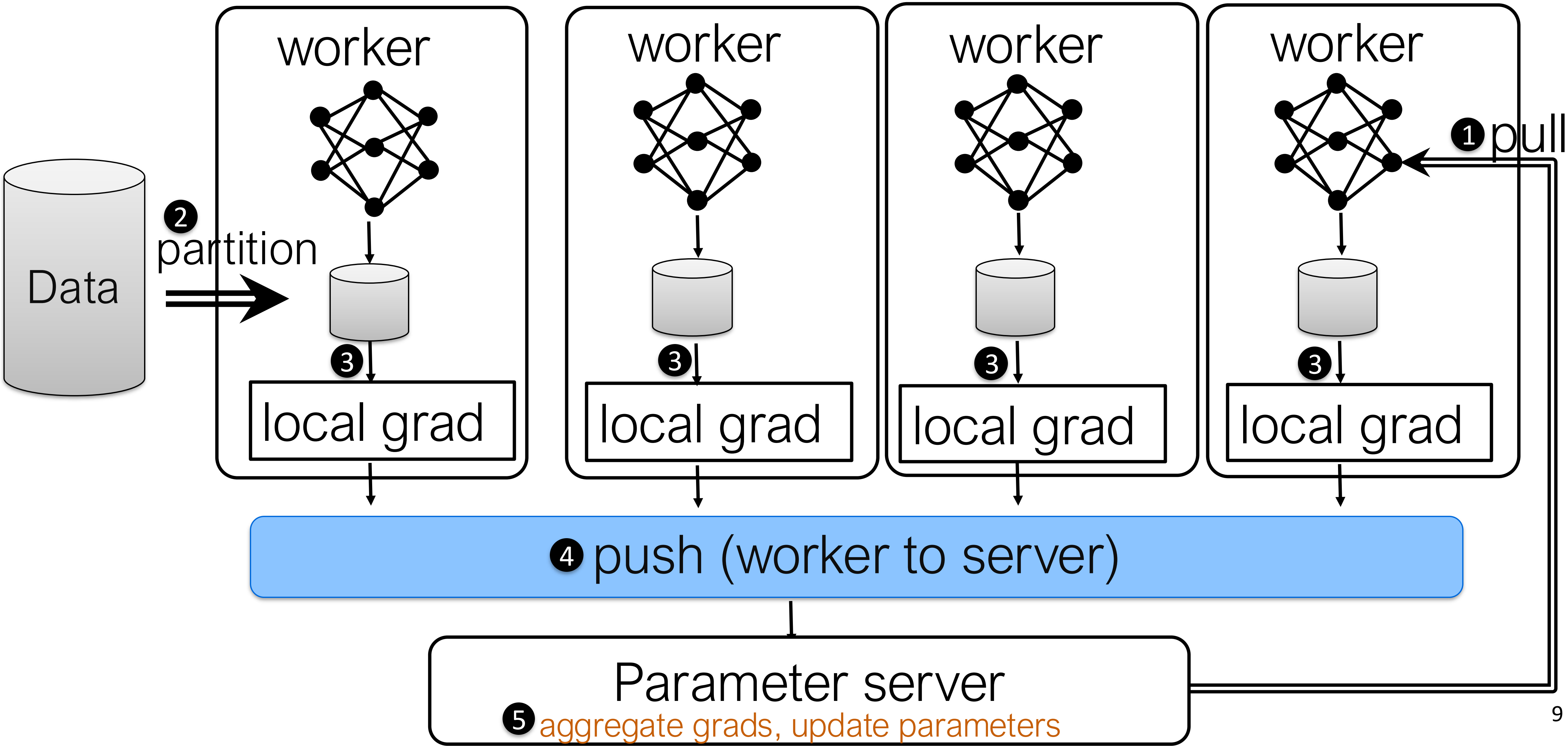
Partition the data

- single node data parallel
- distributed data parallel
- parameter server

Partition the Model

- Model parallel
- Pipeline parallel
- Tensor parallel

Classical Distributed Training: Parameter Server



Outline

- Overview of large-scale model training
- Multi-GPU communication
- Data Parallel Training via AllReduce

Multi-GPU Communication

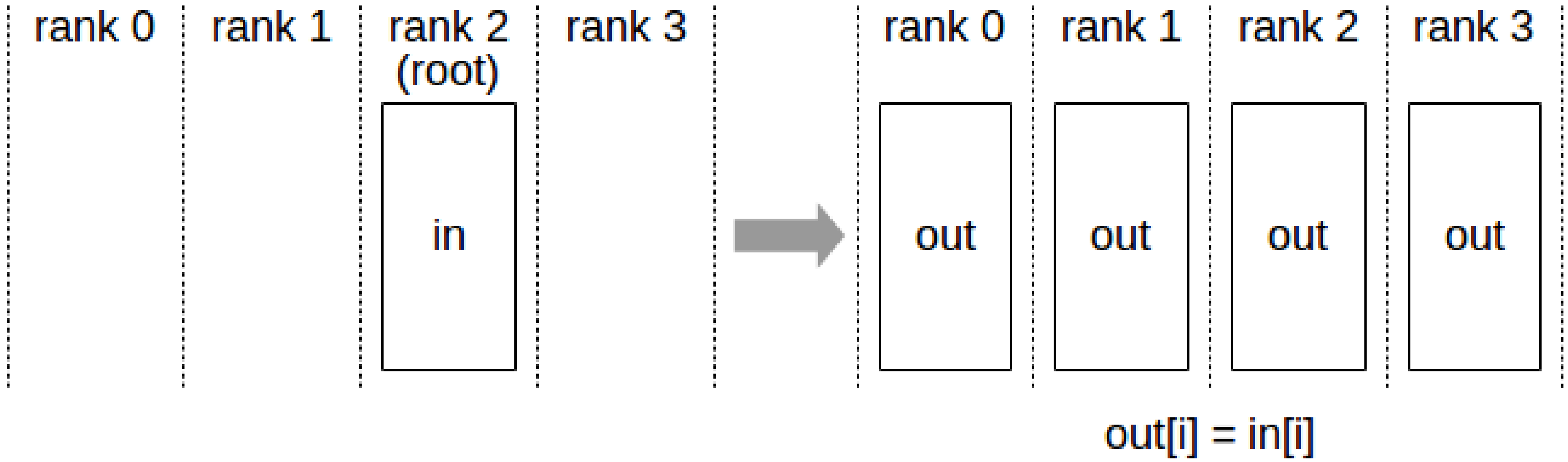
- NCCL (Nvidia Collective Communication Library)
  - provides inter-GPU communication APIs
  - both collective and point-to-point send/receive primitives
  - supports various of interconnect technologies
    - PCIe
    - NVLink
    - InfiniBand
    - IP sockets
  - Operations are tied to a CUDA stream.

NCCL Primitives

- Broadcast
- Reduce
- ReduceScatter
- AllGather
- AllReduce

Broadcast

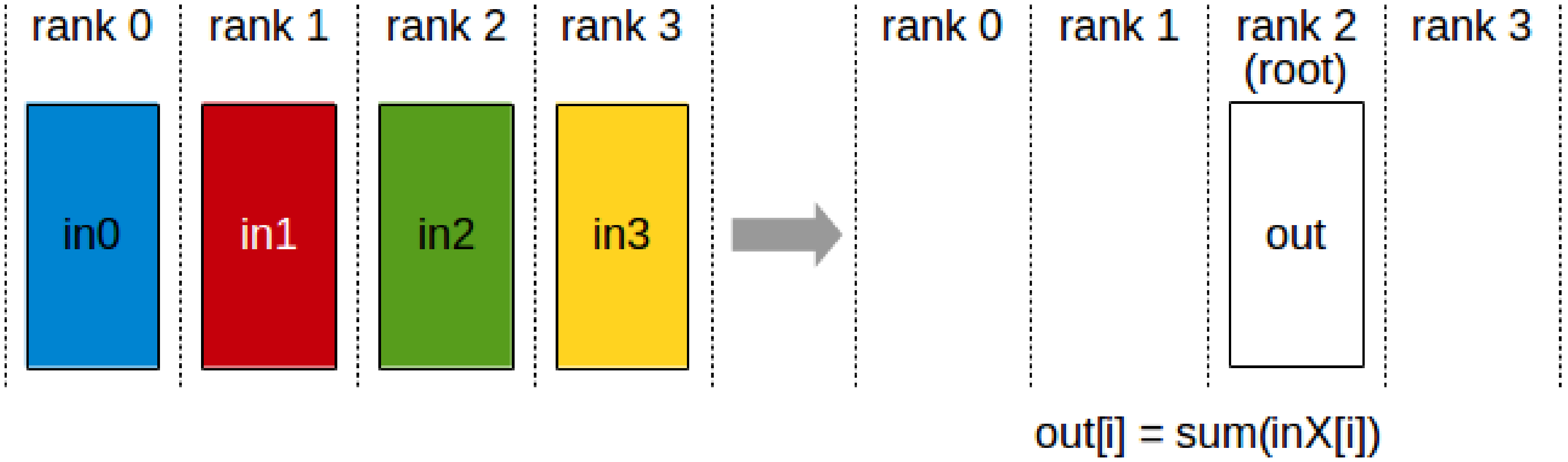
- The Broadcast operation copies an N-element buffer on the root rank to all ranks (devices).



ncclResult_t ncclBroadcast(const void* *sendbuff*, void* *recvbuff*, size_t *count*, ncclDataType_t *datatype*, int *root*, ncclComm_t *comm*, cudaStream_t *stream*)

Reduce

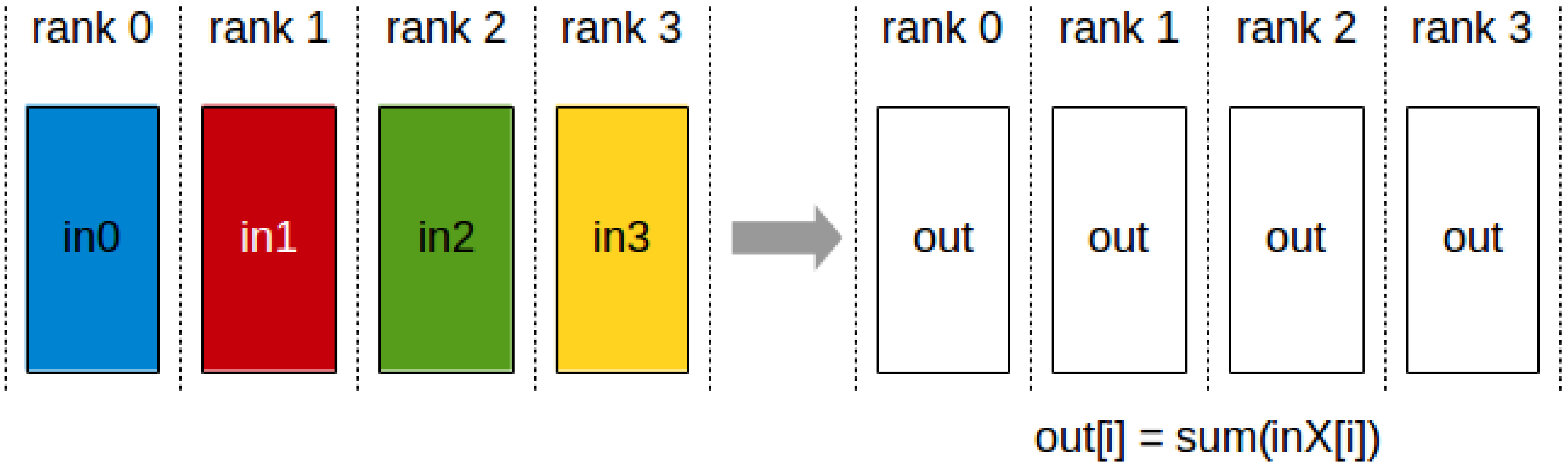
- Compute reduction (max, min, sum) across devices and write on one rank (device)



ncclResult_t ncclReduce(const void* *sendbuff*, void* *recvbuff*, size_t *count*, ncclDataType_t *datatype*, ncclRedOp_t *op*, int *root*, ncclComm_t *comm*, cudaStream_t *stream*)

AllReduce (=Reduce & Broadcast)

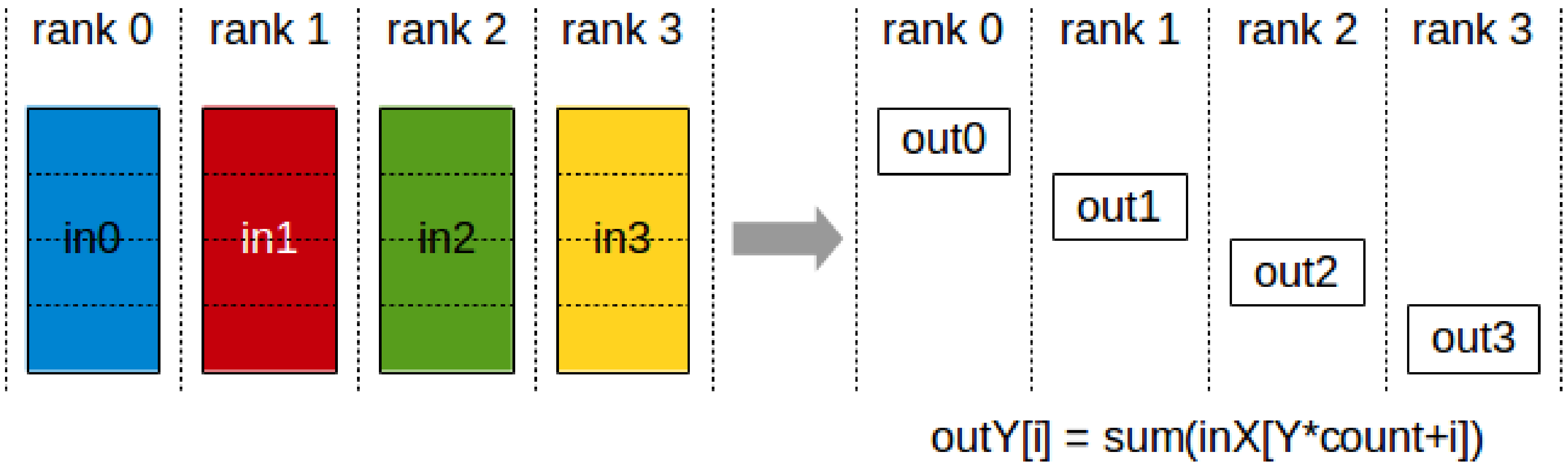
- Compute reduction (sum, min, max) across devices and writing the result in the receive buffers of every rank.



ncclResult_t ncclAllReduce(const void* *sendbuff*, void* *recvbuff*, size_t *count*, ncclDataType_t *datatype*, ncclRedOp_t *op*, ncclComm_t *comm*, cudaStream_t *stream*)

ReduceScatter

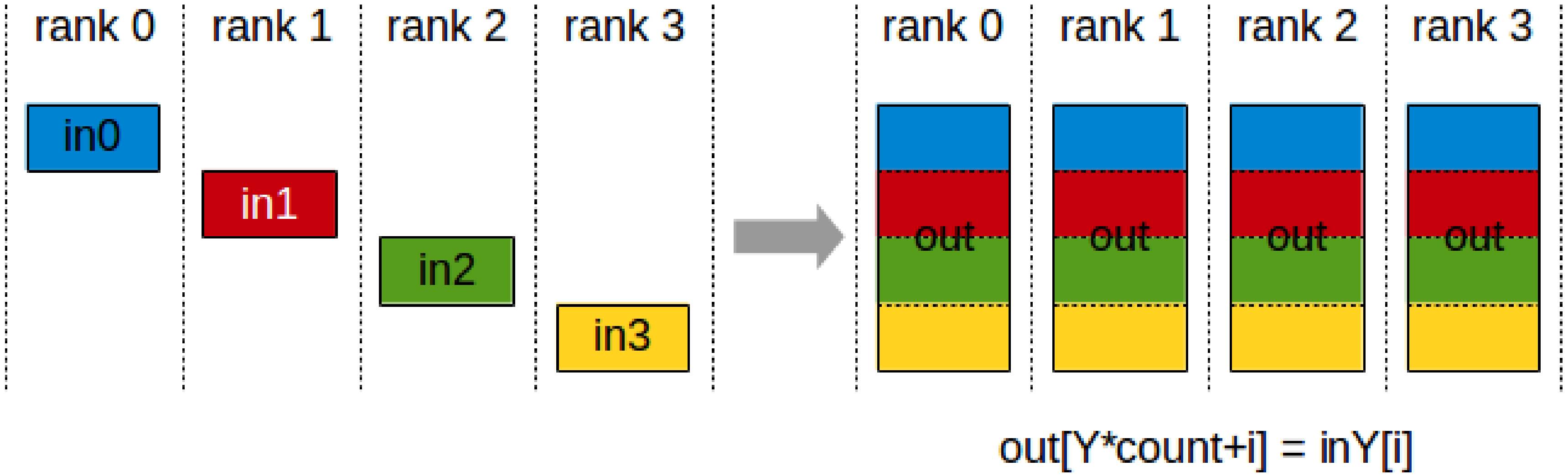
- Compute reduction (sum, min, max) and writing parts of results scattered in ranks



ncclResult_t ncclReduceScatter(const void* *sendbuff*, void* *recvbuff*, size_t *recvcount*, ncclDataType_t *datatype*, ncclRedOp_t *op*, ncclComm_t *comm*, cudaStream_t *stream*)

AllGather

- gathers N values from k ranks into an output of size k*N, and distributes that result to all ranks (devices).



ncclResult_t ncclAllGather(const void* *sendbuff*, void* *recvbuff*, size_t *sendcount*, ncclDataType_t *datatype*, ncclComm_t *comm*, cudaStream_t *stream*)

AllReduce = ReduceScatter & AllGather

Data Pointers in CUDA

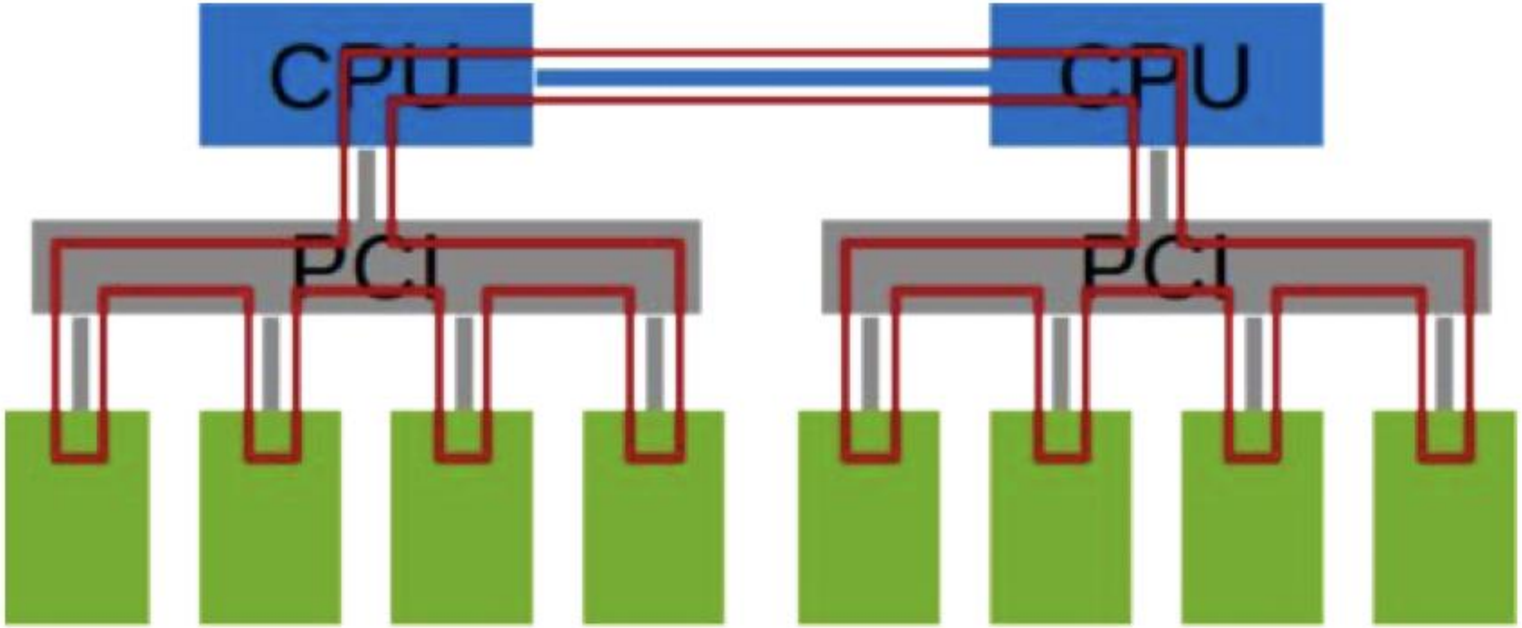
- device memory local to the CUDA device
- host memory registered using cudaHostRegister or cudaGetDevicePointer
- managed and unified memory.

Point-to-Point Communication

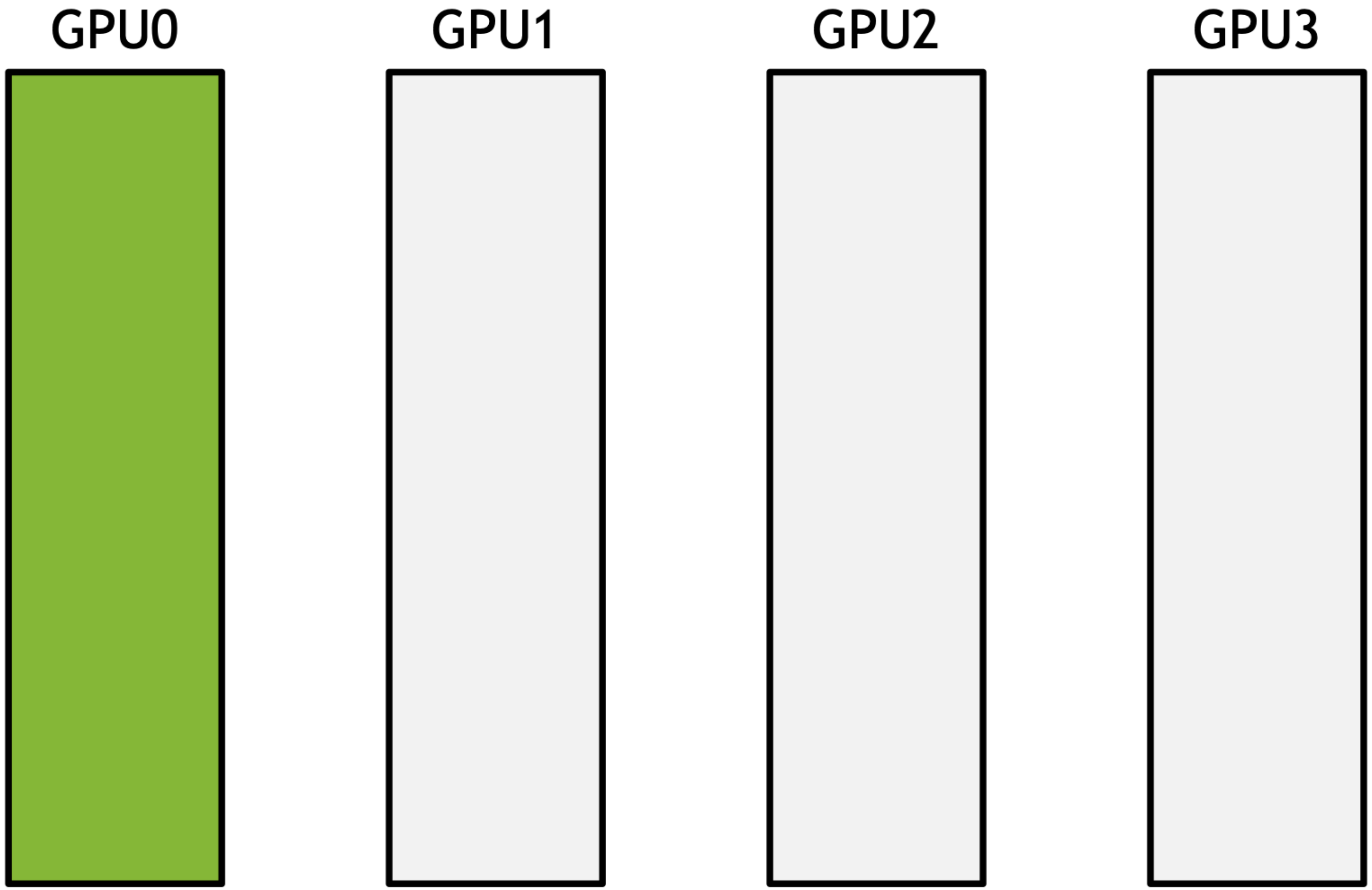
```
ncclGroupStart();

ncclSend(sendbuff, sendcount, sendtype, peer, comm, stream);

ncclRecv(recvbuff, recvcount, recvtype, peer, comm, stream);

ncclGroupEnd();
```

How Reduce is Implemented?

- NCCL uses rings to move data across all GPUs and perform reductions.

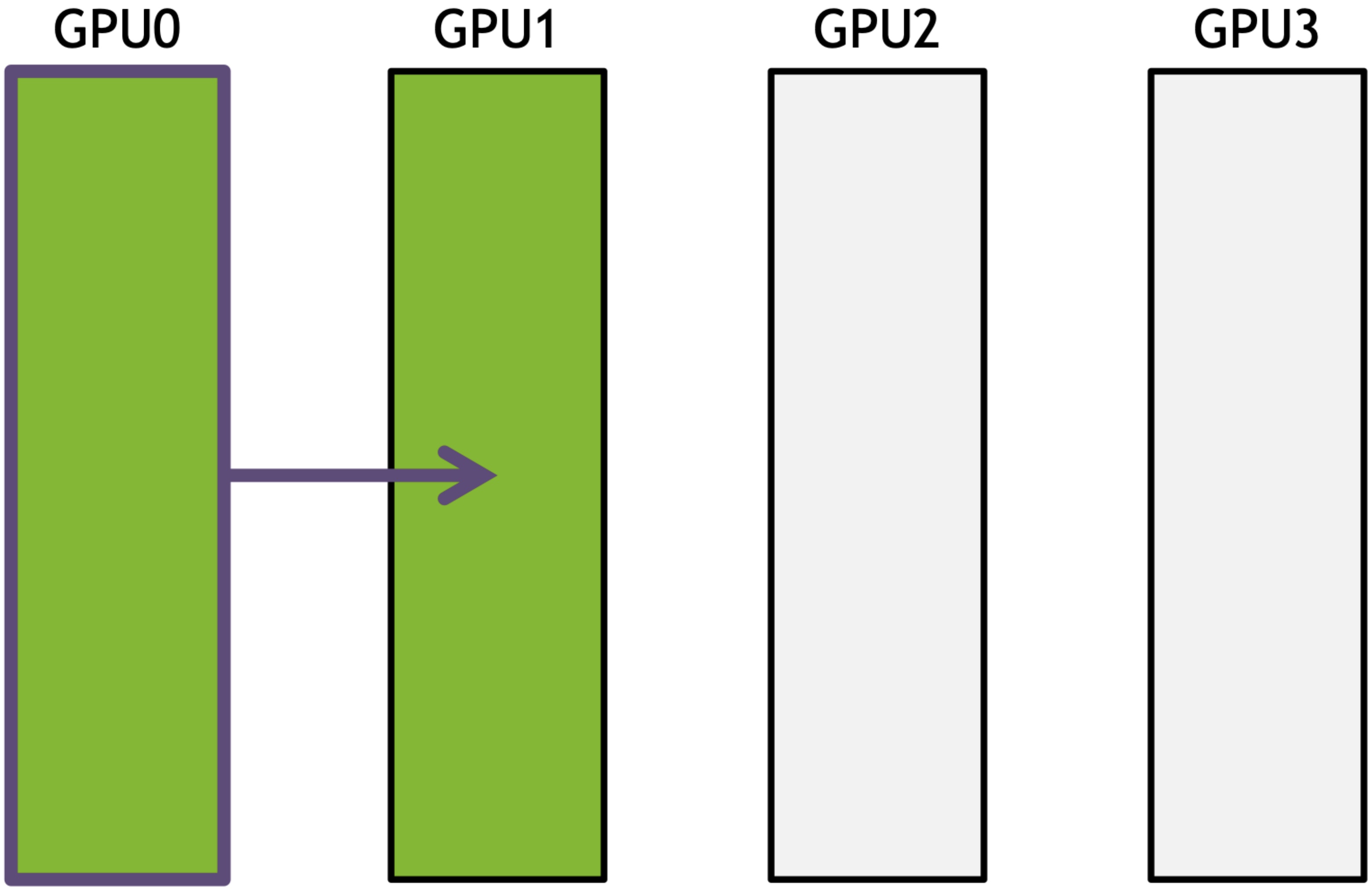


Broadcast with unidirectional ring



N=bytes to transfer
B=bandwidth

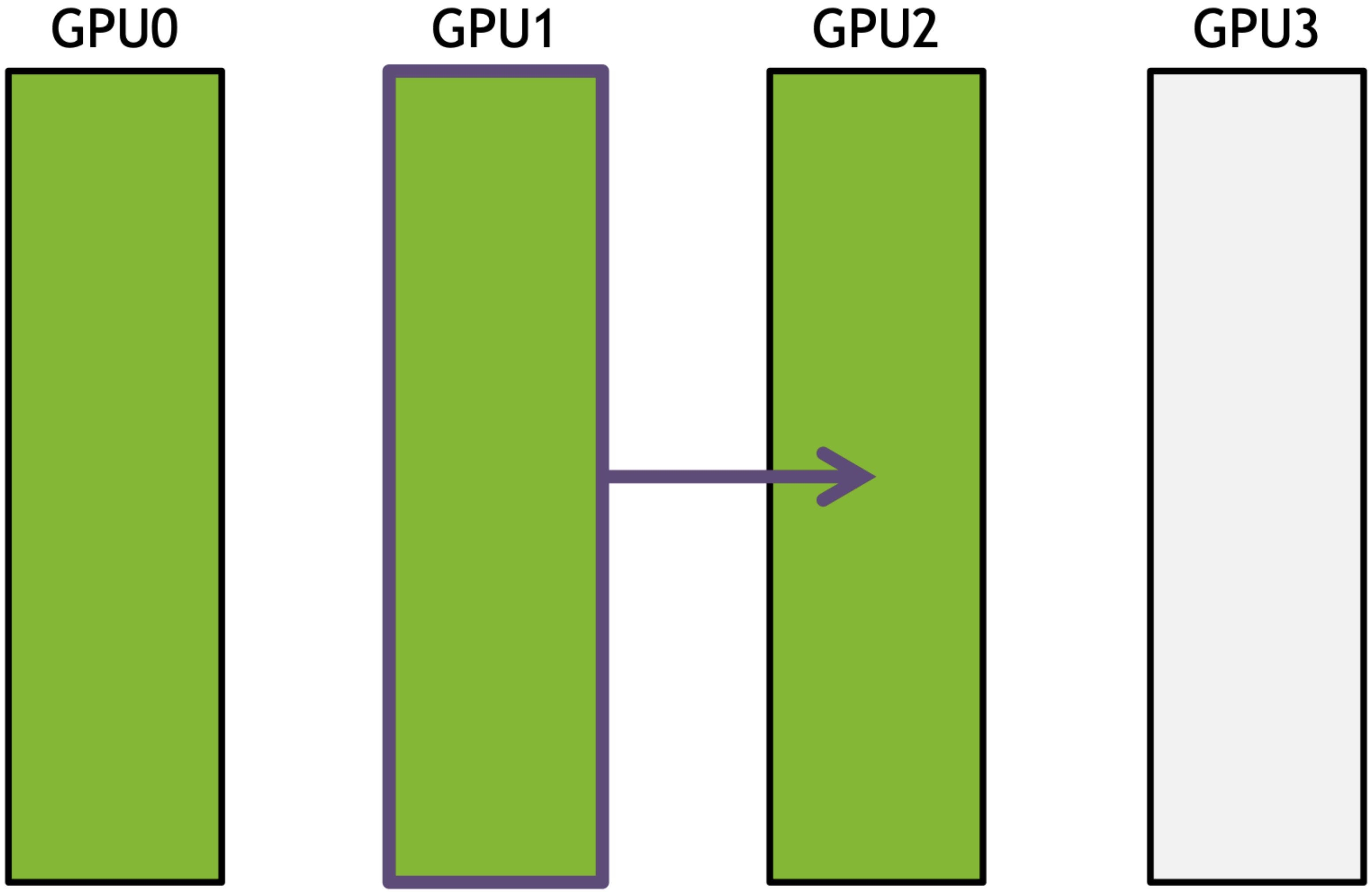
Broadcast with unidirectional ring



Step 1: t = N/B

N=bytes to transfer
B=bandwidth

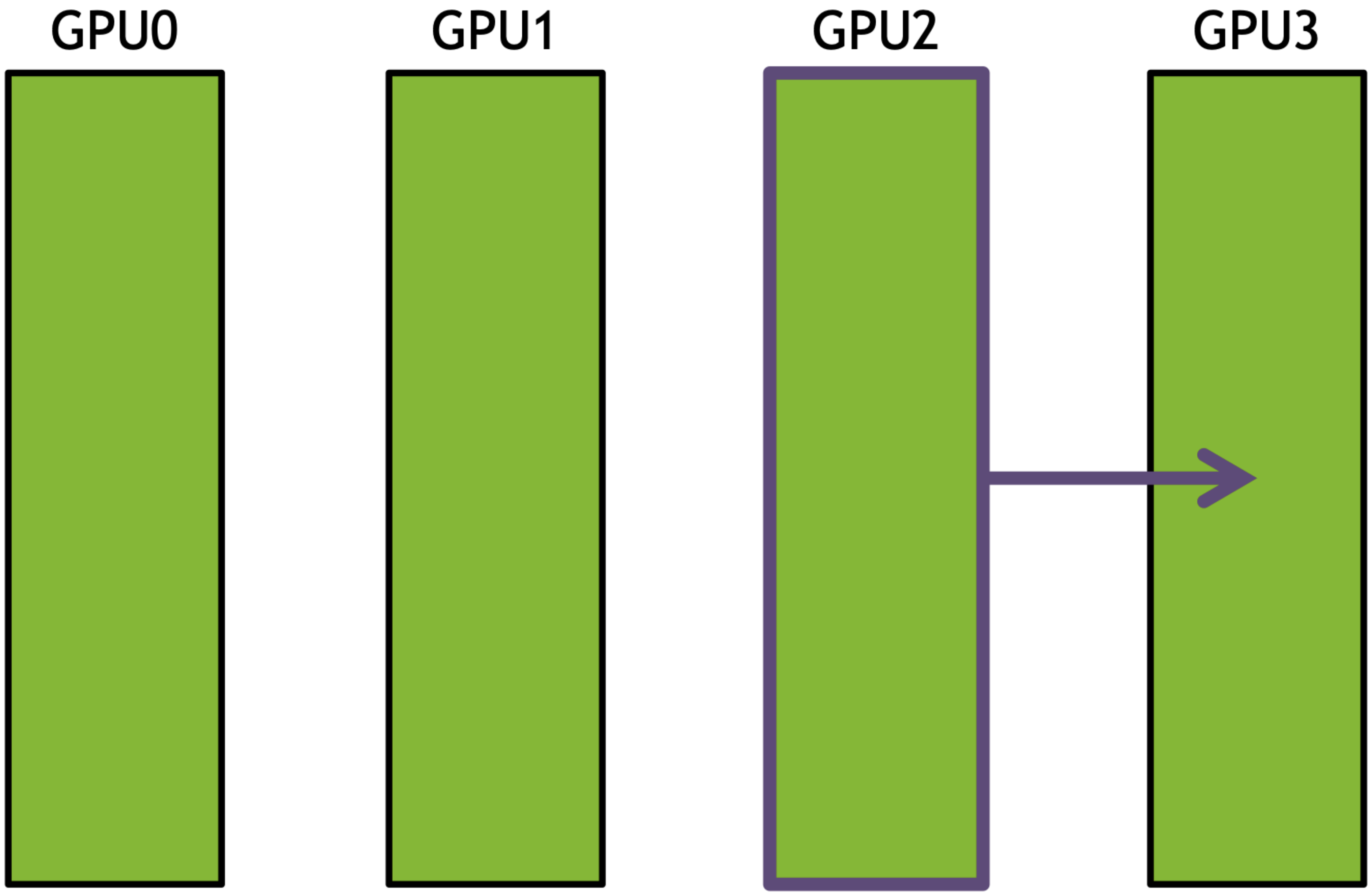
Broadcast with unidirectional ring



Step 1: $t = N/B$
Step 2: $t = N/B$

N=bytes to transfer
B=bandwidth

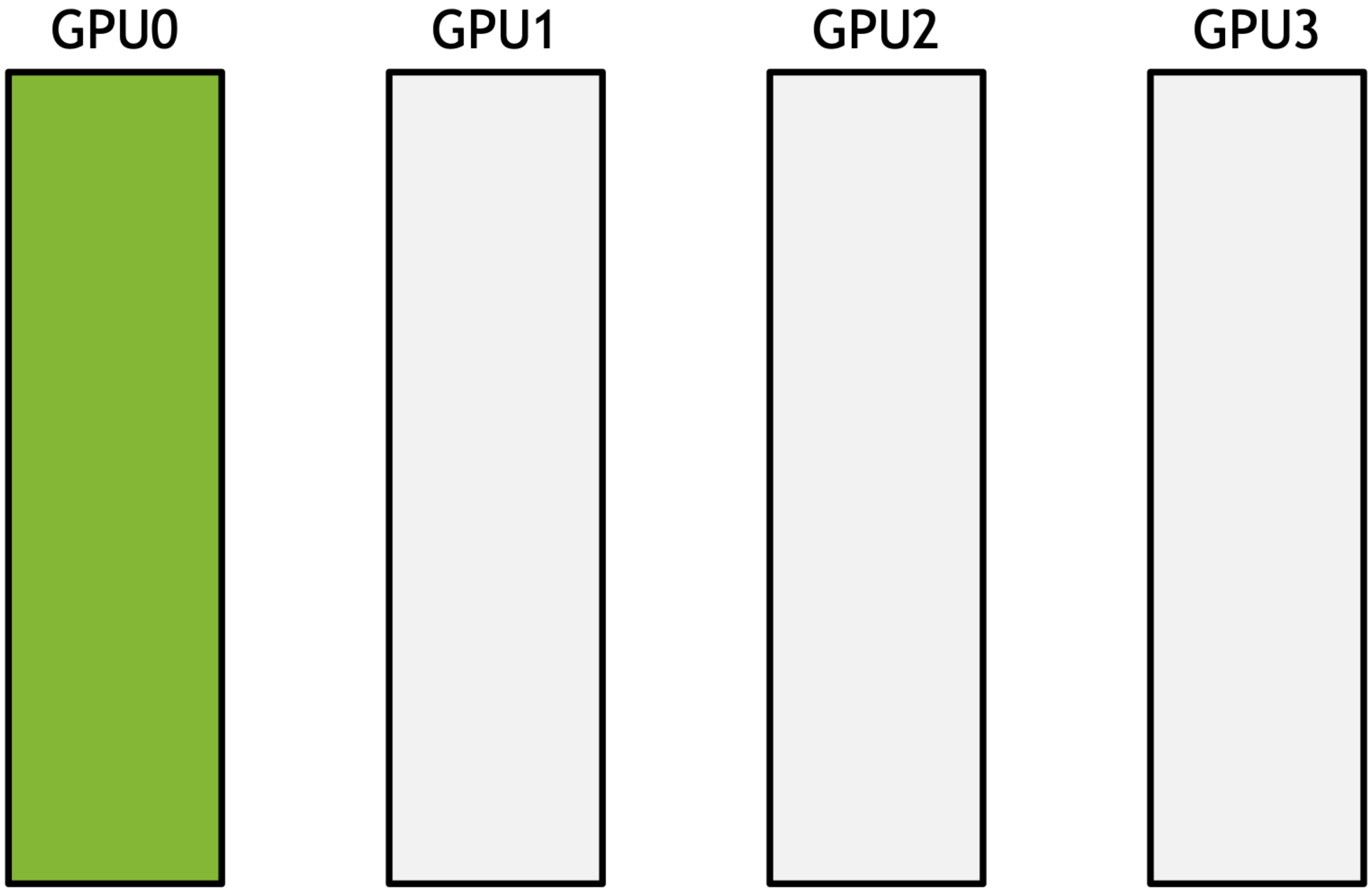
Broadcast with unidirectional ring



Step 1: t = N/B
Step 2: t = N/B
Step 3: t = N/B
total time=(K-1) N/B

N=bytes to transfer
B=bandwidth

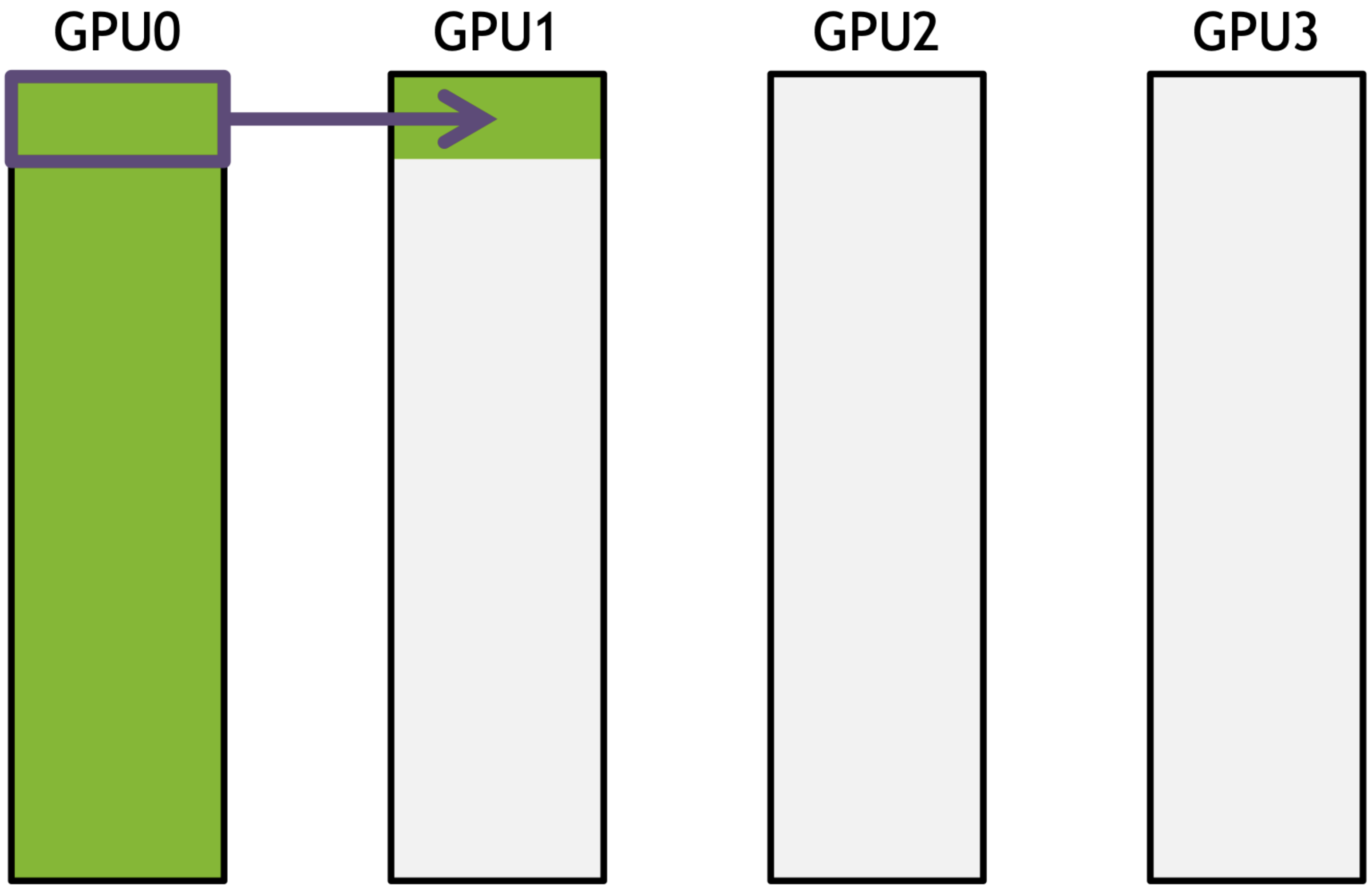
Broadcast with unidirectional ring



N=bytes to transfer
B=bandwidth

Broadcast with unidirectional ring

break data into S messages



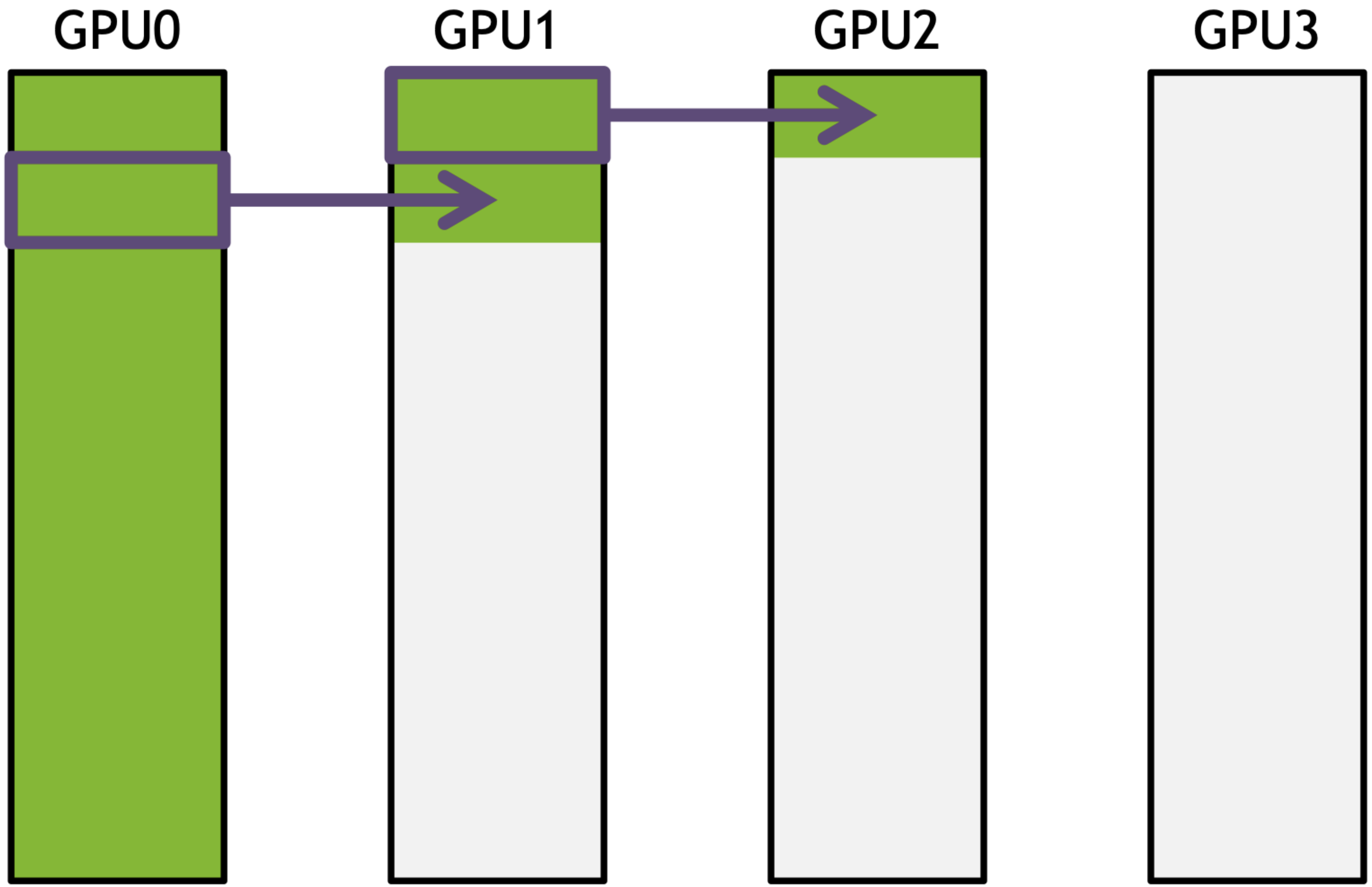
Step 1: $t = N/SB$

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: t = N/SB

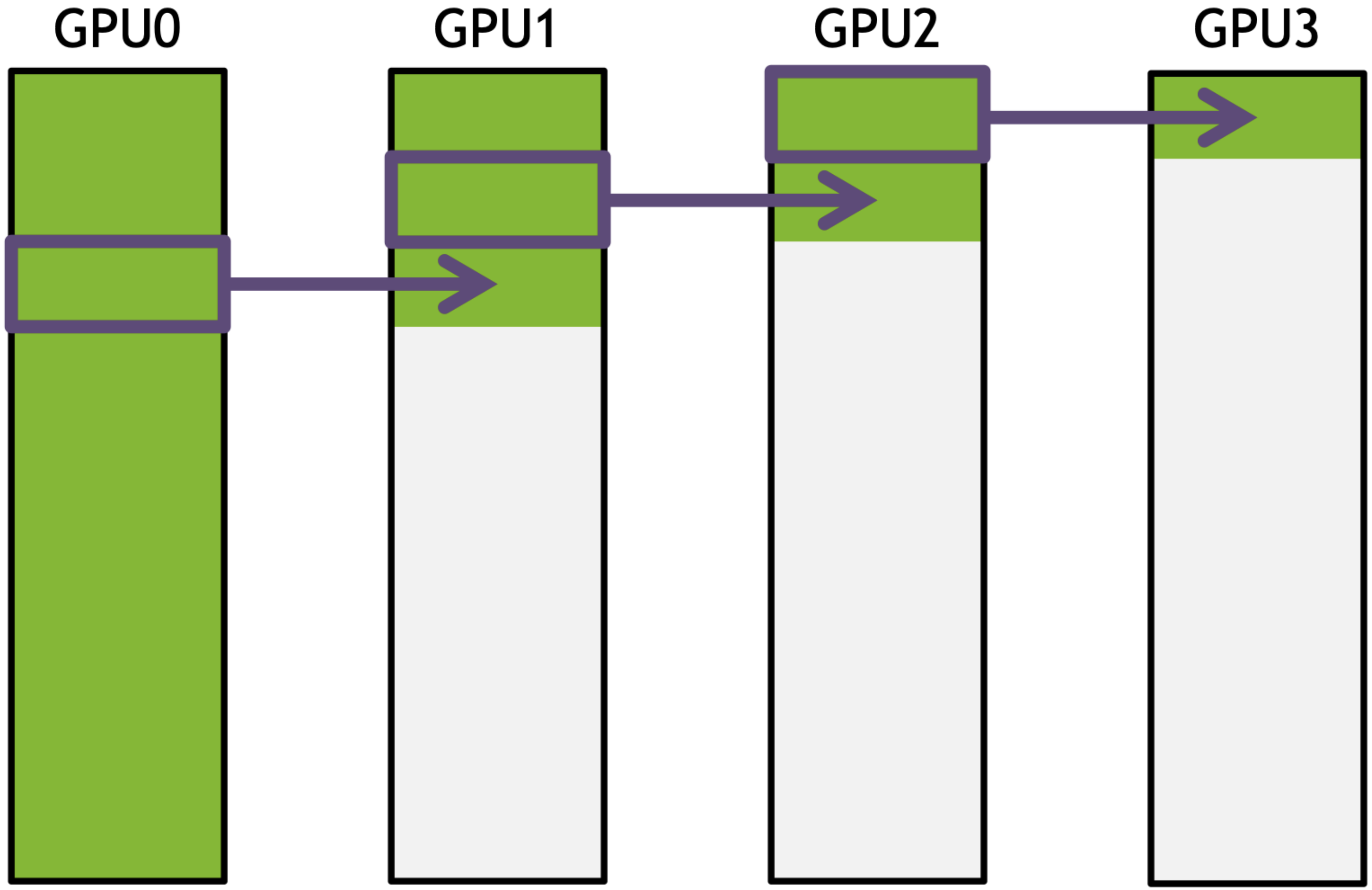
Step 2: t = N/SB

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages

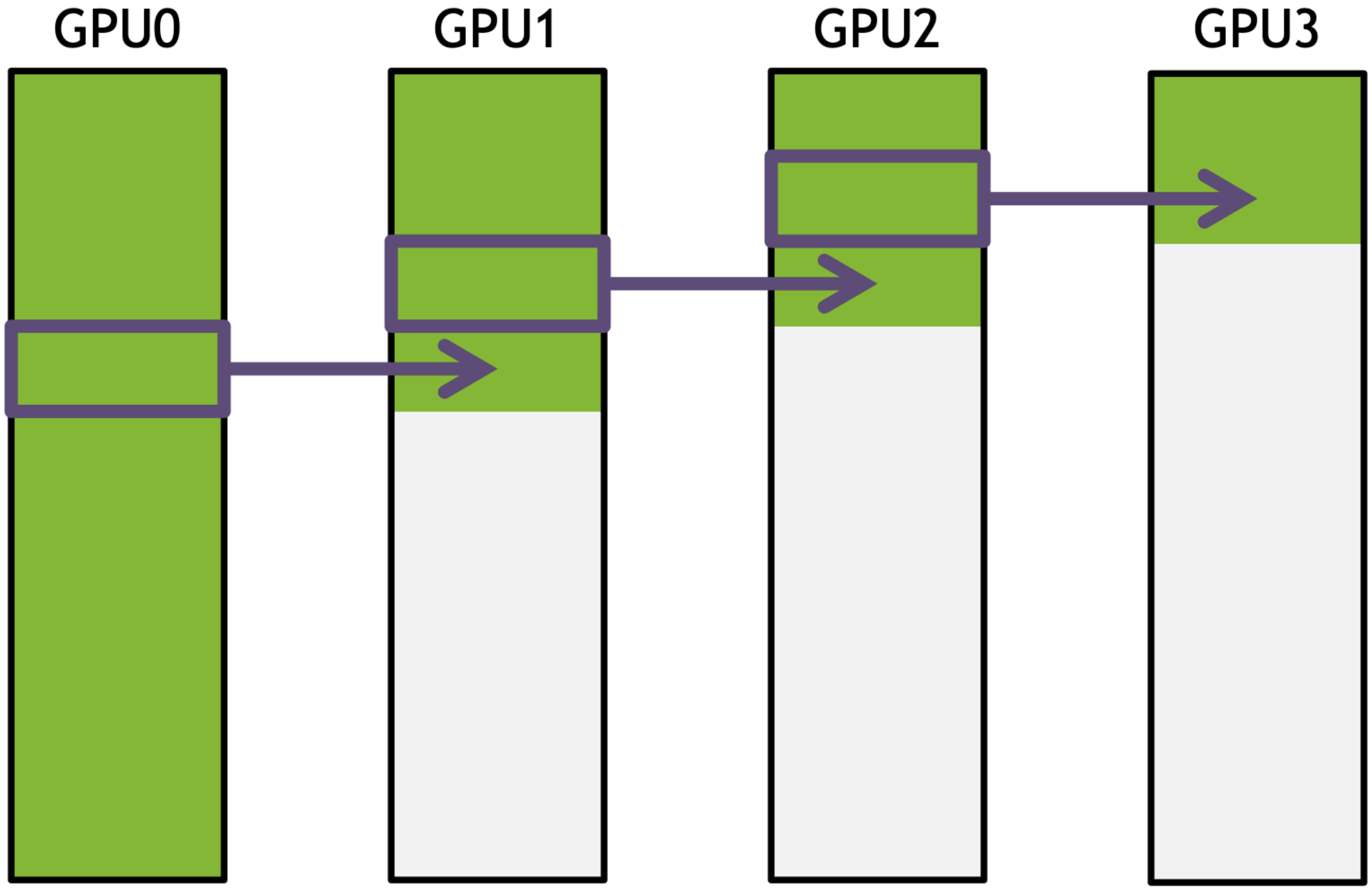


Step 1: t = N/SB
Step 2: t = N/SB
Step 3: t = N/SB

N=bytes to transfer
B=bandwidth

Broadcast with unidirectional ring

break data into S messages

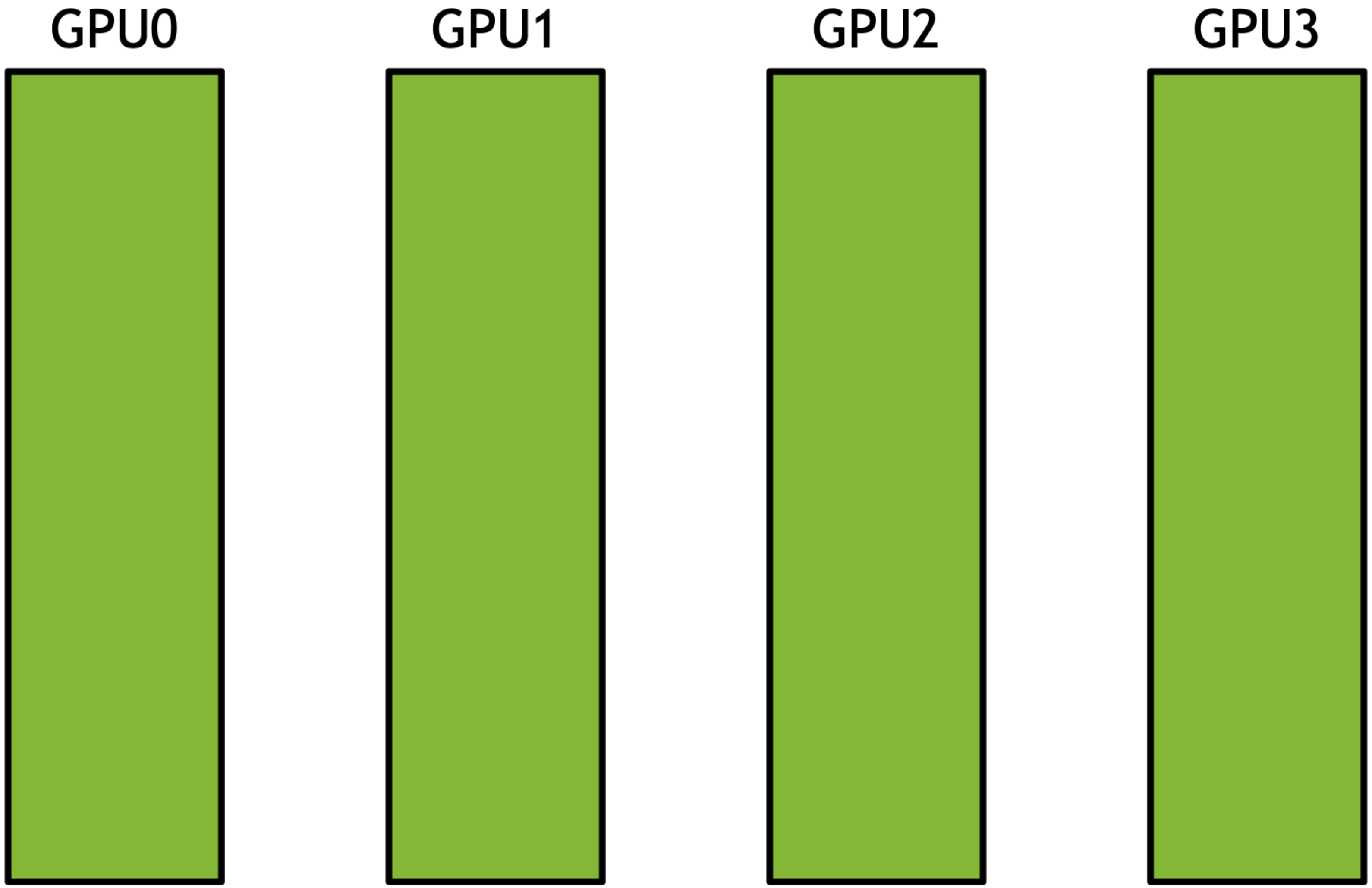


Step 1: t = N/SB
Step 2: t = N/SB
Step 3: t = N/SB
Step 4: t = N/SB

N=bytes to transfer
B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: t = N/SB

Step 2: t = N/SB

Step 3: t = N/SB

Step 4: t = N/SB

…

total time=(K-2+S)N/SB

~=N/B

N=bytes to transfer

B=bandwidth

Example

```
//initializing NCCL, group API is required around ncclCommInitRank as it is
//called across multiple GPUs in each thread/process NCCLCHECK(ncclGroupStart());
for (int i=0; i<nDev; i++) {
  CUDACHECK(cudaSetDevice(localRank*nDev + i));
  NCCLCHECK(ncclCommInitRank(comms+i, nRanks*nDev, id, myRank*nDev + i));
}
NCCLCHECK(ncclGroupEnd());
//calling NCCL communication API. Group API is required when using
//multiple devices per thread/process
NCCLCHECK(ncclGroupStart());
for (int i=0; i<nDev; i++)
  NCCLCHECK(ncclAllReduce((const void*)sendbuff[i], (void*)recvbuff[i], size, ncclFloat, ncclSum, comms[i], s[i]));
NCCLCHECK(ncclGroupEnd());
//synchronizing on CUDA stream to complete NCCL communication
for (int i=0; i<nDev; i++)
  CUDACHECK(cudaStreamSynchronize(s[i]));
```

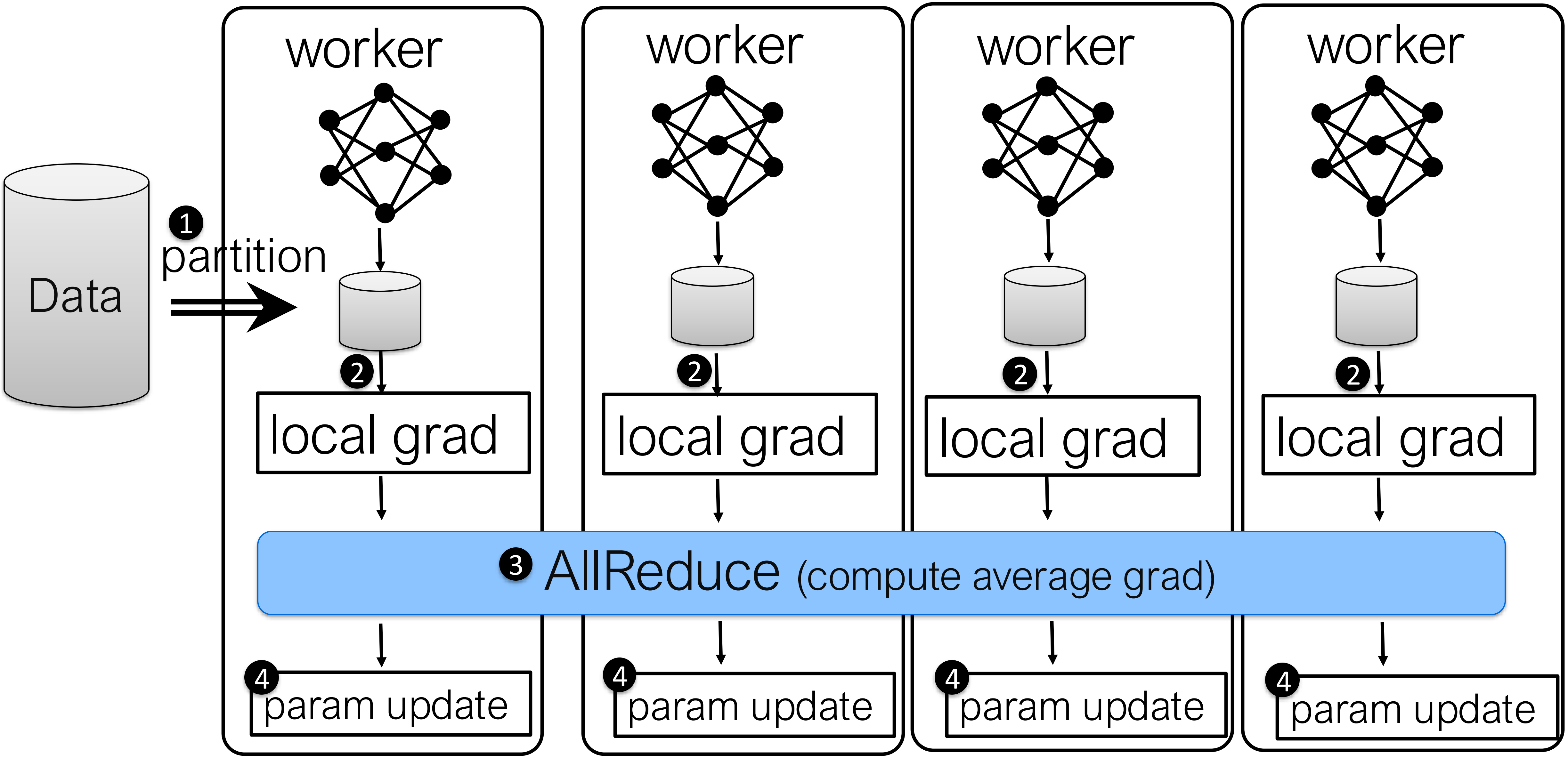
Implementing Parameter Server using NCCL

- Pull: ncclBroadcast
  - Parameter server to send parameters to all workers
- Push: ncclReduce
  - workers send grads to server and sum
- Synchronization:
  - workers need to wait for server to send param
  - server need to wait for work to send grad * N

Outline

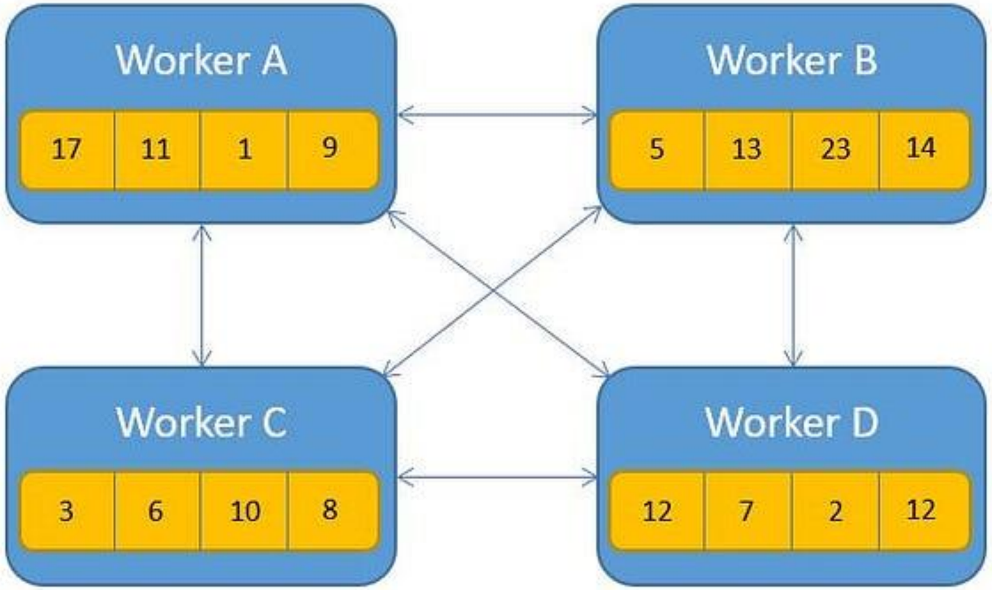
- Overview of large-scale model training
- Multi-GPU communication
- Data Parallel Training via AllReduce

Data Parallel Training

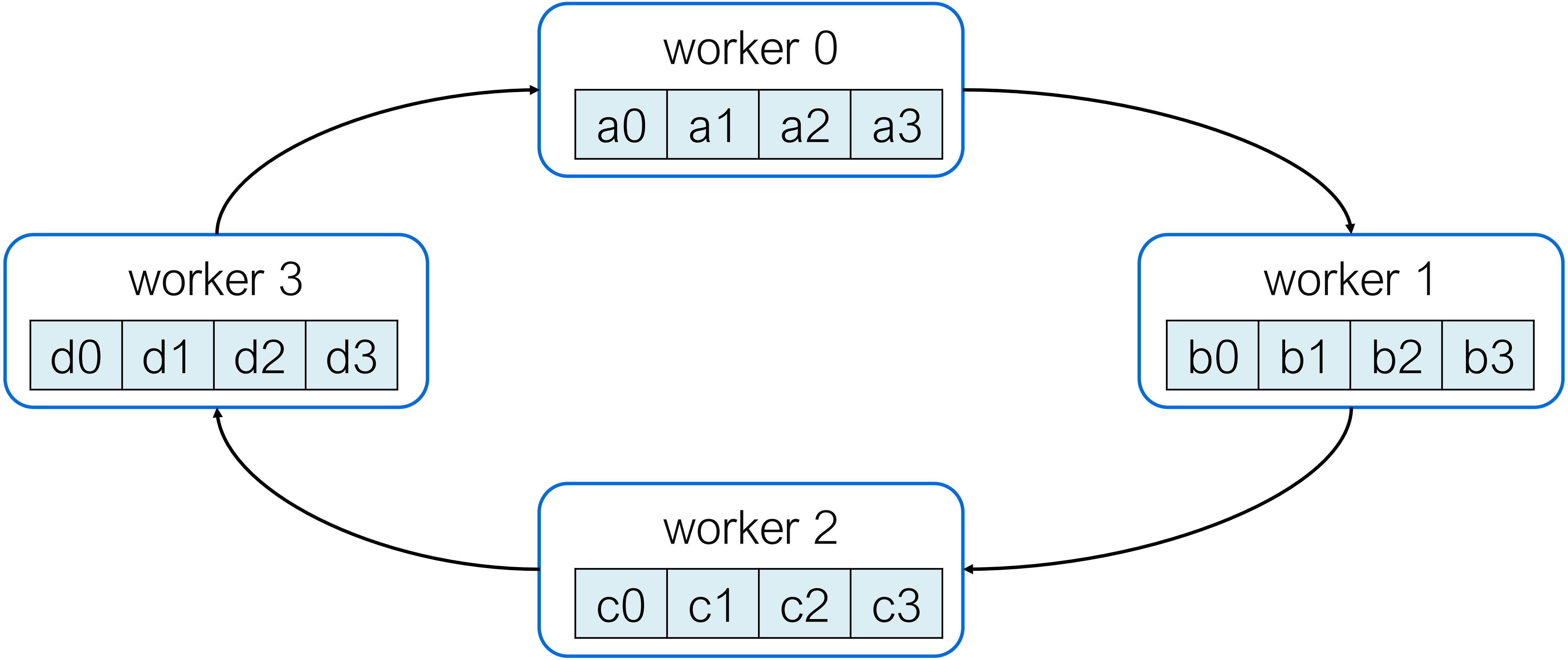


How to Synchronize Gradients?

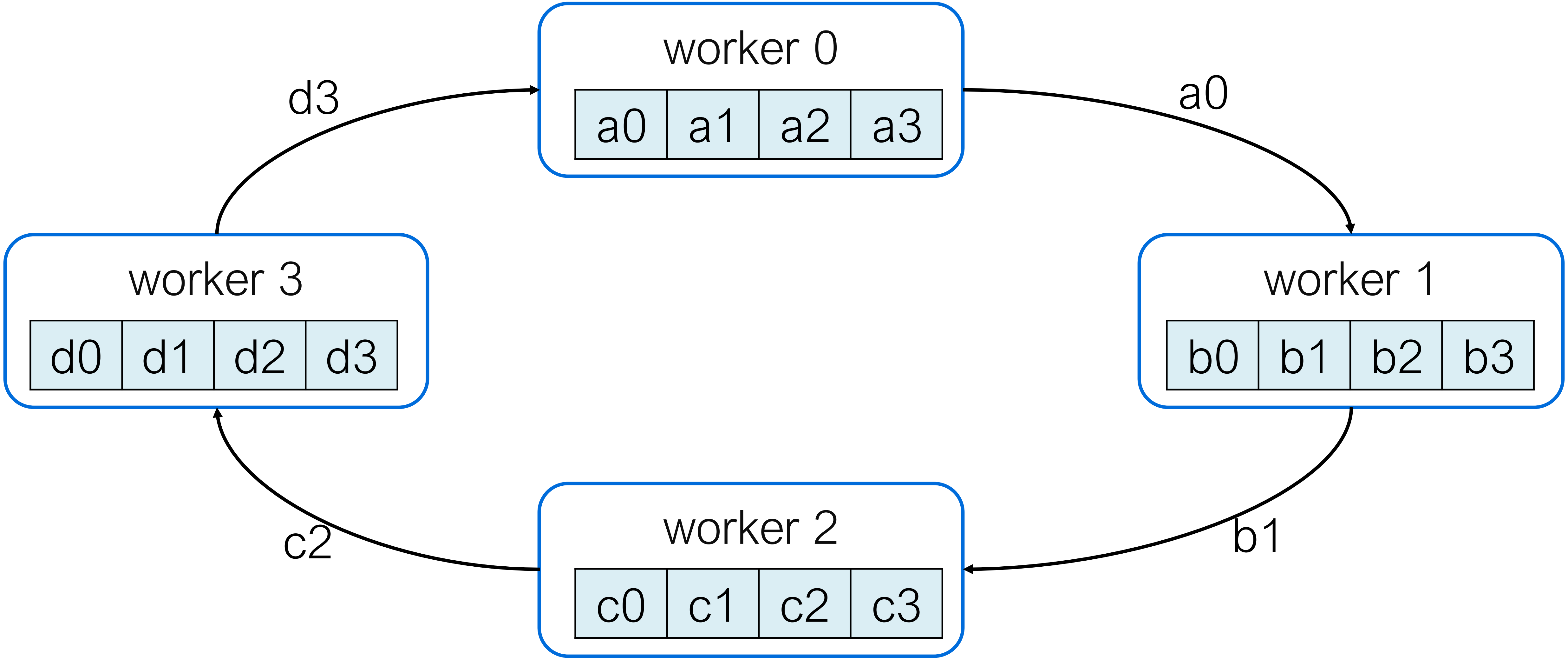
- Naïve all-reduce



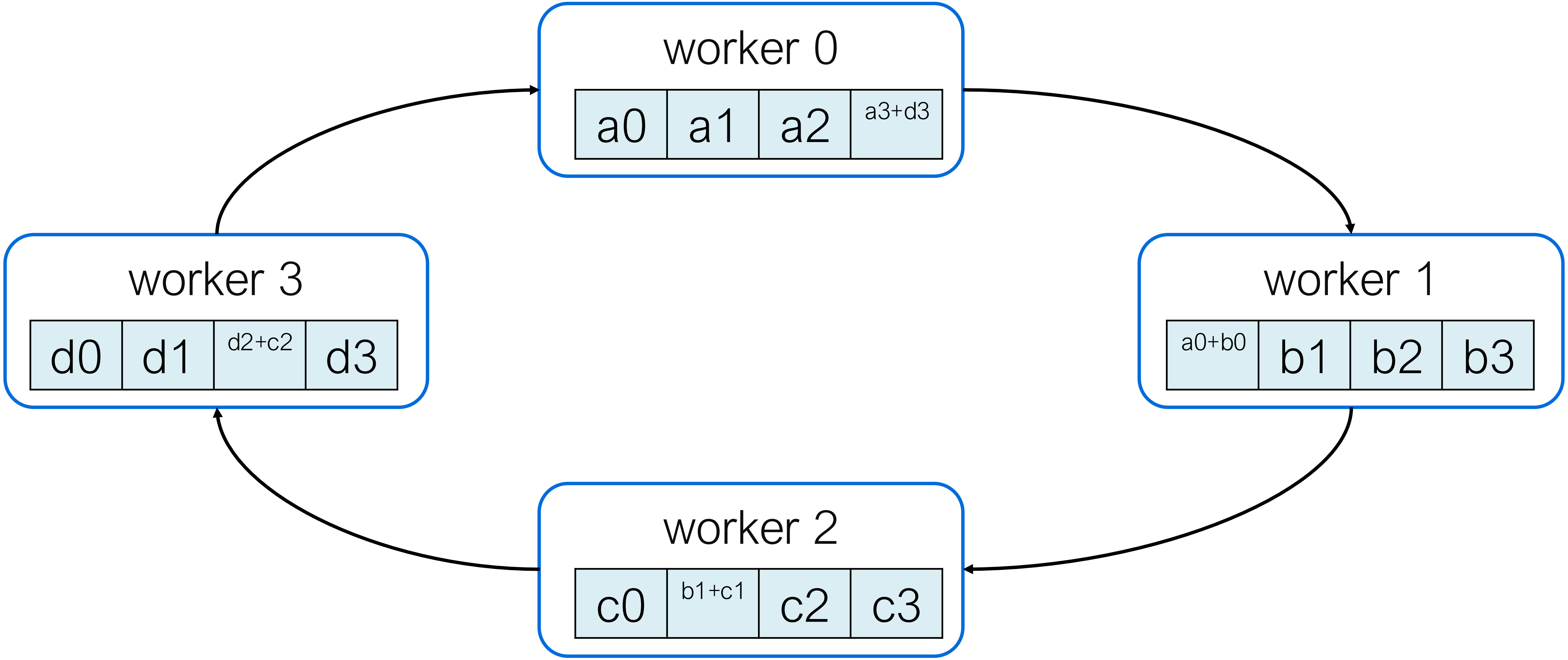
Ring AllReduce



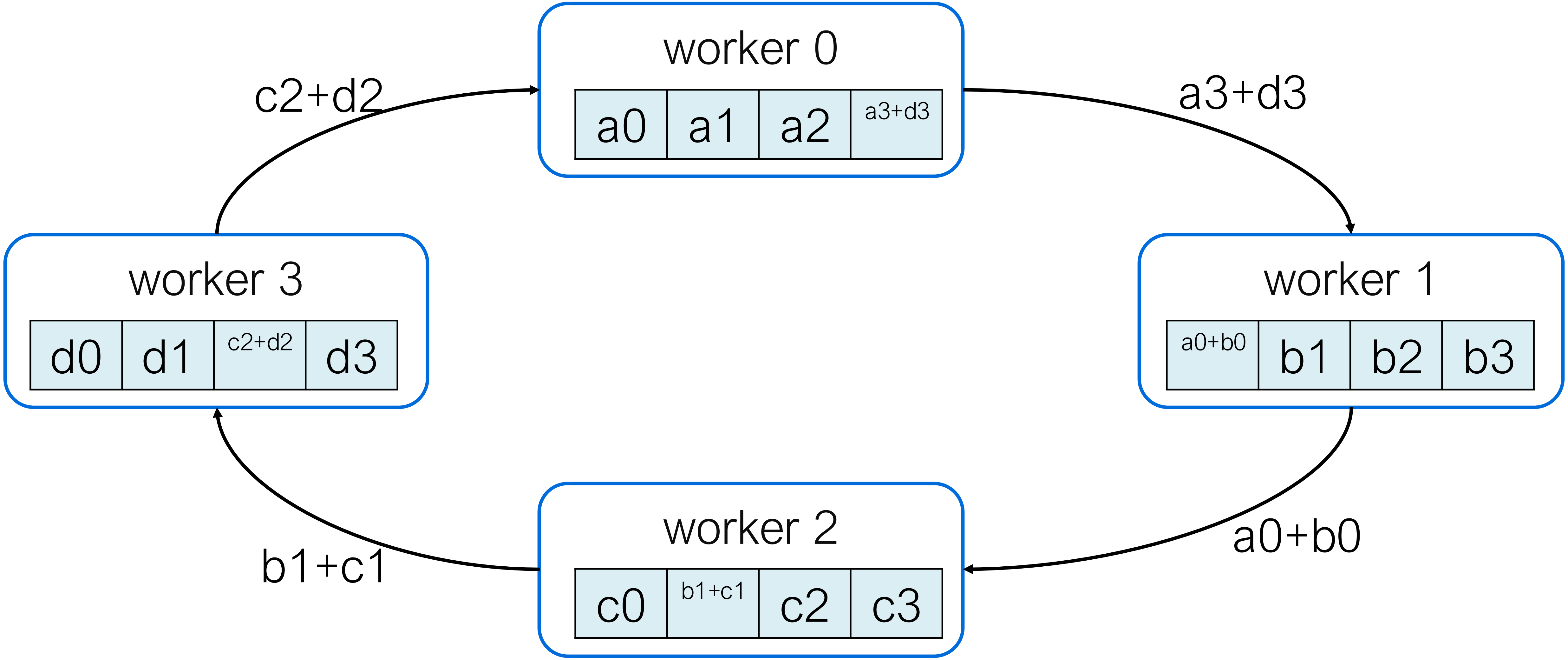
Ring AllReduce



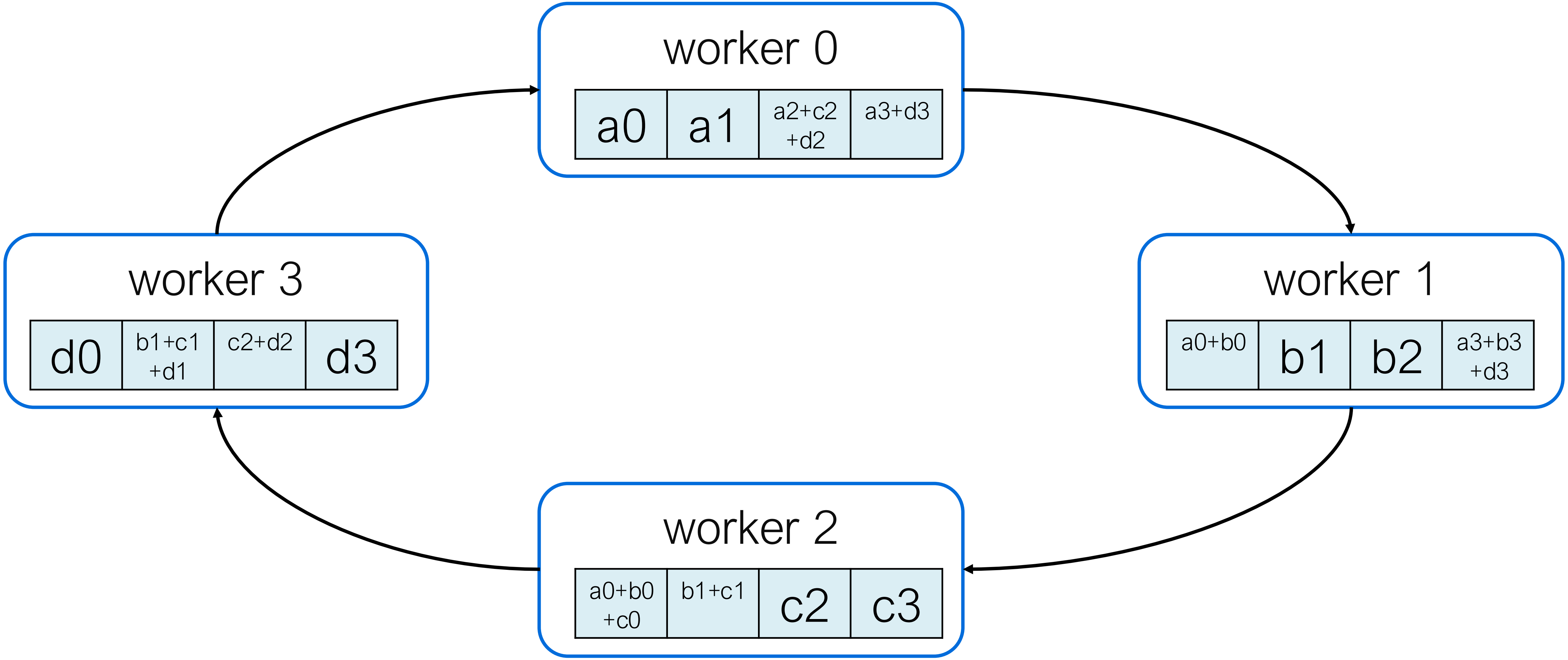
Ring AllReduce



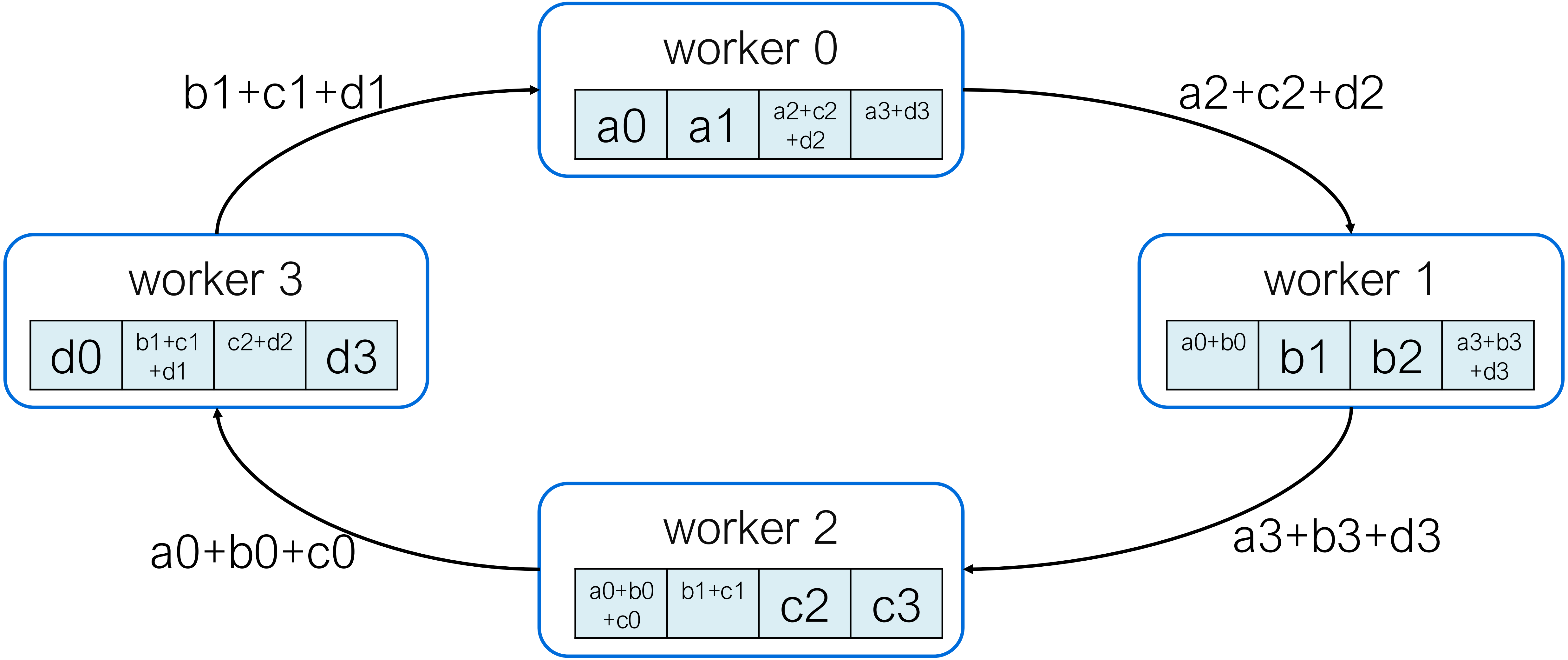
Ring AllReduce



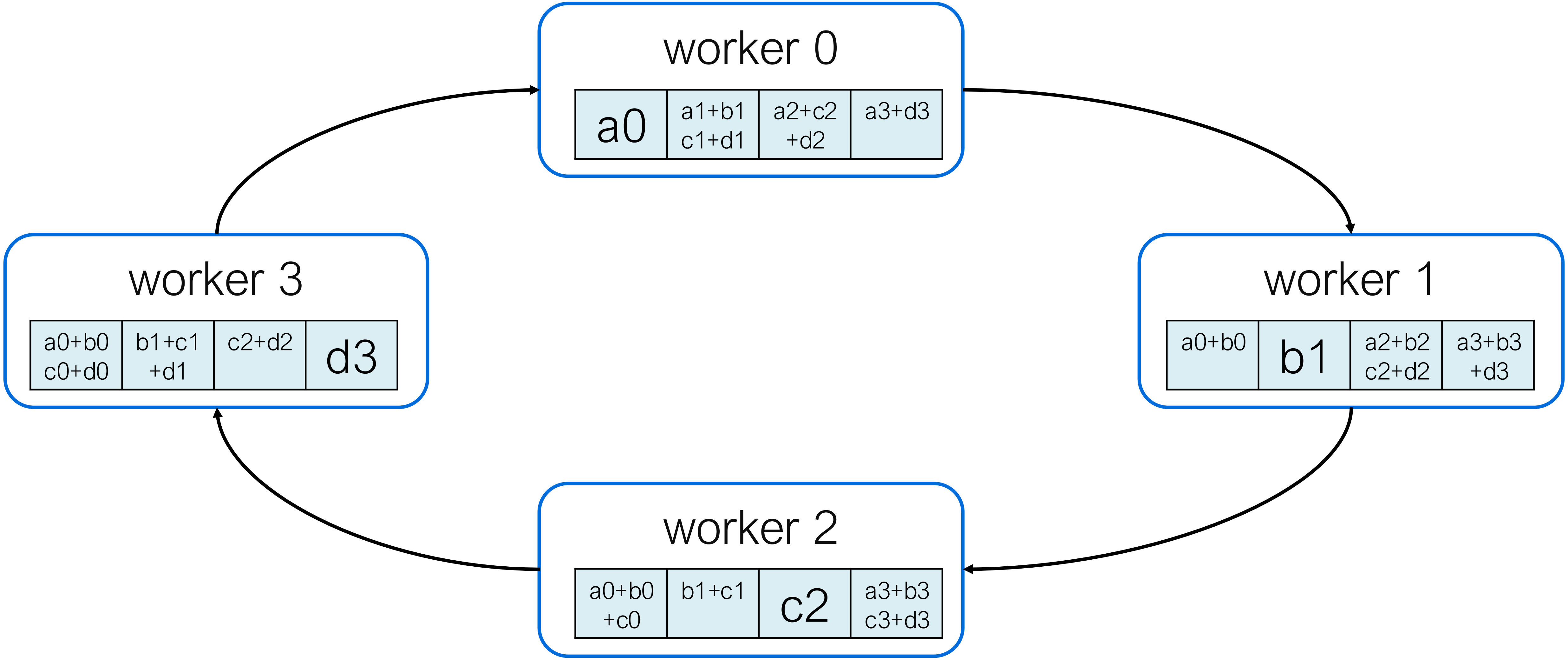
Ring AllReduce



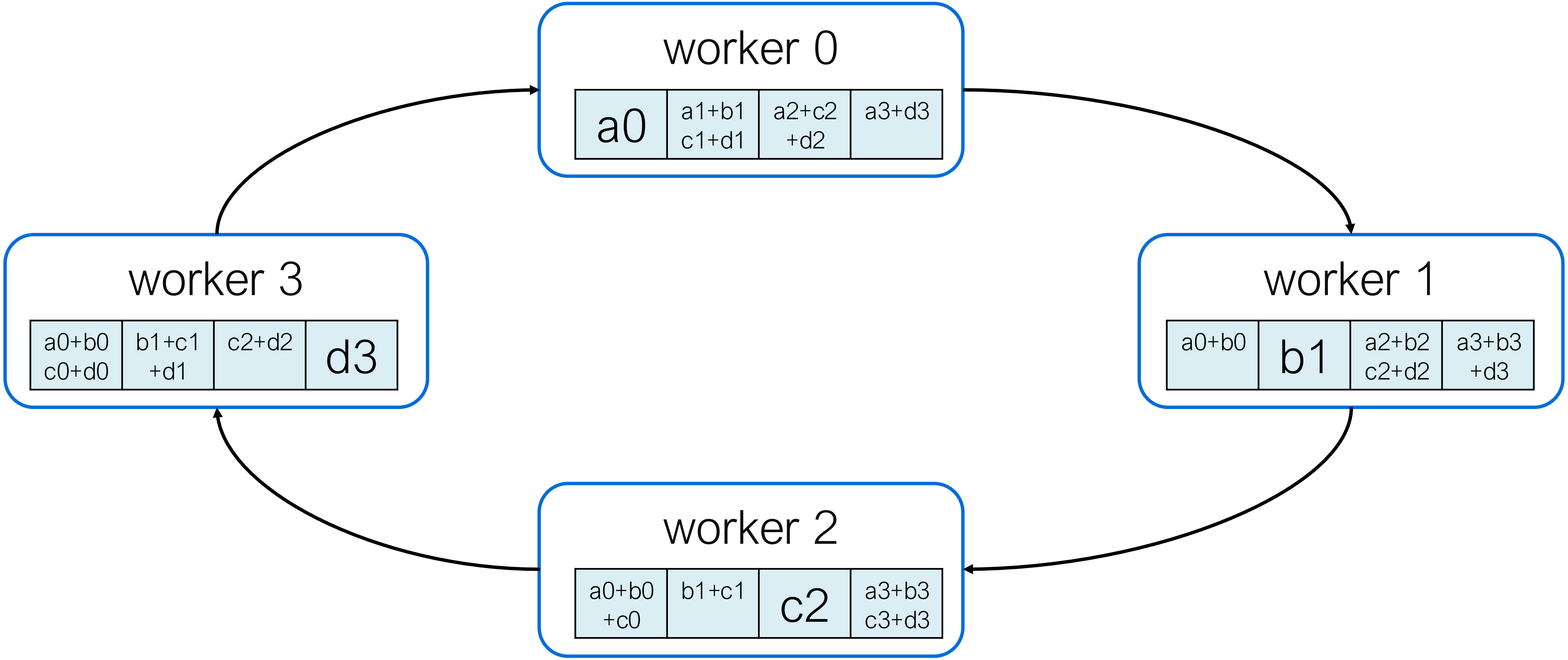
Ring AllReduce



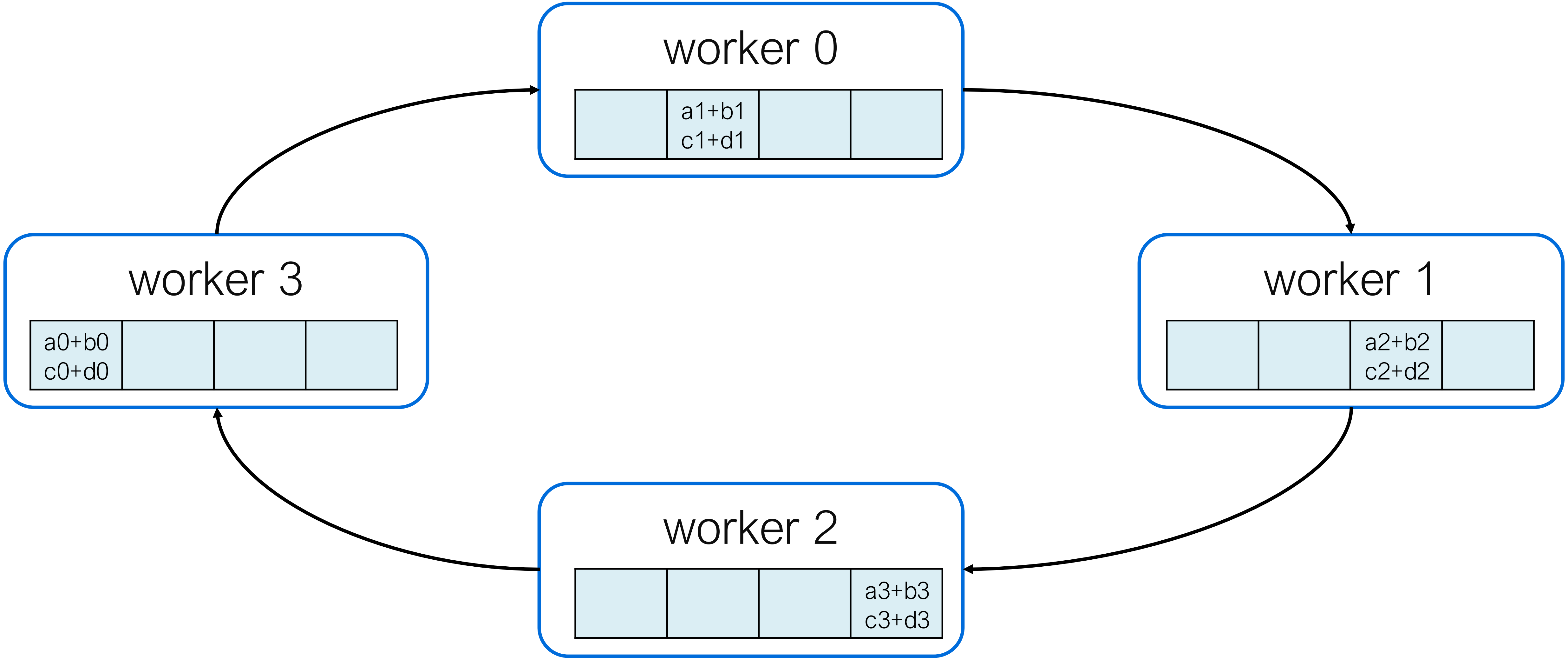
Ring AllReduce



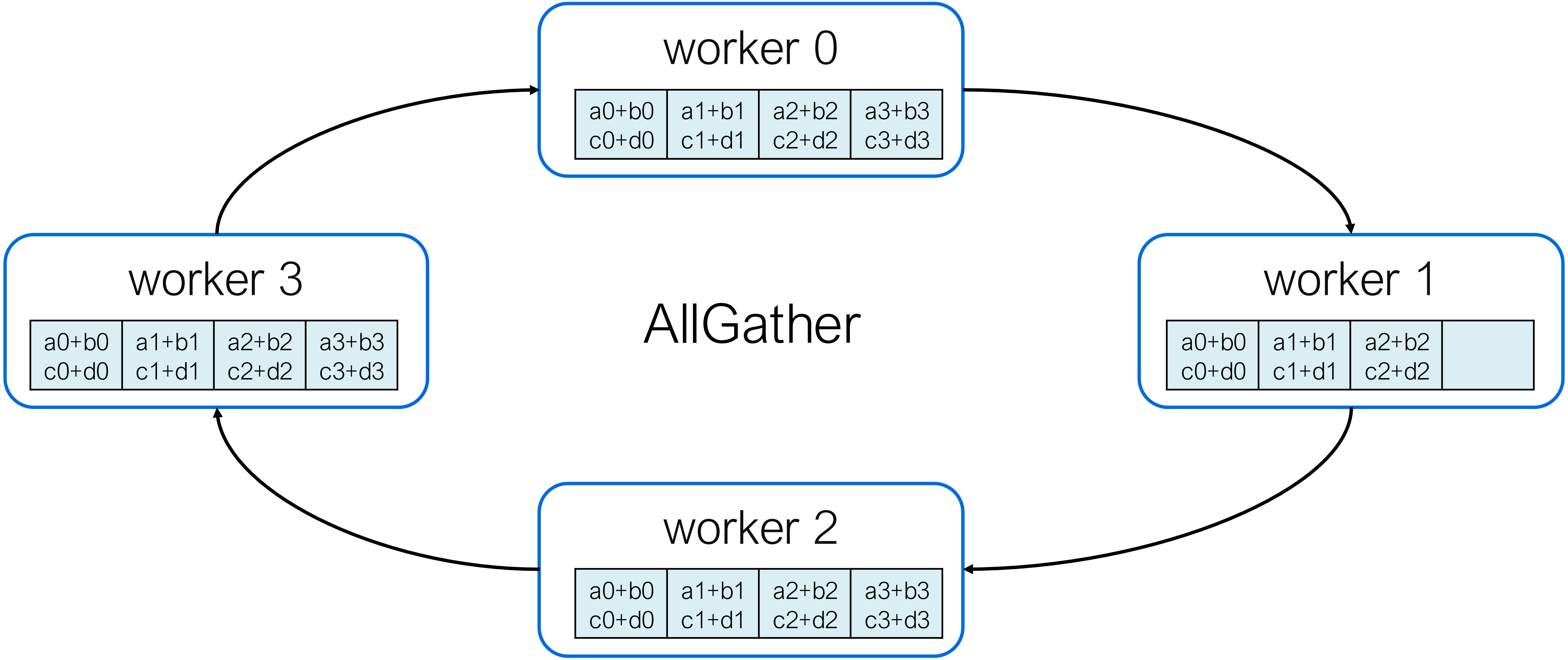
Ring AllReduce



Ring AllReduce



Ring AllReduce



Ring all-reduce: Implementing Scatter-reduce

```
for (int i = 0; i < size - 1; i++) {
        int recv_chunk = (rank - i - 1 + size) % size;
        int send_chunk = (rank - i + size) % size;
        float* segment_send = &(output[segment_ends[send_chunk] -
                segment_sizes[send_chunk]]);

        MPI_Irecv(buffer, segment_sizes[recv_chunk],
                datatype, recv_from, 0, MPI_COMM_WORLD, &recv_req);

        MPI_Send(segment_send, segment_sizes[send_chunk],
                MPI_FLOAT, send_to, 0, MPI_COMM_WORLD);

        float *segment_update = &(output[segment_ends[recv_chunk] -
                        segment_sizes[recv_chunk]]);

        // Wait for recv to complete before reduction
        MPI_Wait(&recv_req, &recv_status);

        reduce(segment_update, buffer, segment_sizes[recv_chunk]);
}
```

Ring all-reduce: Implementing All-gather

```
for (size_t i = 0; i < size_t(size - 1); ++i) {
        int send_chunk = (rank - i + 1 + size) % size;
        int recv_chunk = (rank - i + size) % size;

        // Segment to send - at every iteration we send segment (r+1-i)
        float* segment_send = &(output[segment_ends[send_chunk] -
                segment_sizes[send_chunk]]);

        // Segment to recv - at every iteration we receive segment (r-i)
        float* segment_recv = &(output[segment_ends[recv_chunk] -
                segment_sizes[recv_chunk]]);
        MPI_Sendrecv(segment_send, segment_sizes[send_chunk],
                datatype, send_to, 0, segment_recv,
                segment_sizes[recv_chunk], datatype, recv_from,
                0, MPI_COMM_WORLD, &recv_status);
}
```

Parameter Server vs AllReduce Data Parallel

- Parameter Server
  - needs to synchronize twice (parameters and local gradients)

- AllReduce Data Parallel
  - each local worker needs to update parameters
  - redundant?
  - local updating is much faster than transferring data across gpus

Summary

- Overall idea: partition the data, distribute the forward/backward
- Parameter Server
  - server to update and distribute parameters, worker to get local grad
- NCCL Multi-GPU communication
  - using ring and batching to reduce the latency for Broadcast
- Data Parallel Training via All Reduce
  - Efficient Ring AllReduce (ScatterReduce+AllGather)

Reading for next lecture

- PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.