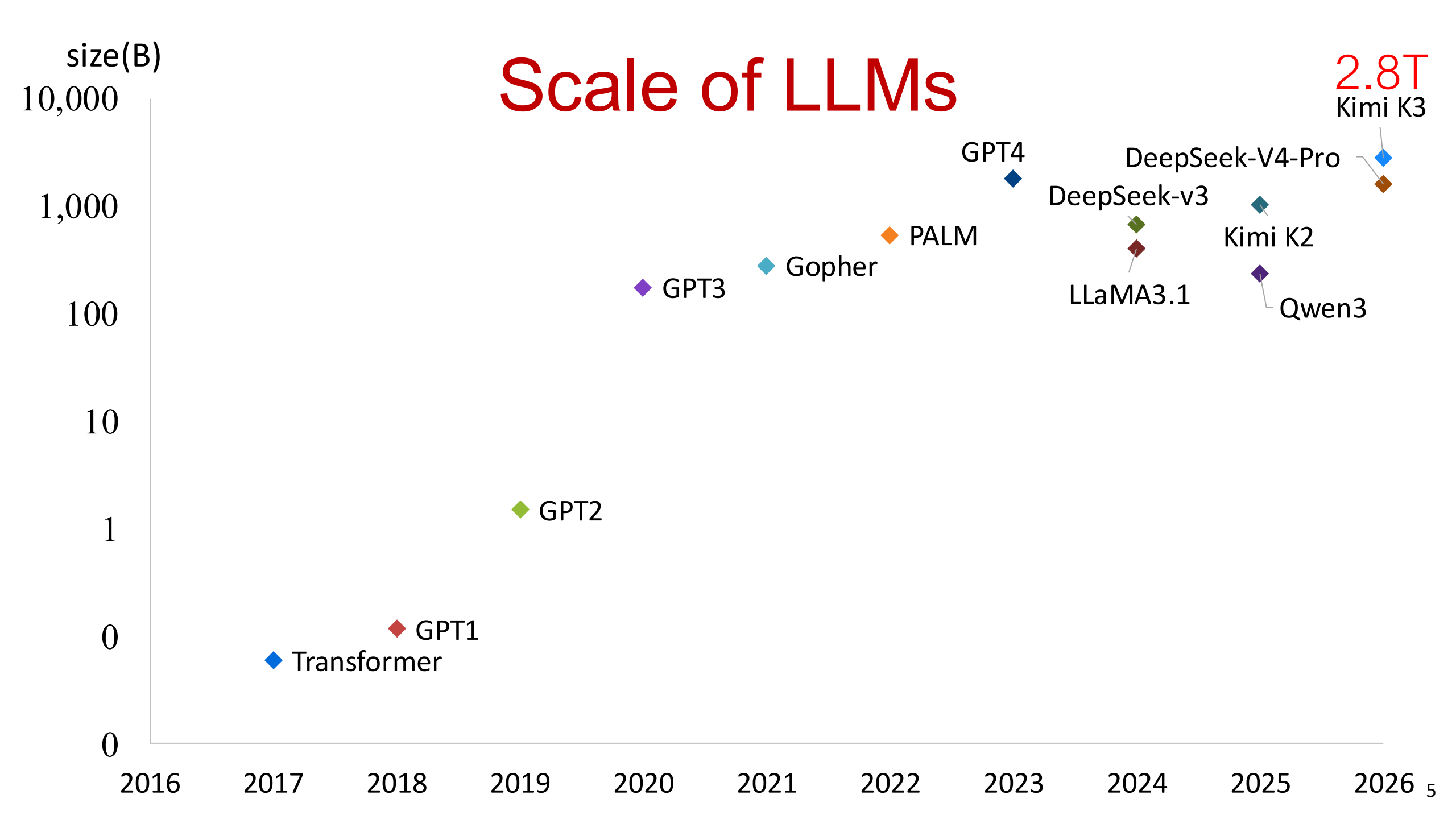
Distributed Data Parallel Training

Lei Li

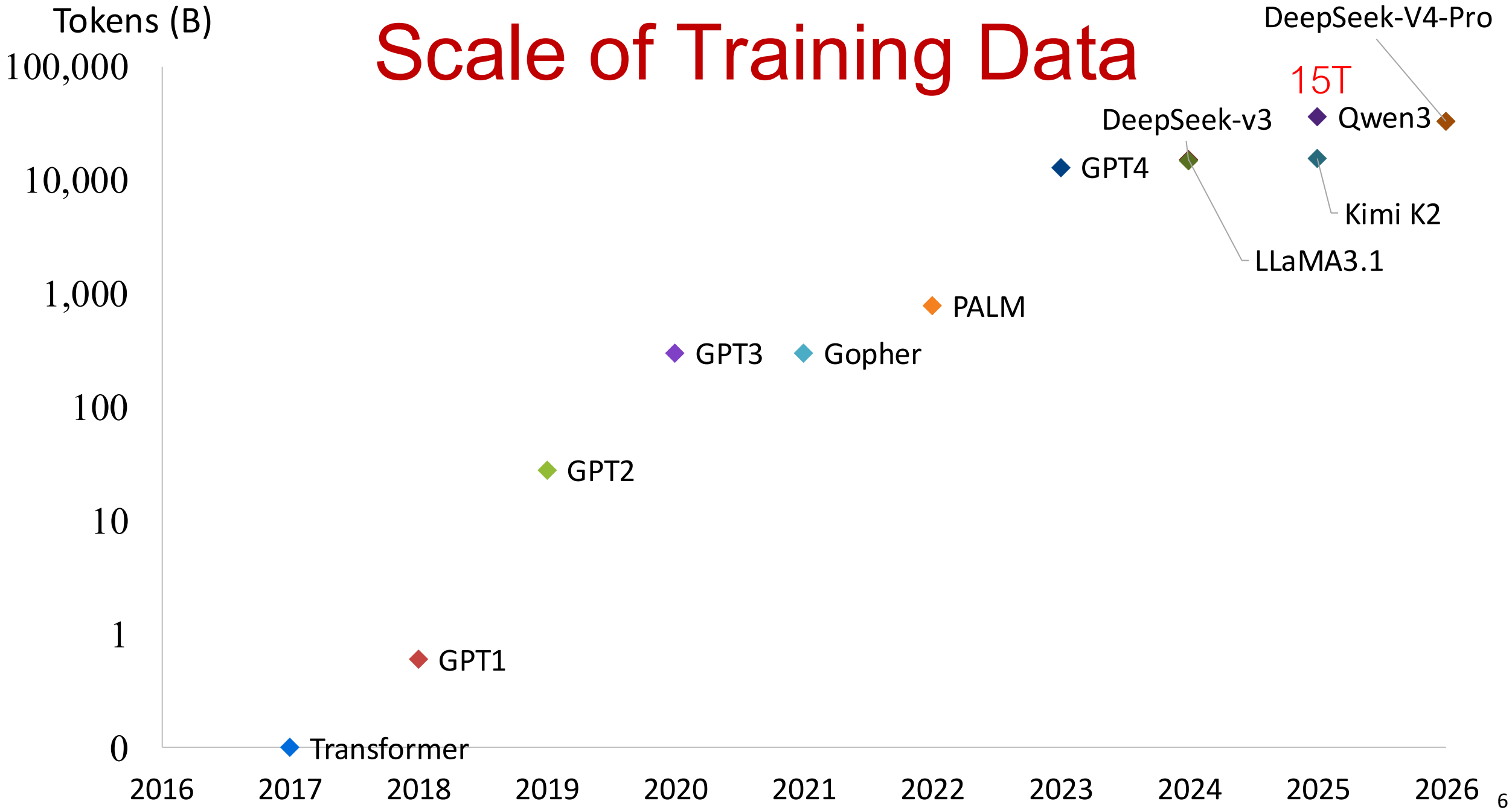
Language Technologies Institute | Carnegie Mellon University School of Computer Science

Outline

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


size(B)
Scale of LLMs
2.8T
Kimi K3
10,000
1,000
100
10
1
0
0
GPT4
DeepSeek-V4-Pro
DeepSeek-v3
PALM
Kimi K2
Gopher
GPT3
LLaMA3.1
Qwen3
GPT2
GPT1
Transformer
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026

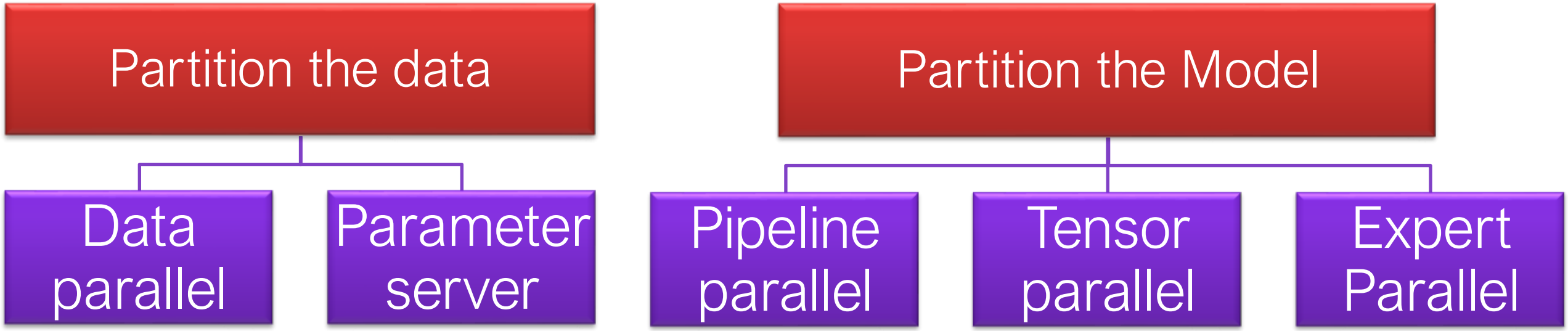
Scale of Training Data



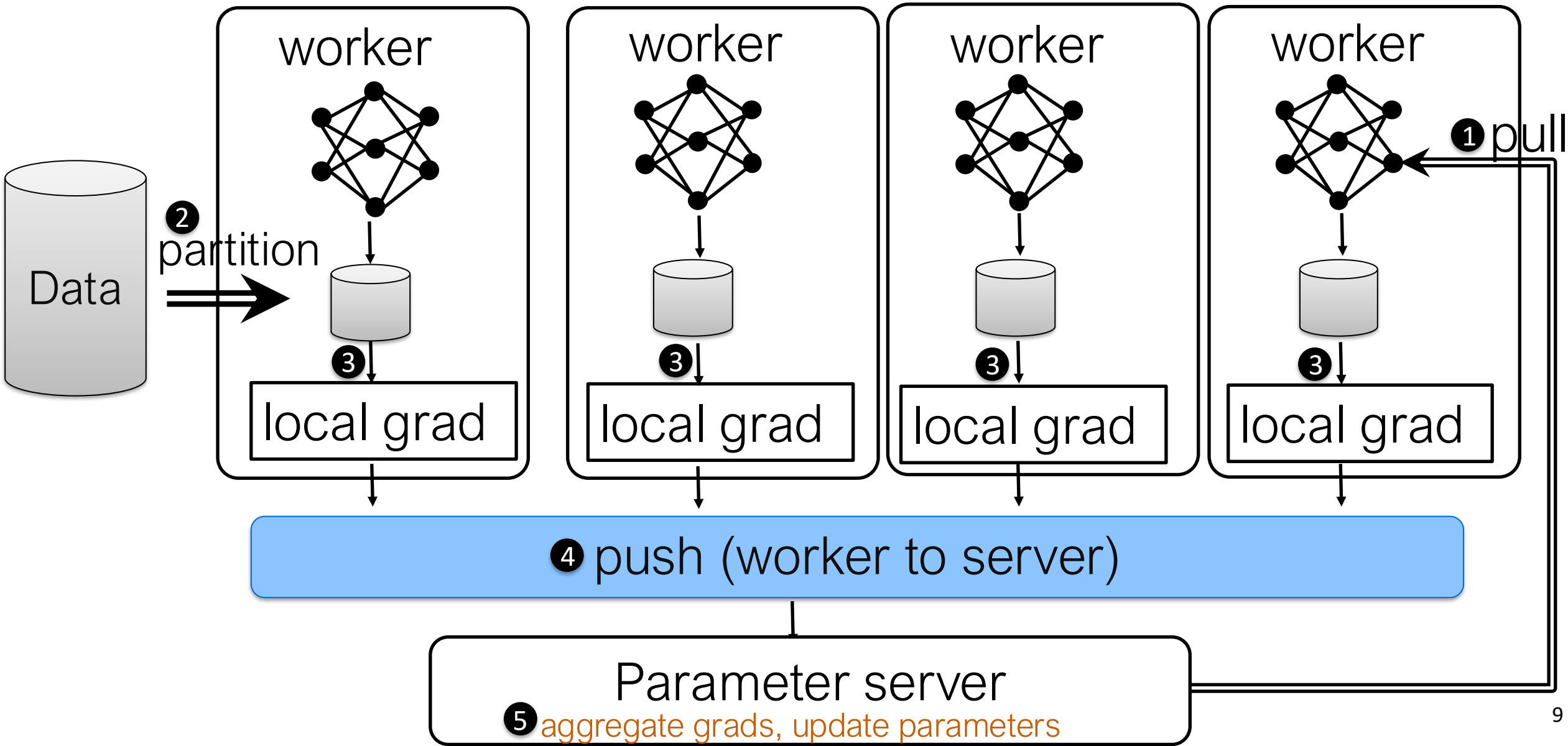
Large-scale Distribution Training

- 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



Classical Distributed Training: Parameter Server



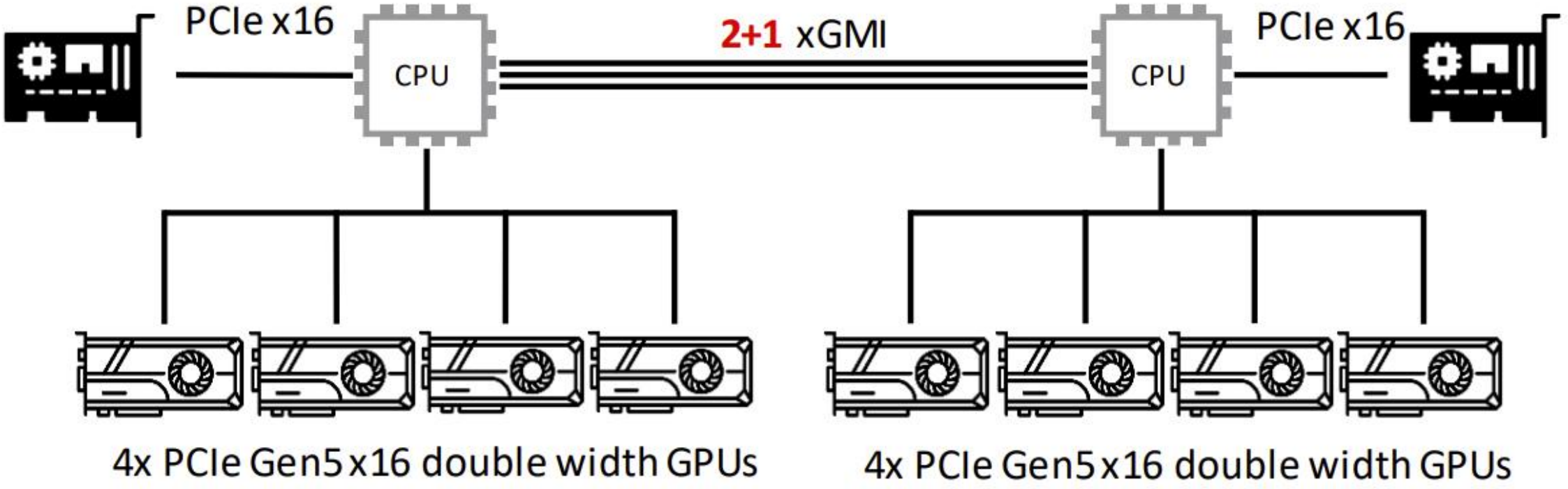
Outline

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

Multi-GPU Communication

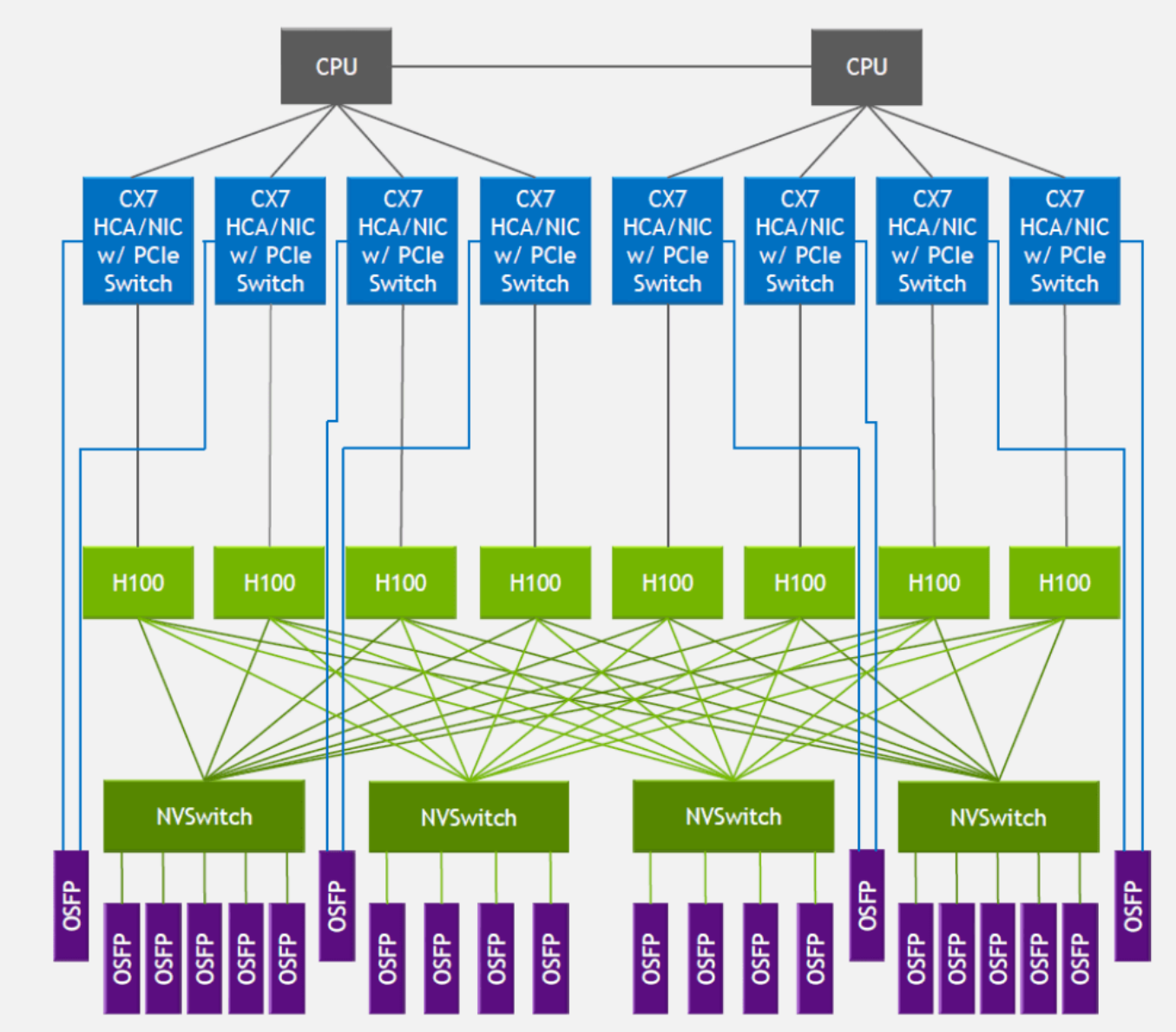
- Required for passing gradients in LLM training
- NCCL (Nvidia Collective Communication Library)
  - provides inter-GPU communication APIs
  - both collective and point-to-point send/receive primitives
  - supports various of interconnect technologies
    - PCIe
    - NVLink
    - InfiniBand
    - IP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

GPU Server



PCIe Gen5 provides 128GB/s bidirectional bandwidth (64GB/s ea)

GPU with NVSwitch



NVLink provides 900GB/s in aggregated bandwidth

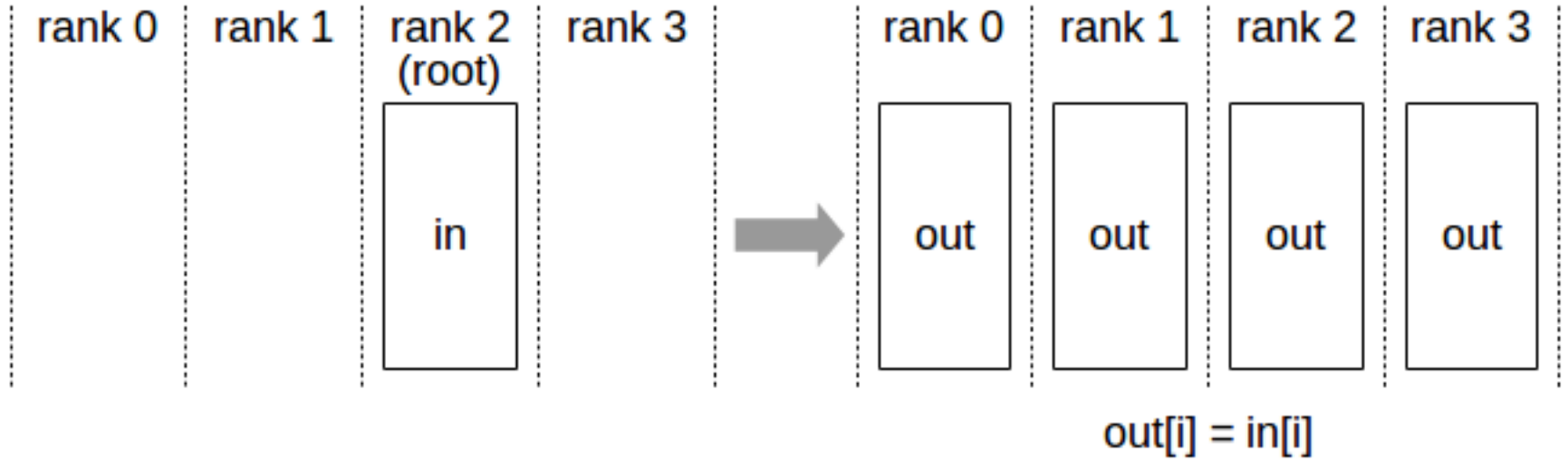
NVSwitch provides all-to-all communication

NCCL Primitives

- Broadcast
- Reduce
- ReduceScatter
- AllGather
- AllReduce

Broadcast

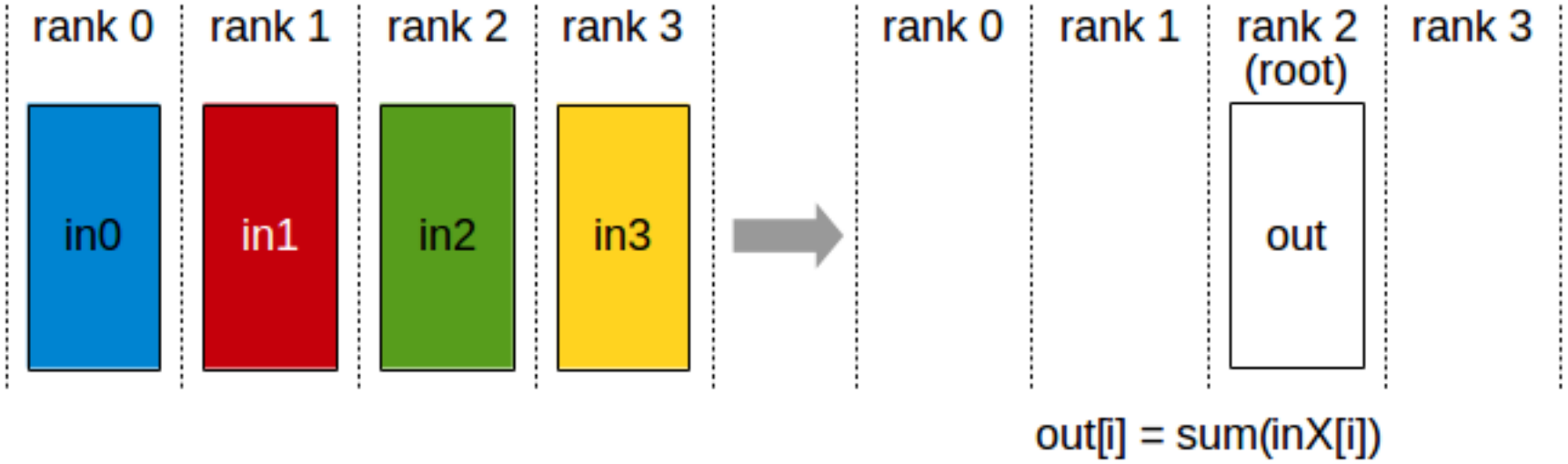
- The Broadcast operation copies an N-element buffer on the root rank to all ranks (devices).



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

Reduce

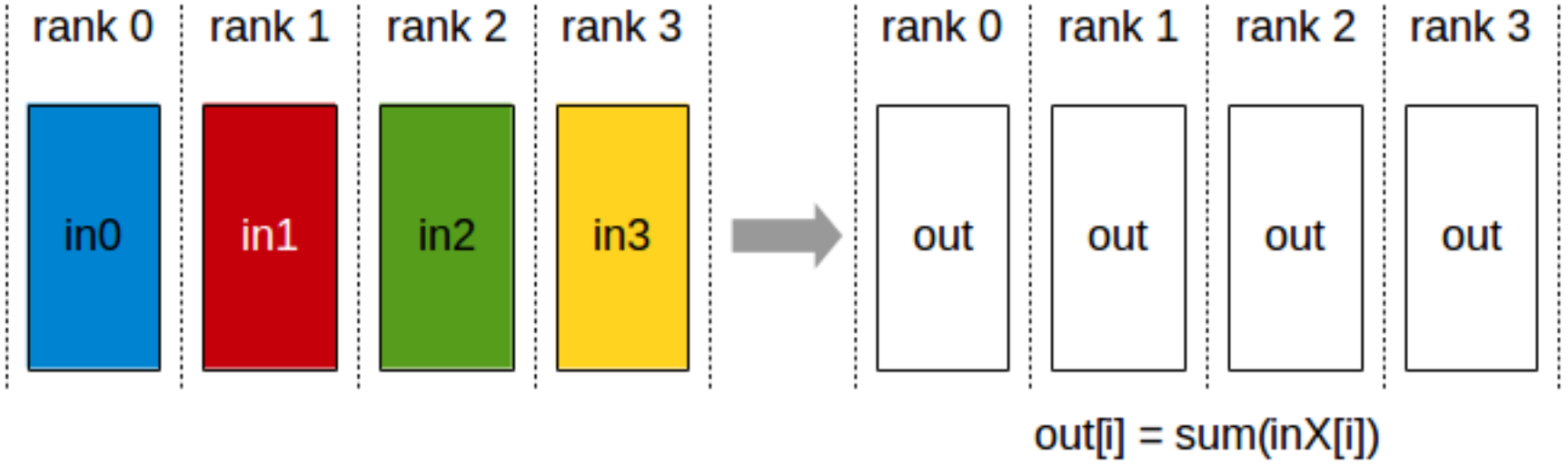
- Compute reduction (max, min, sum) across devices and write on one rank (device)



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)

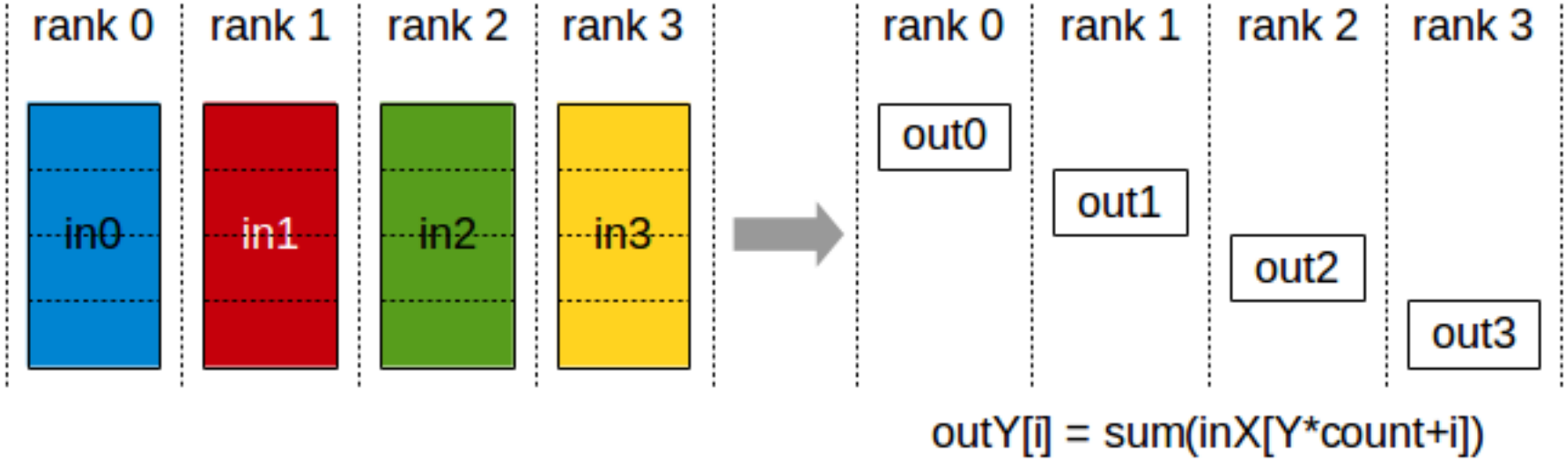
- Compute reduction (sum, min, max) across devices and writing the result in the receive buffers of every rank.



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

ReduceScatter

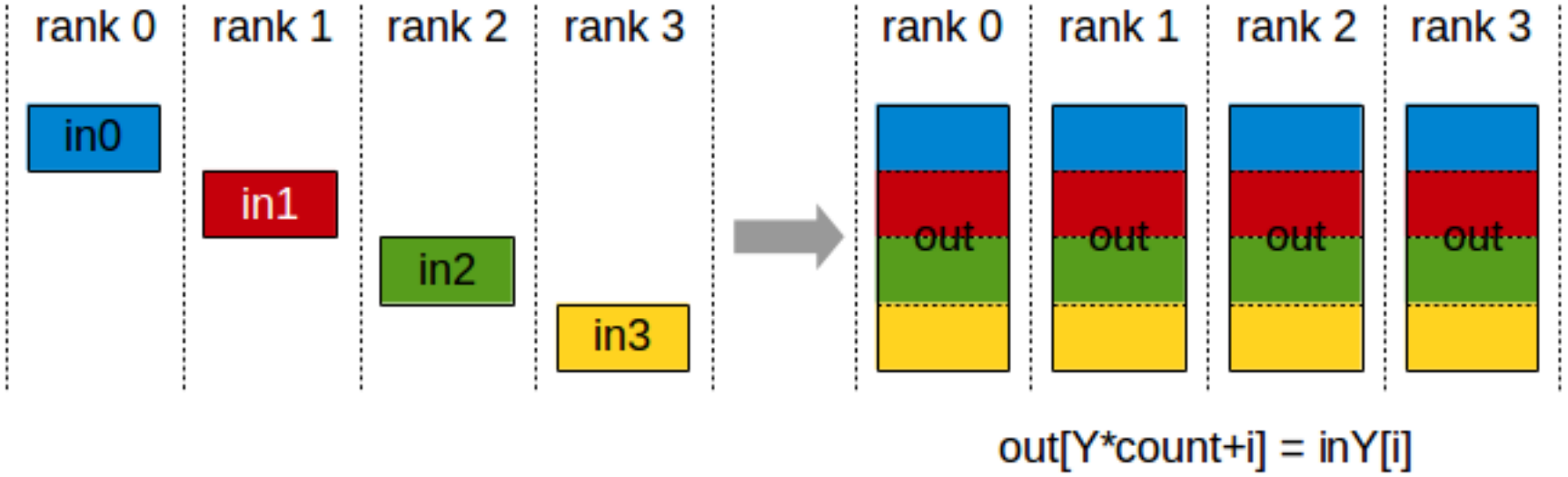
- Compute reduction (sum, min, max) and writing parts of results scattered in ranks



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

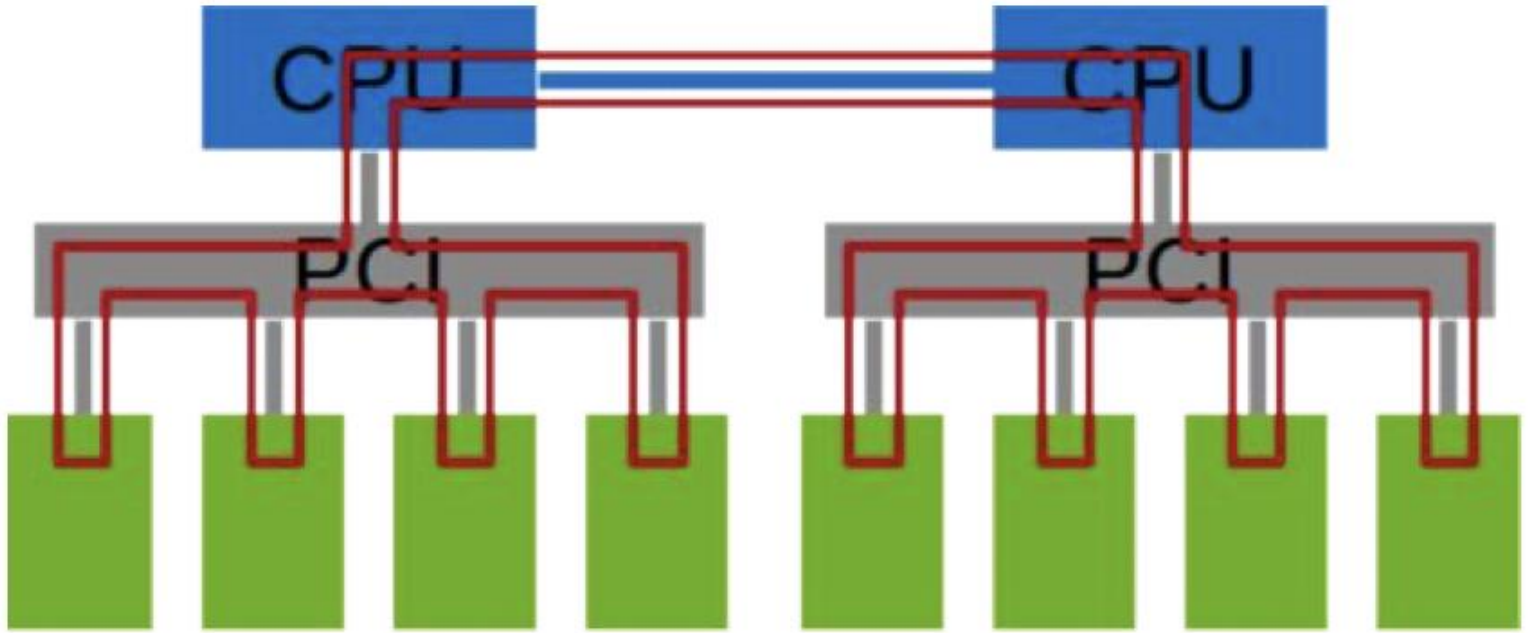
- device memory local to the CUDA device
- host memory registered using cudaHostRegister or cudaGetDevicePointer
- managed and unified memory.

Point-to-Point Communication

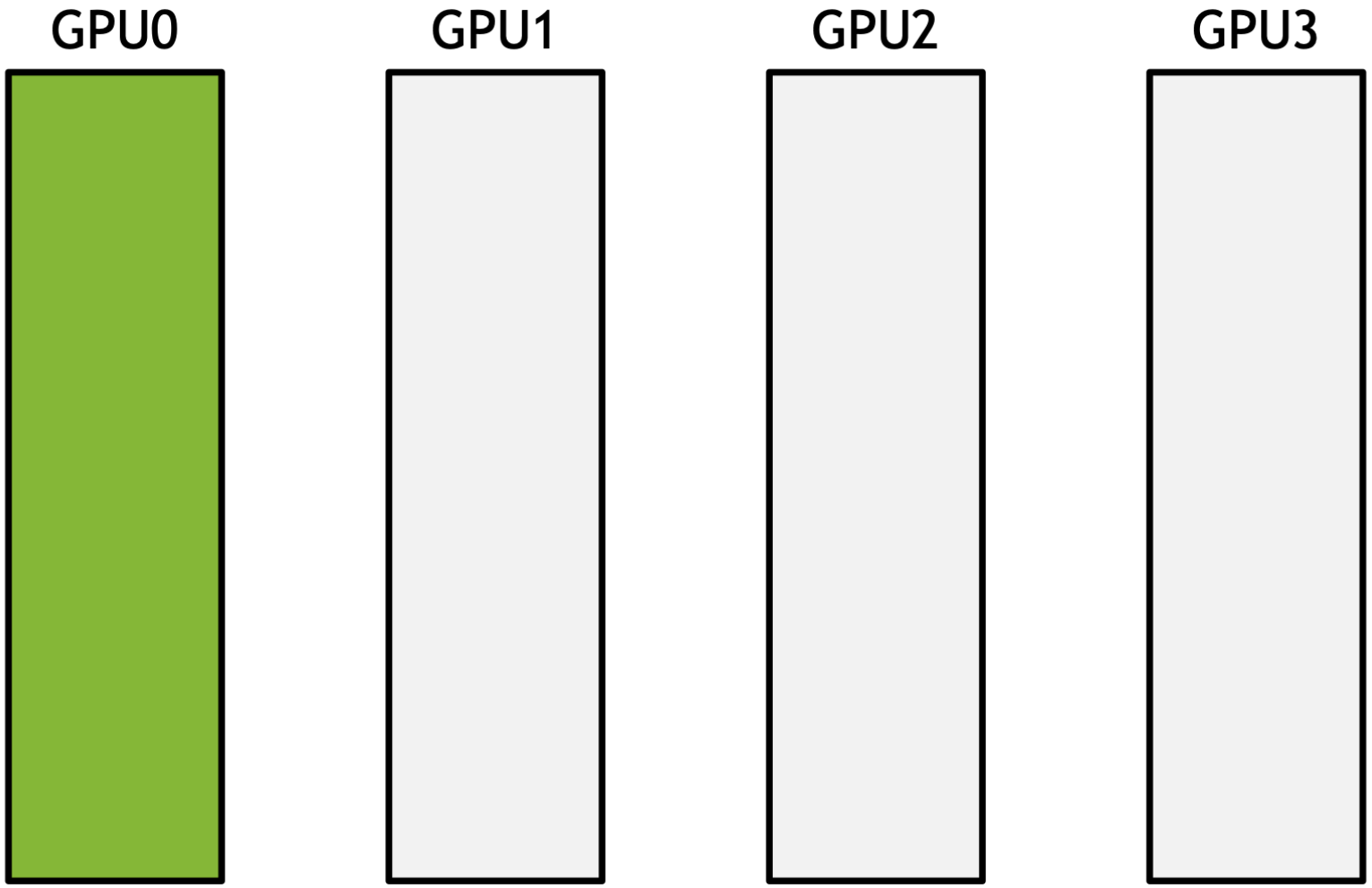
```
ncclGroupStart();

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

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

ncclGroupEnd();
```

How Broadcast/Reduce is Implemented?

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

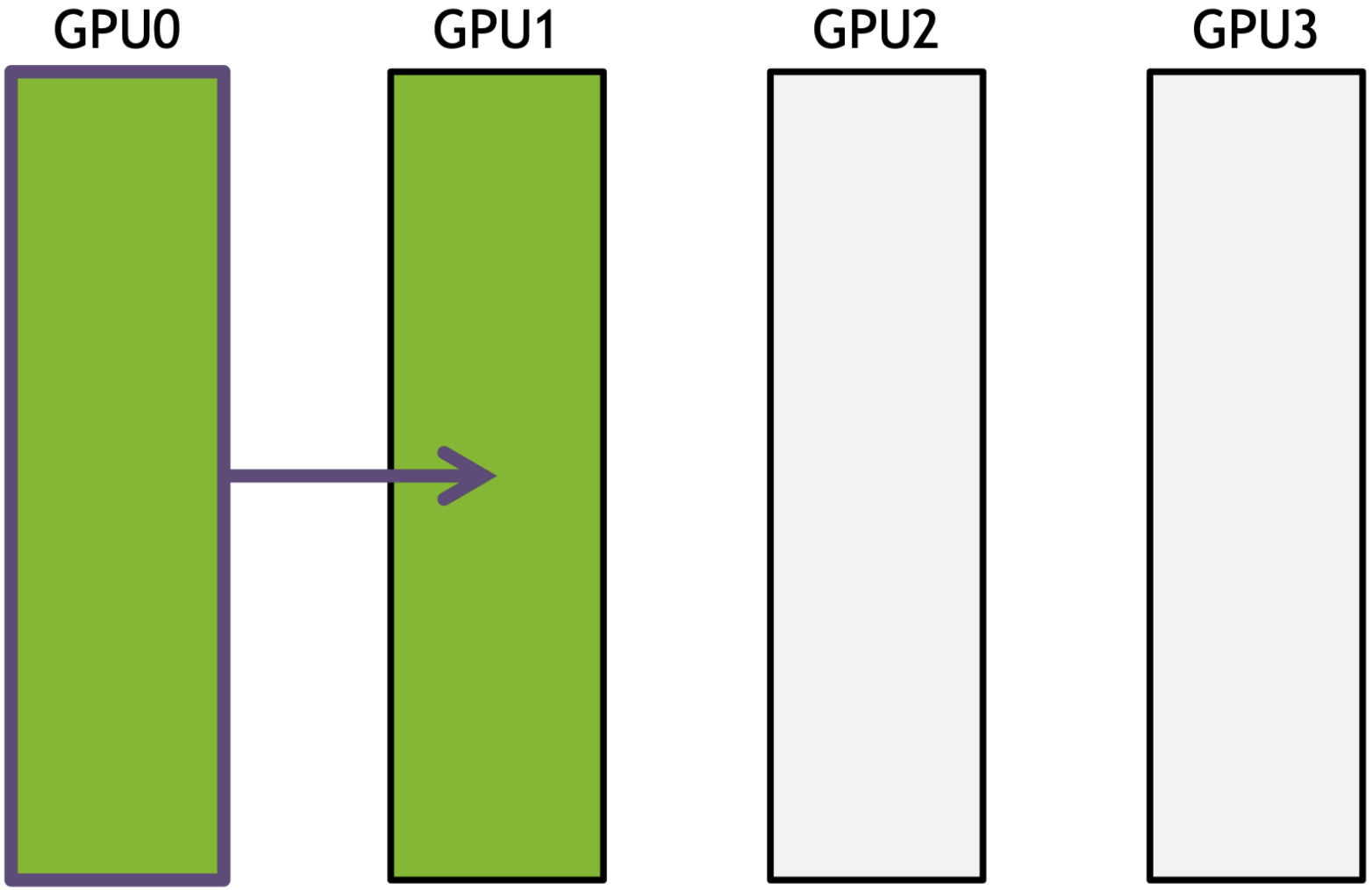


Broadcast with unidirectional ring



N=bytes to transfer
B=bandwidth

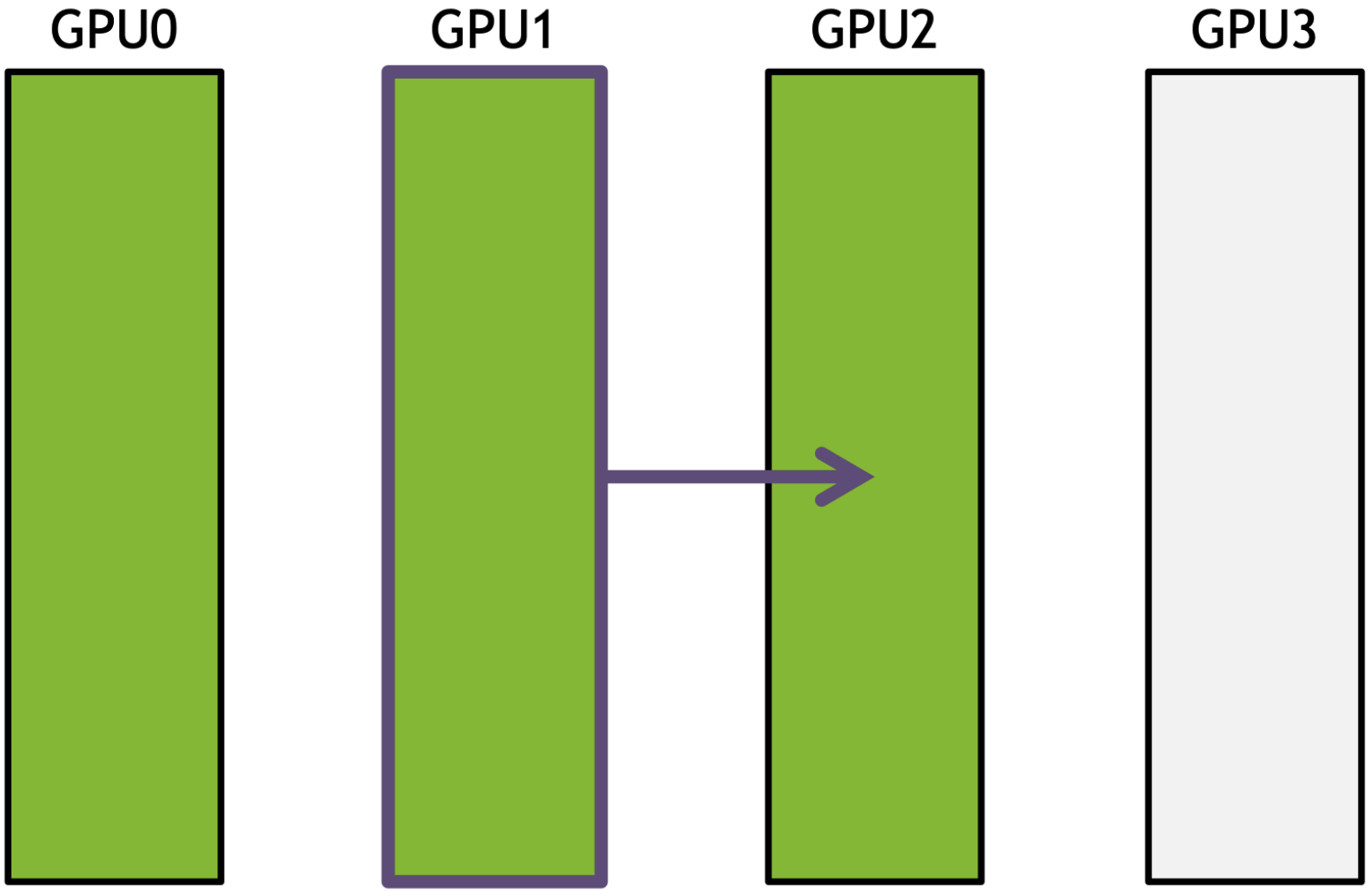
Broadcast with unidirectional ring



Step 1: t = N/B

N=bytes to transfer
B=bandwidth

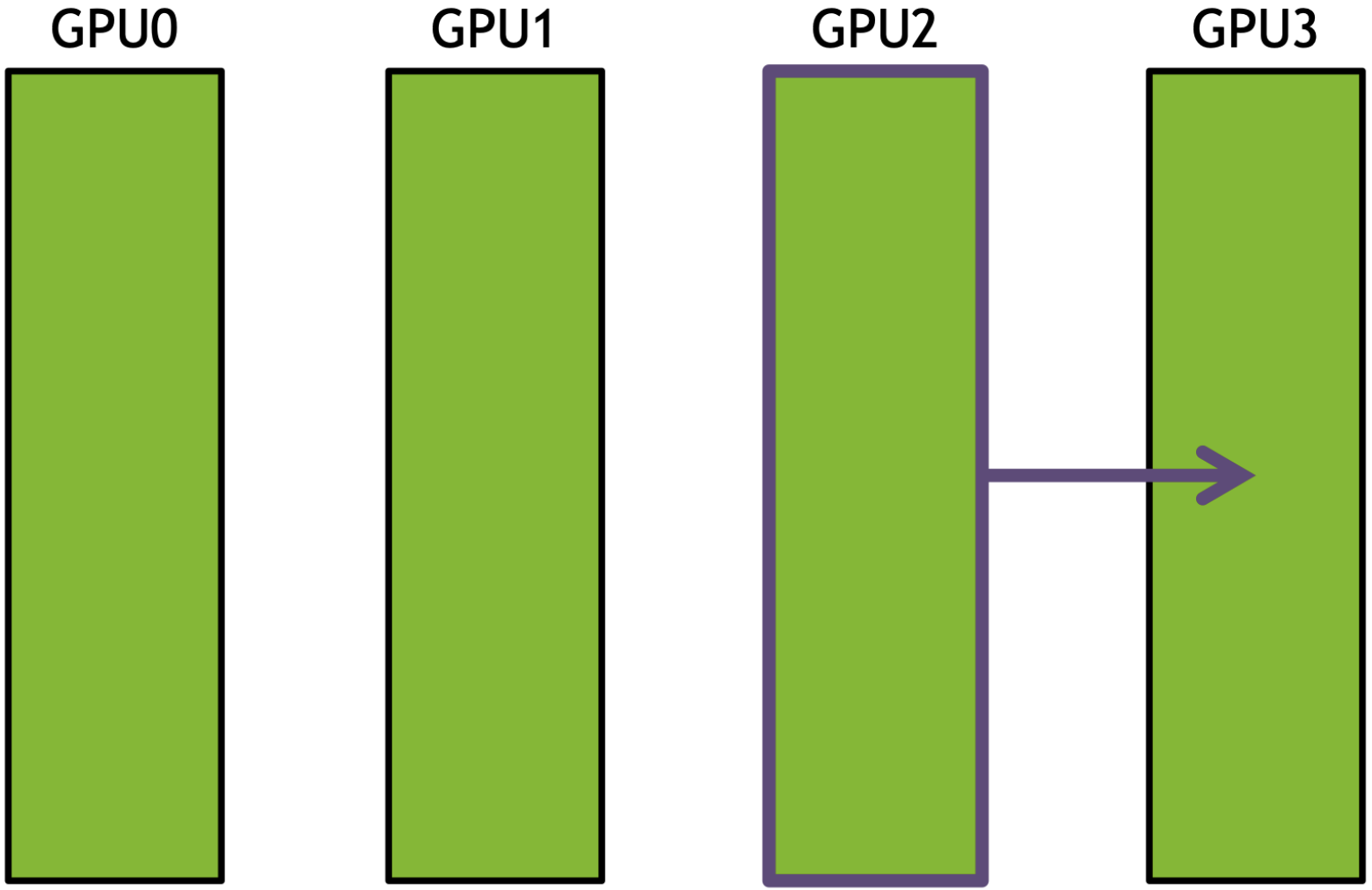
Broadcast with unidirectional ring



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

N=bytes to transfer
B=bandwidth

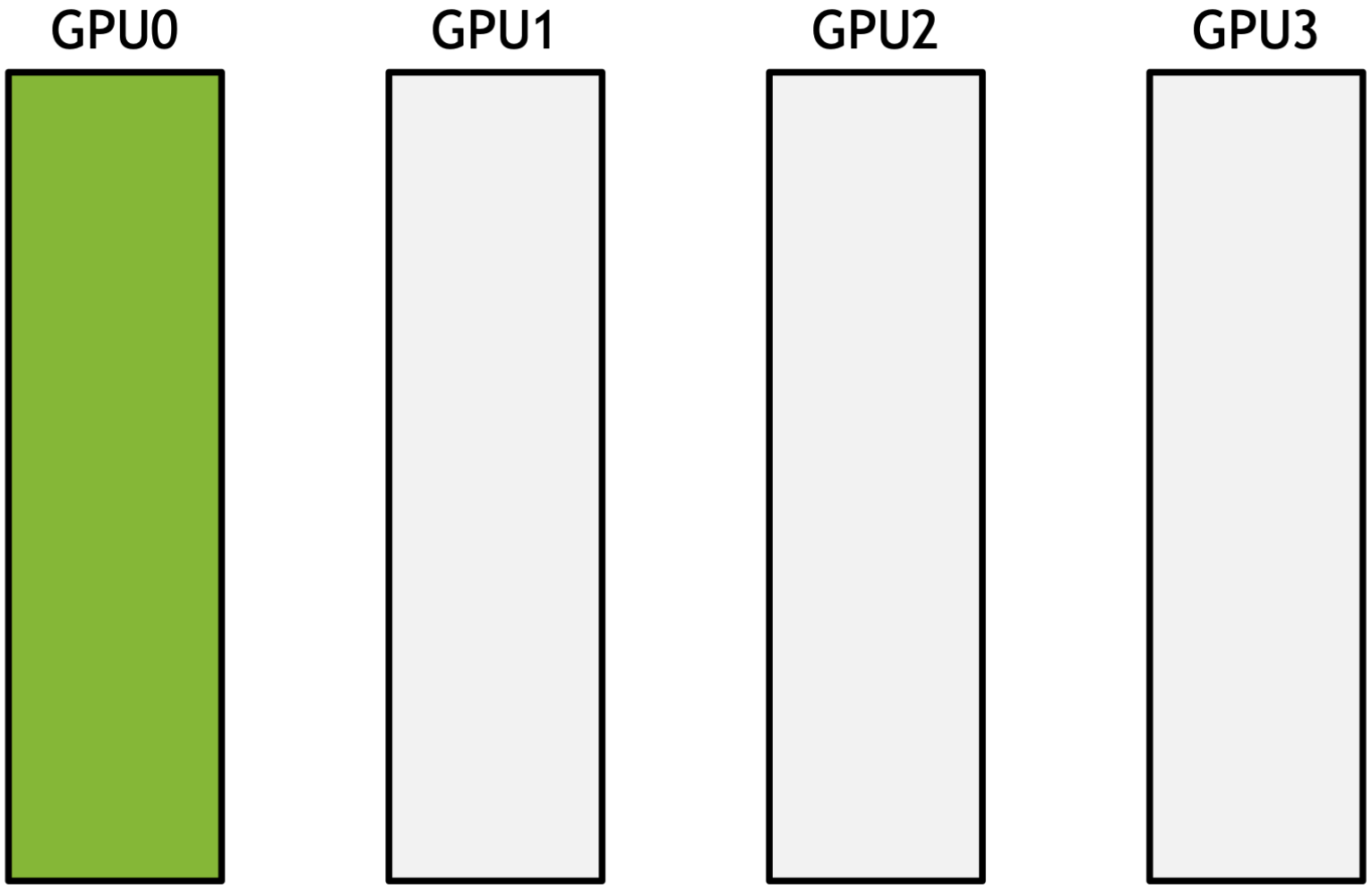
Broadcast with unidirectional ring



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

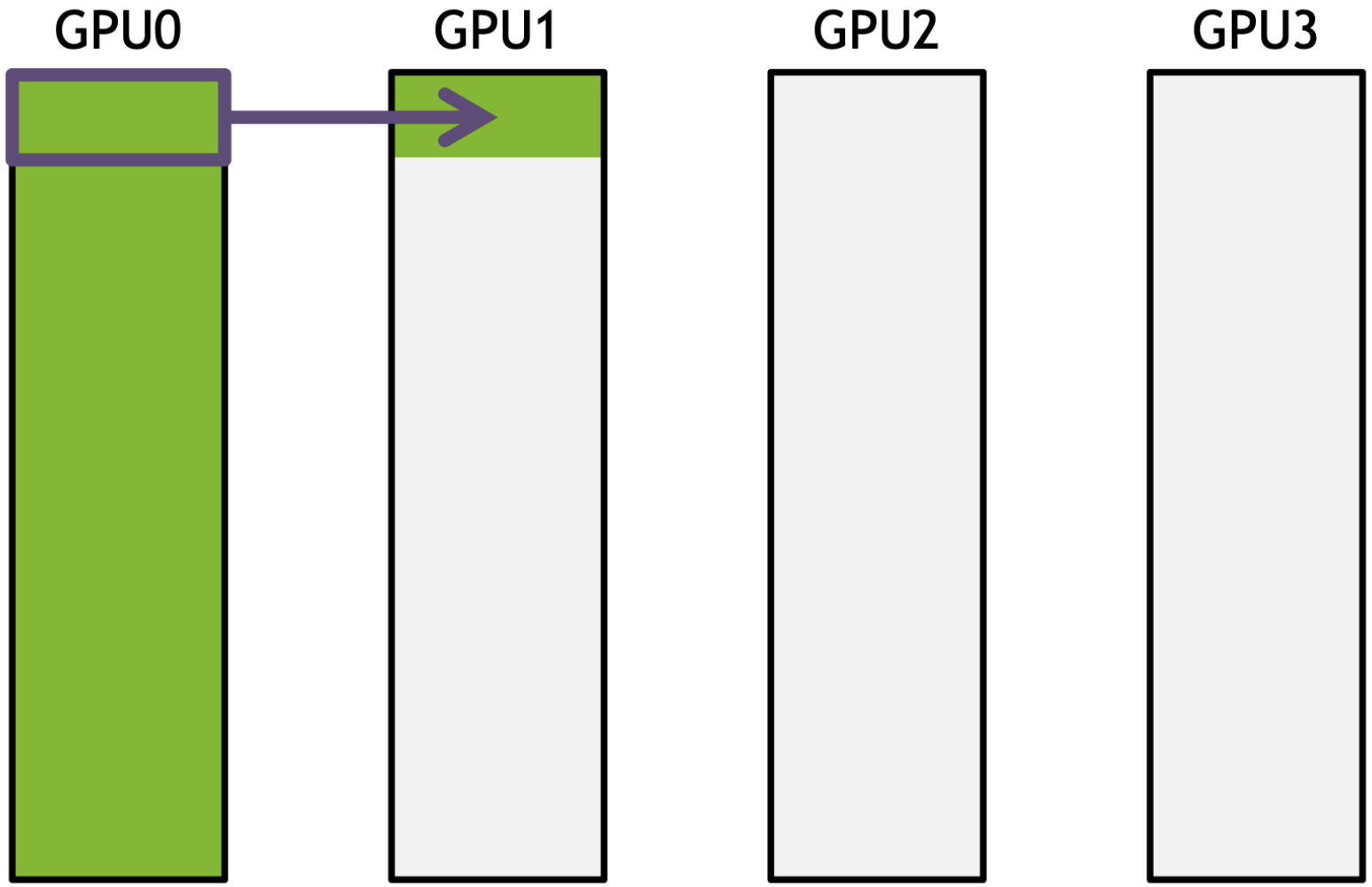
Broadcast with unidirectional ring



N=bytes to transfer
B=bandwidth

Broadcast with unidirectional ring

break data into S messages



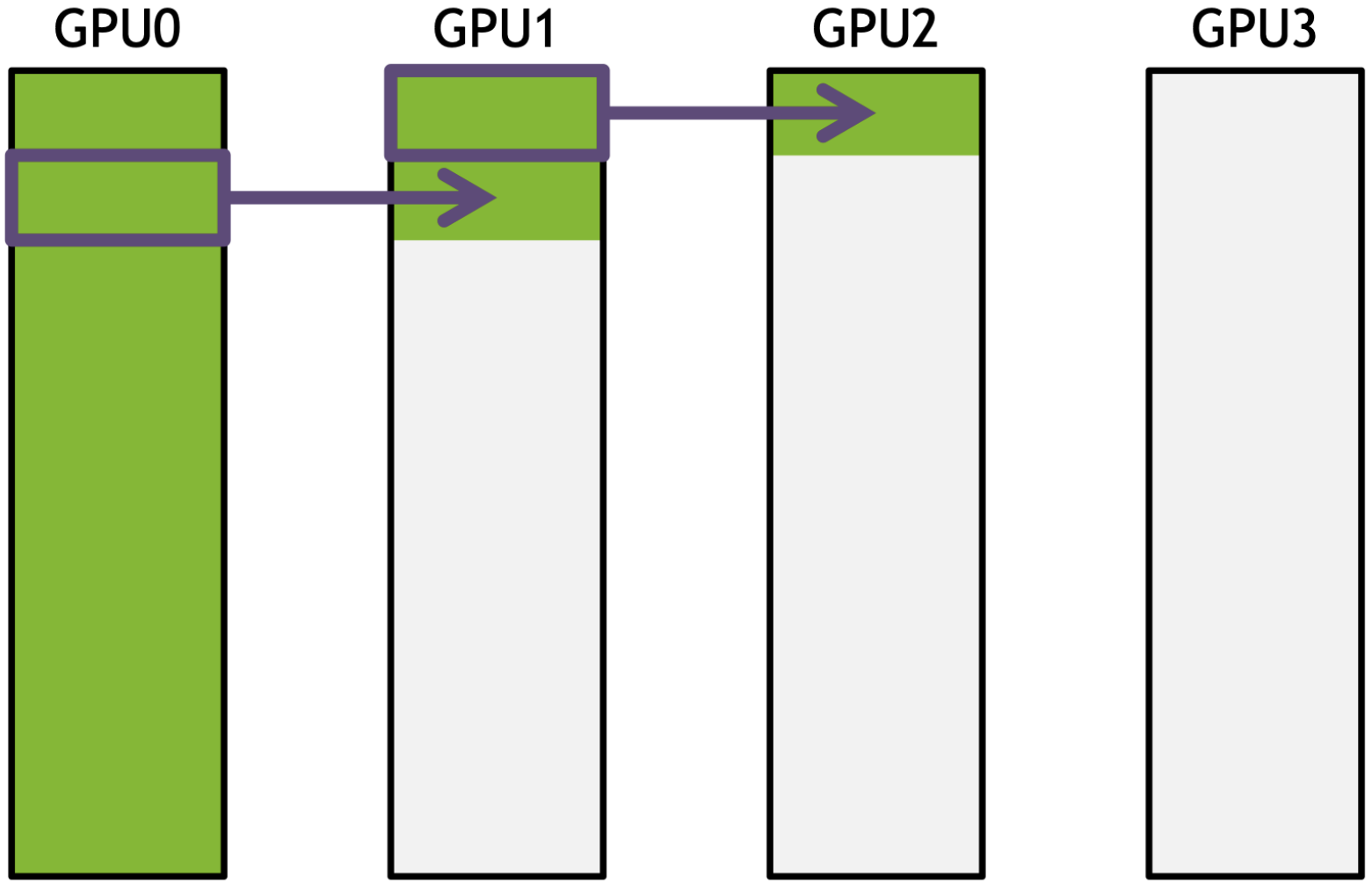
Step 1: t = N/SB

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: t = N/SB

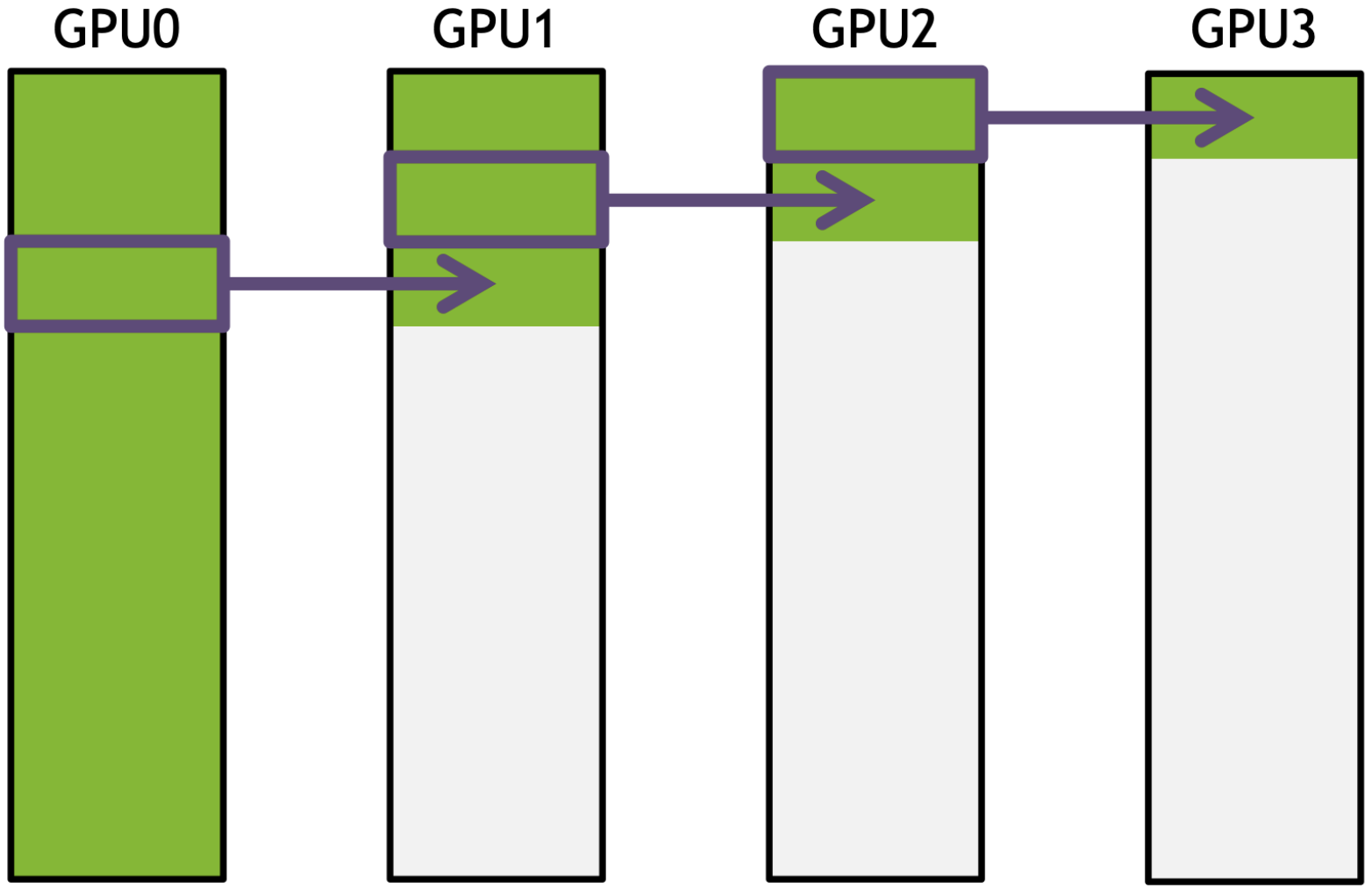
Step 2: t = N/SB

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

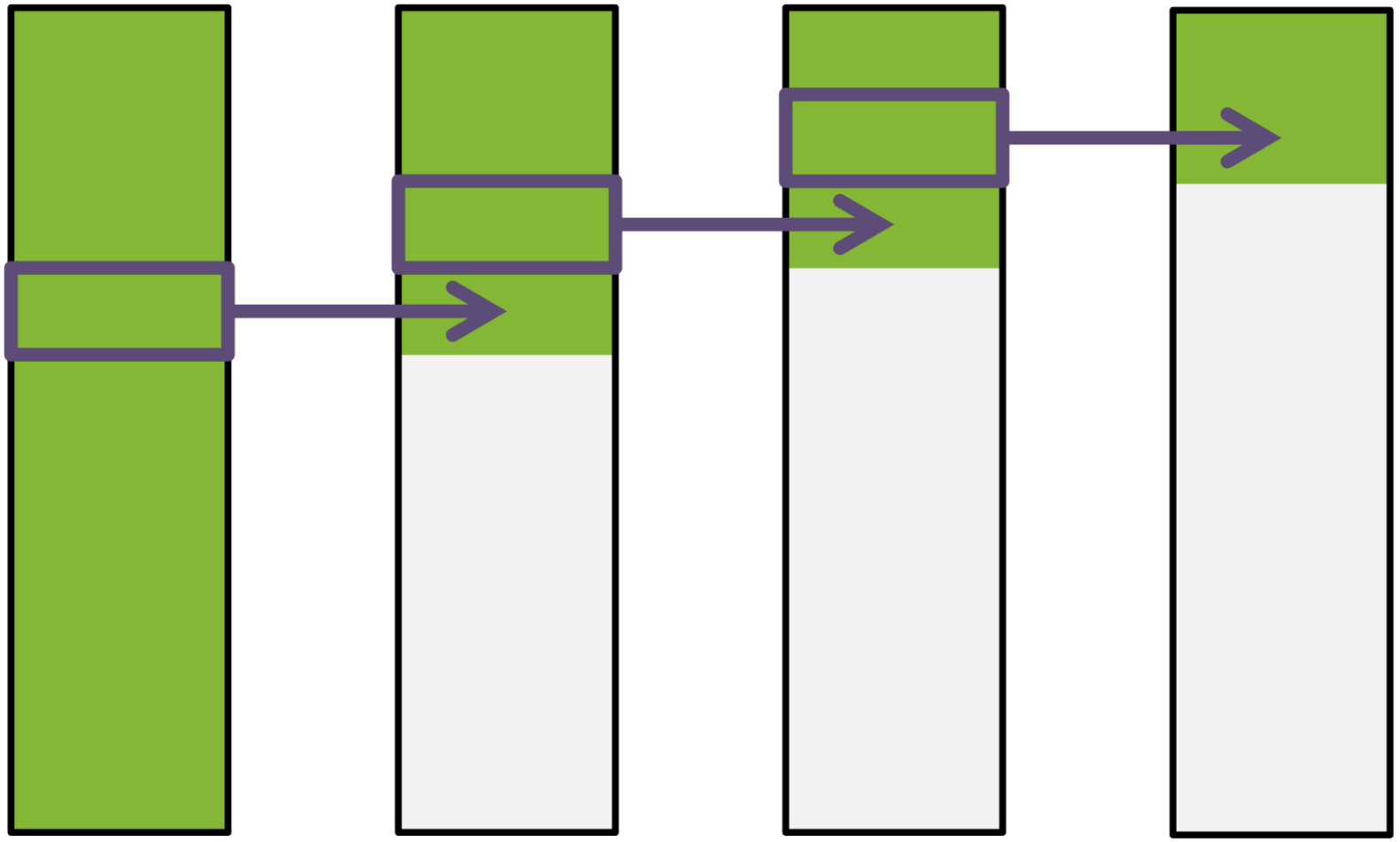
break data into S messages



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

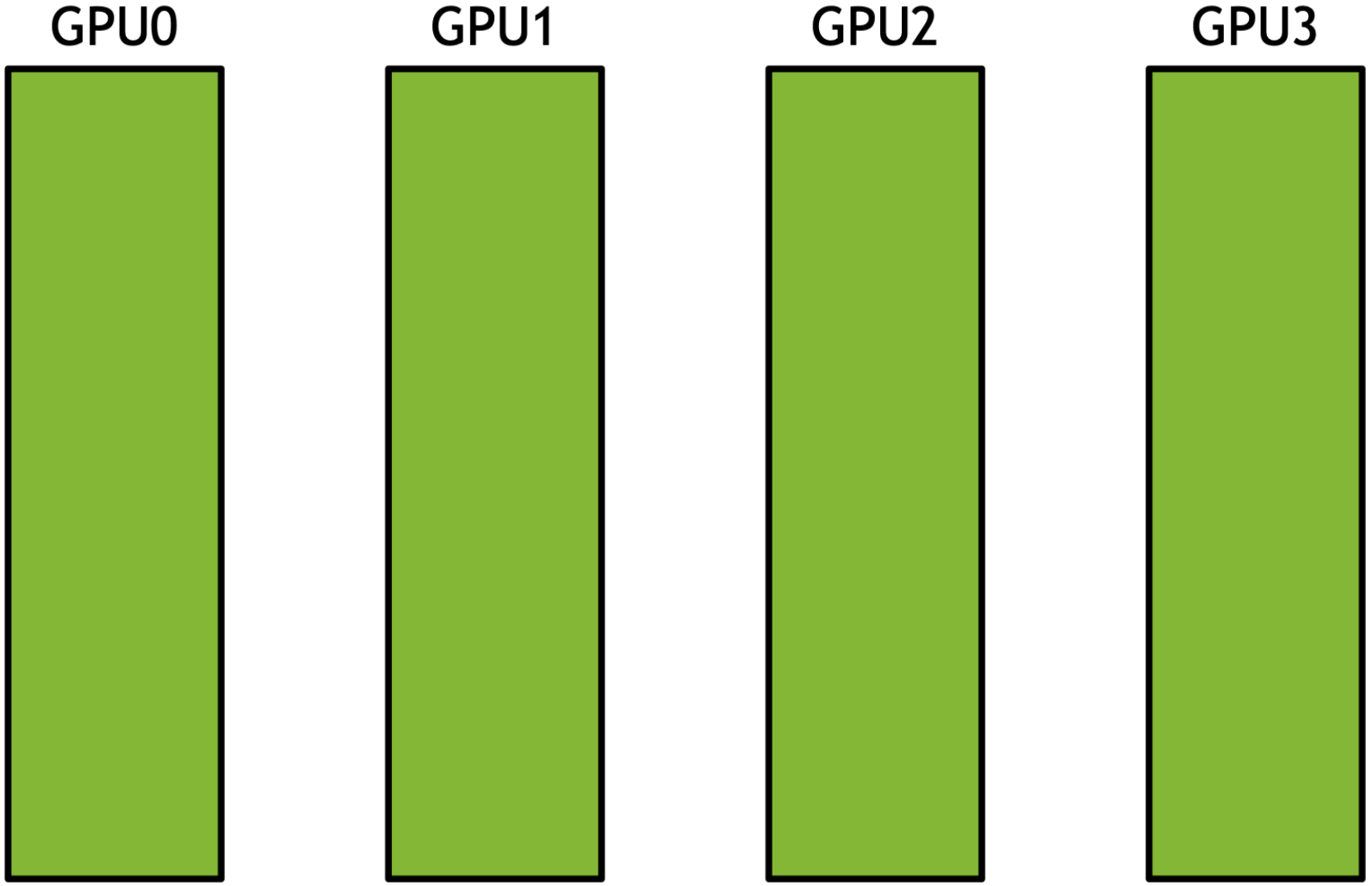


break data into S messages

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

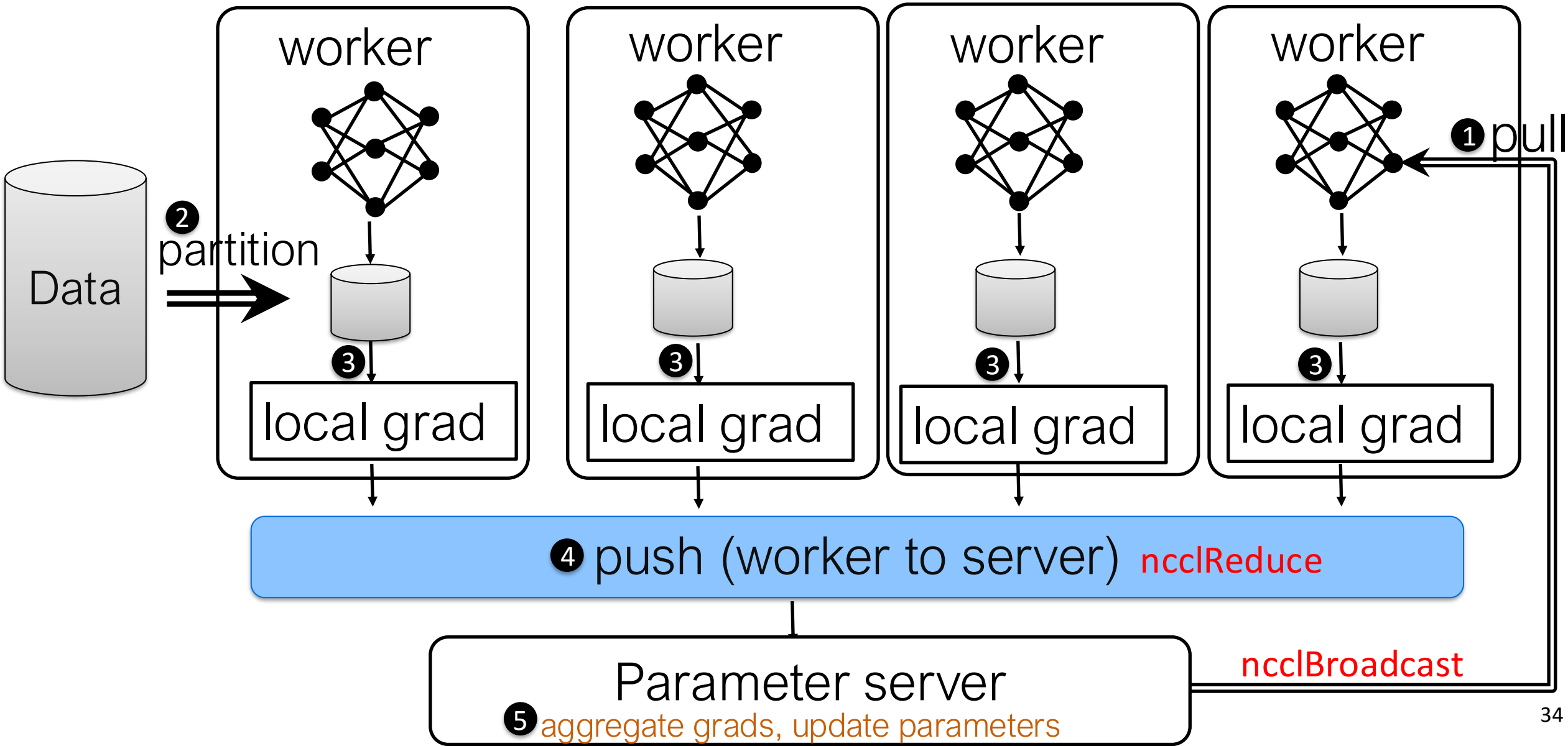
N=bytes to transfer

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]));
```

Classical Distributed Training: Parameter Server



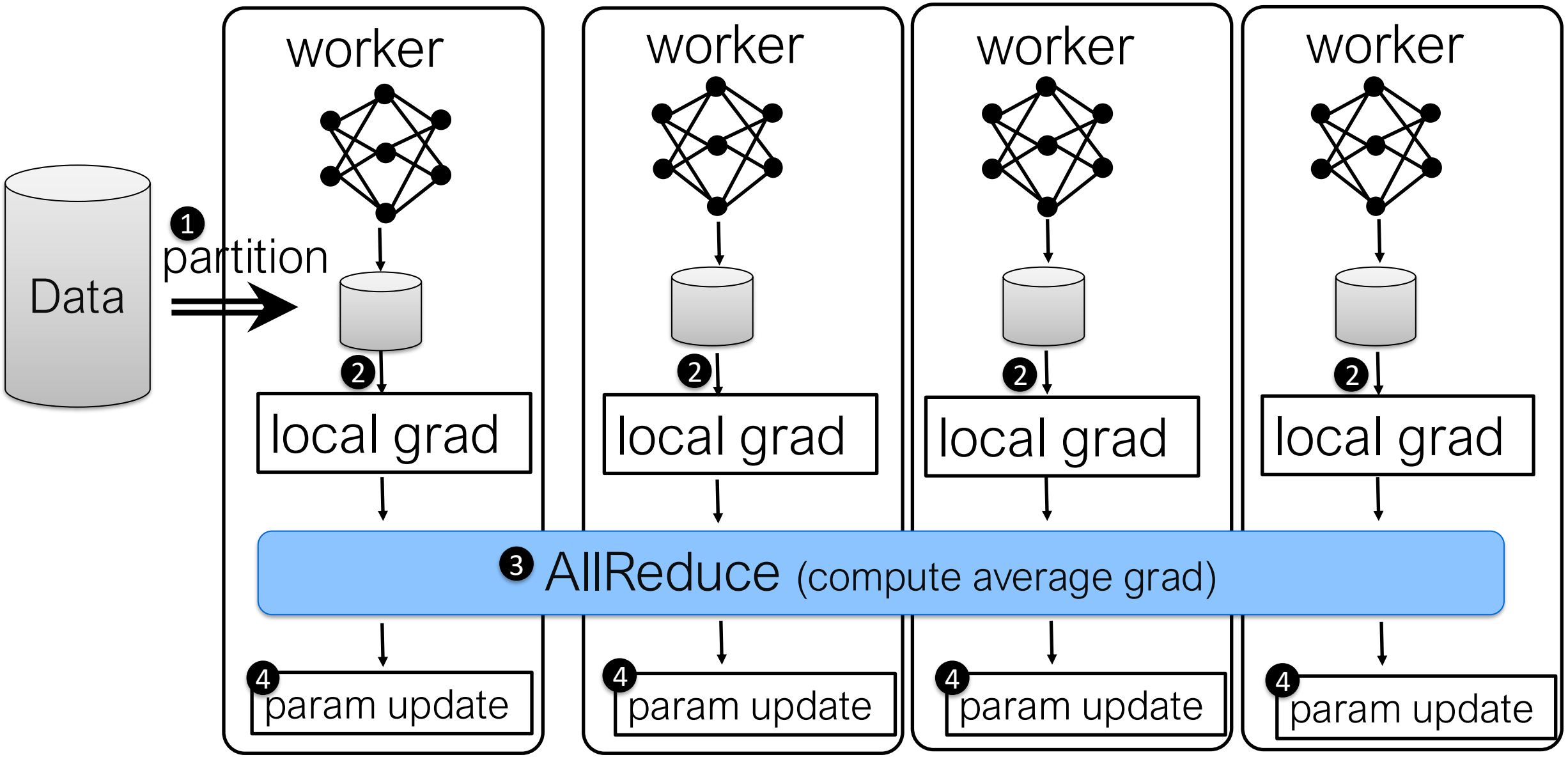
Implementing Parameter Server using NCCL

- 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

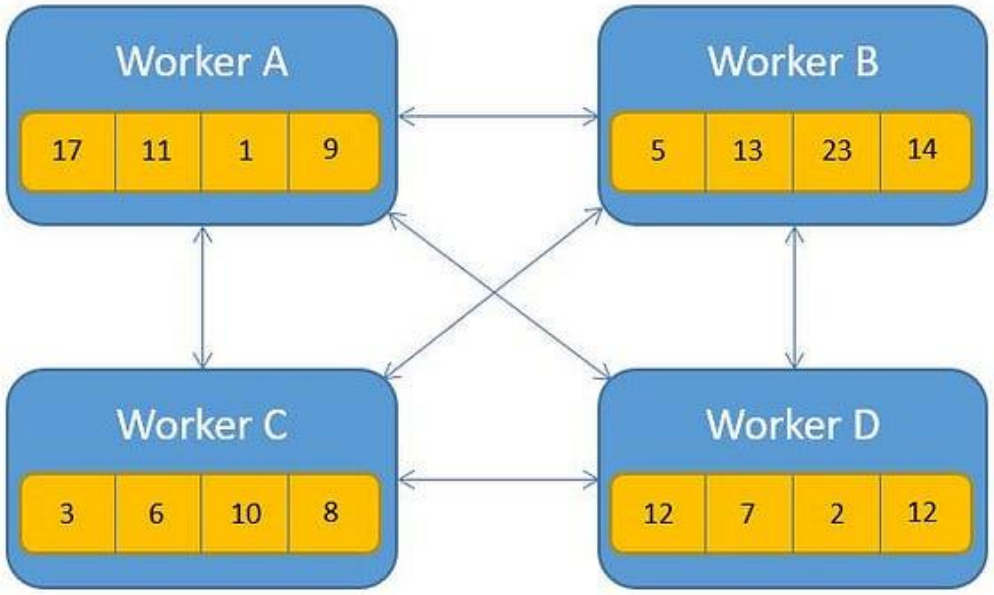
- Overview of large-scale model training
- Multi-GPU communication
- Data Parallel Training via AllReduce
- Distributed Data Parallel Training: hiding communication

Data Parallel Training

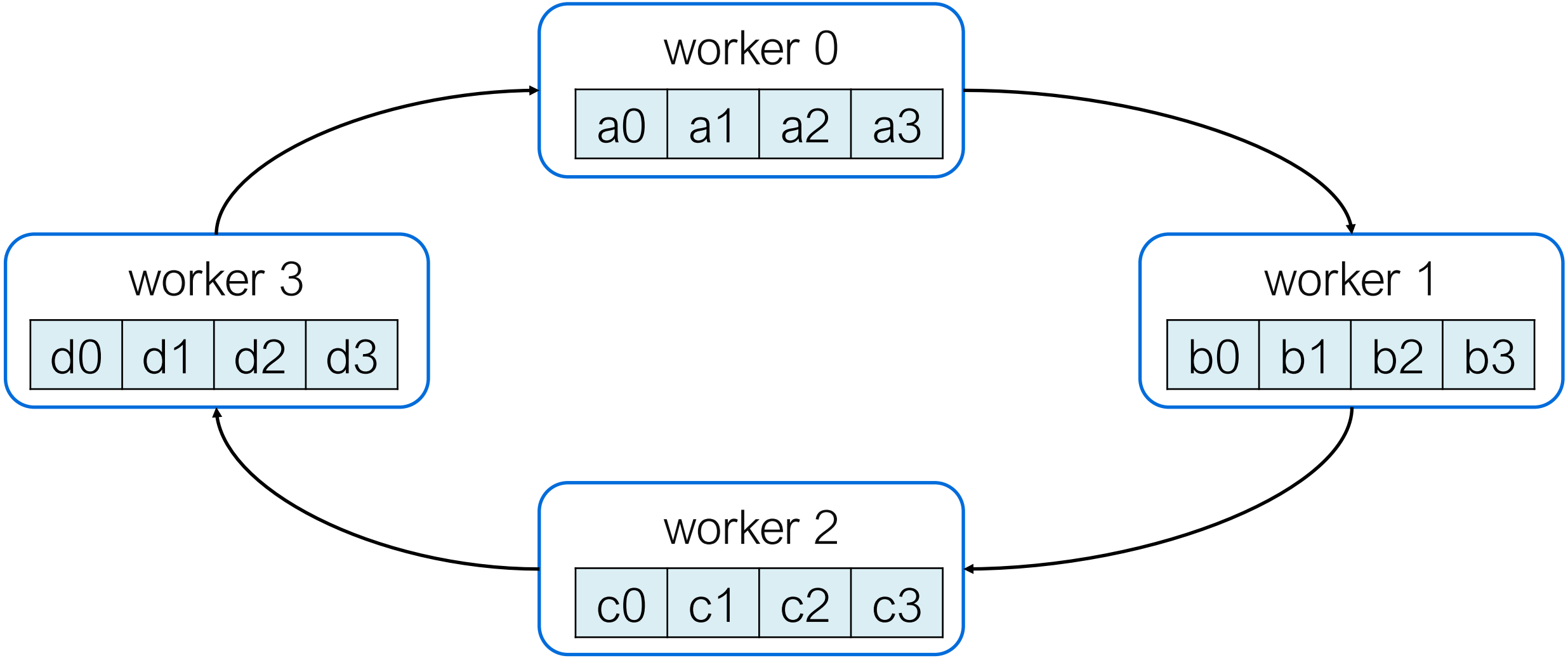


How to Synchronize Gradients?

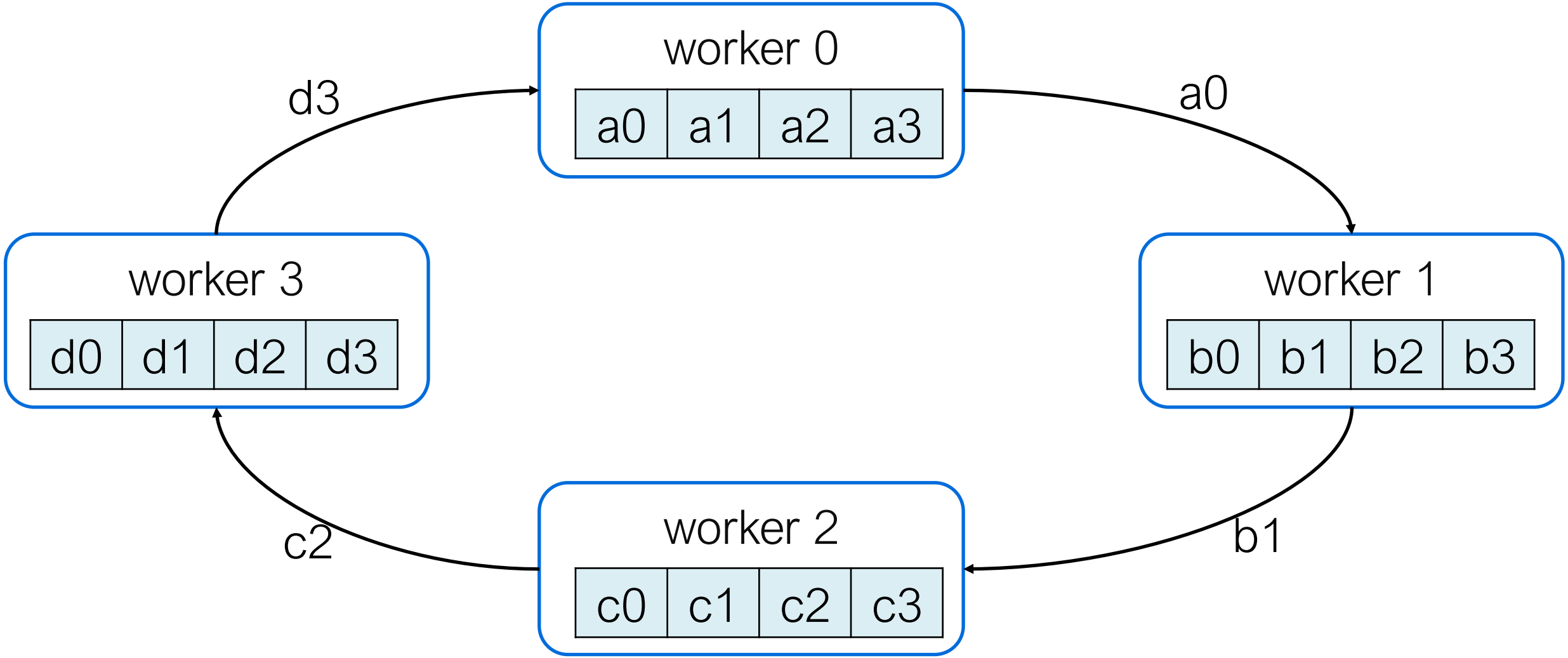
- Naïve all-reduce



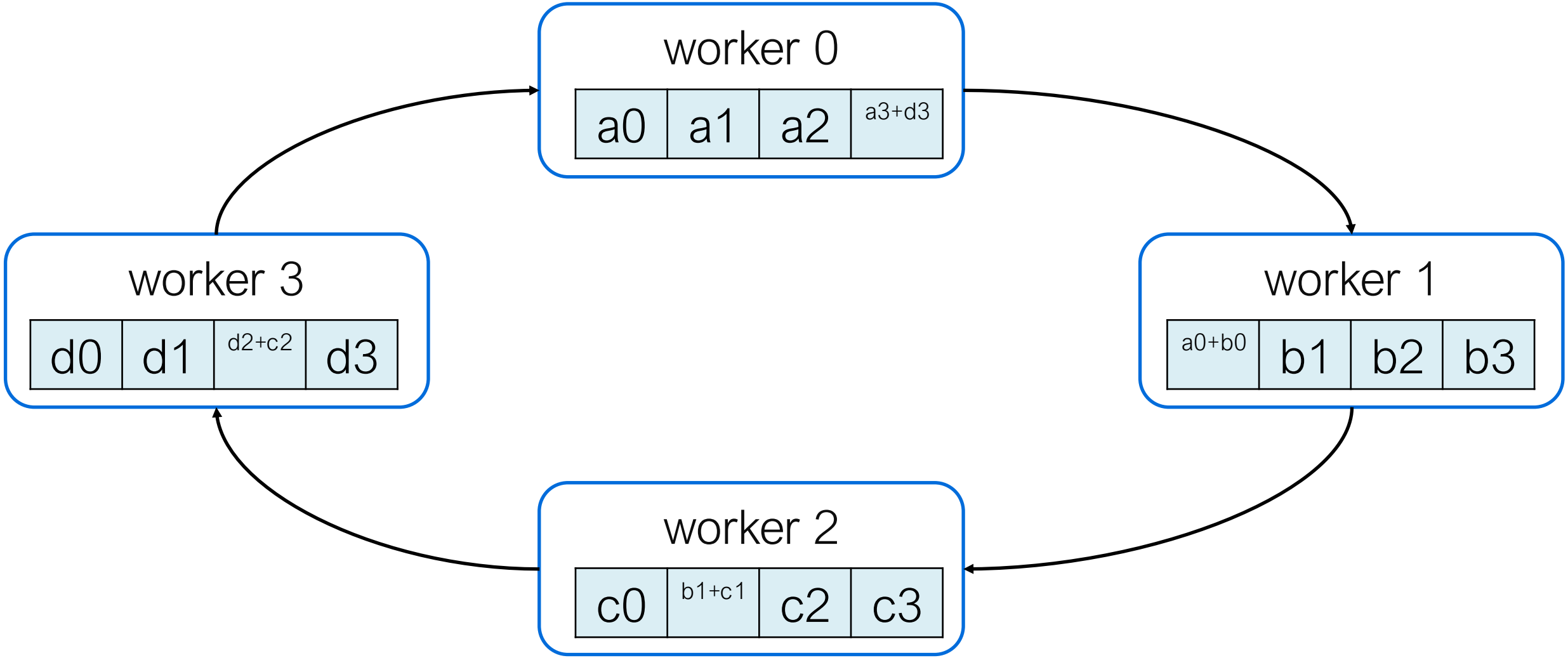
Ring AllReduce



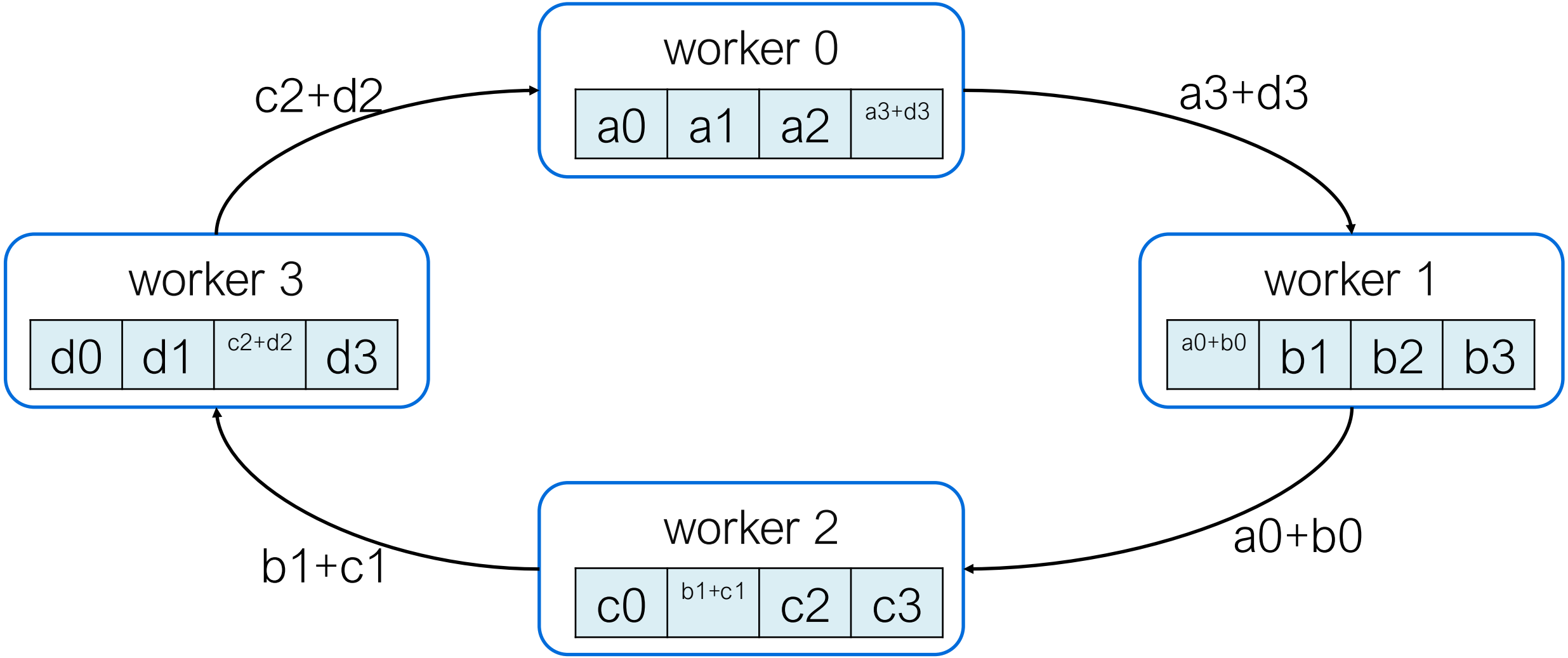
Ring AllReduce



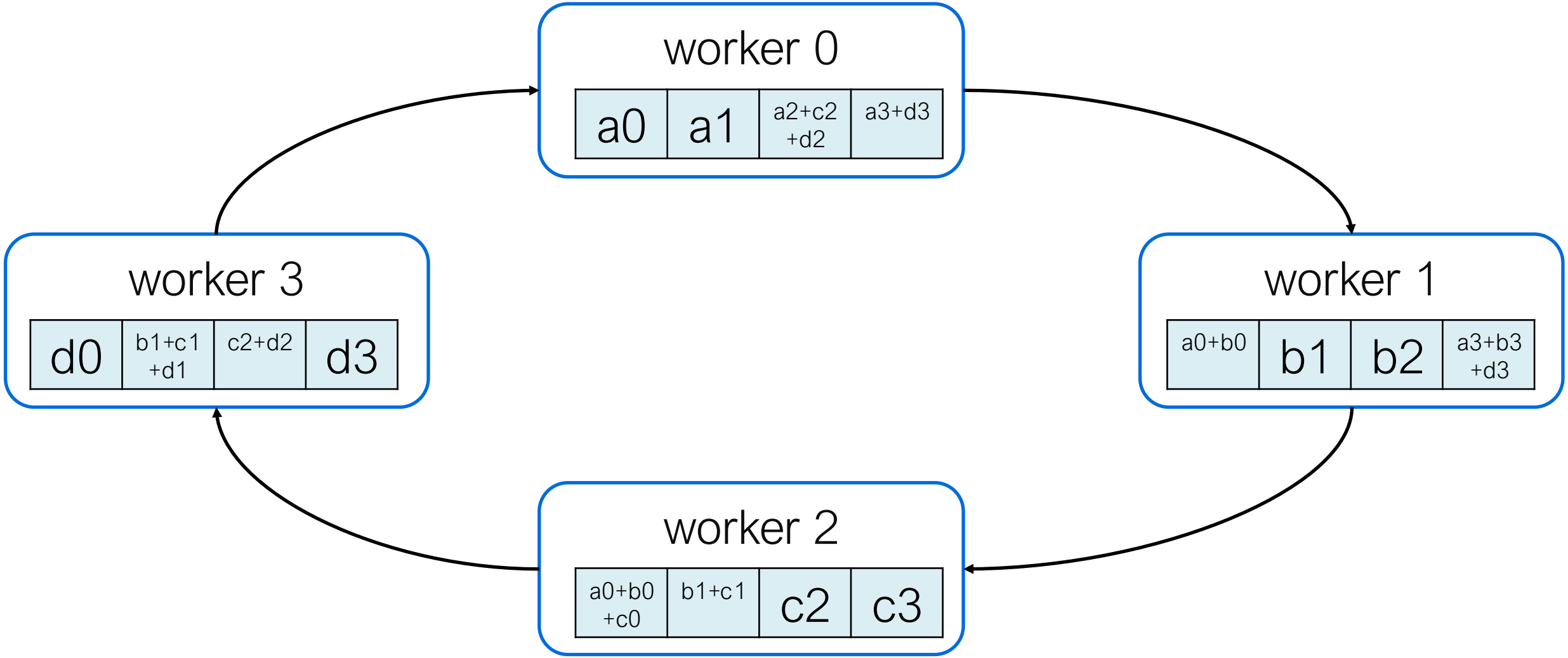
Ring AllReduce



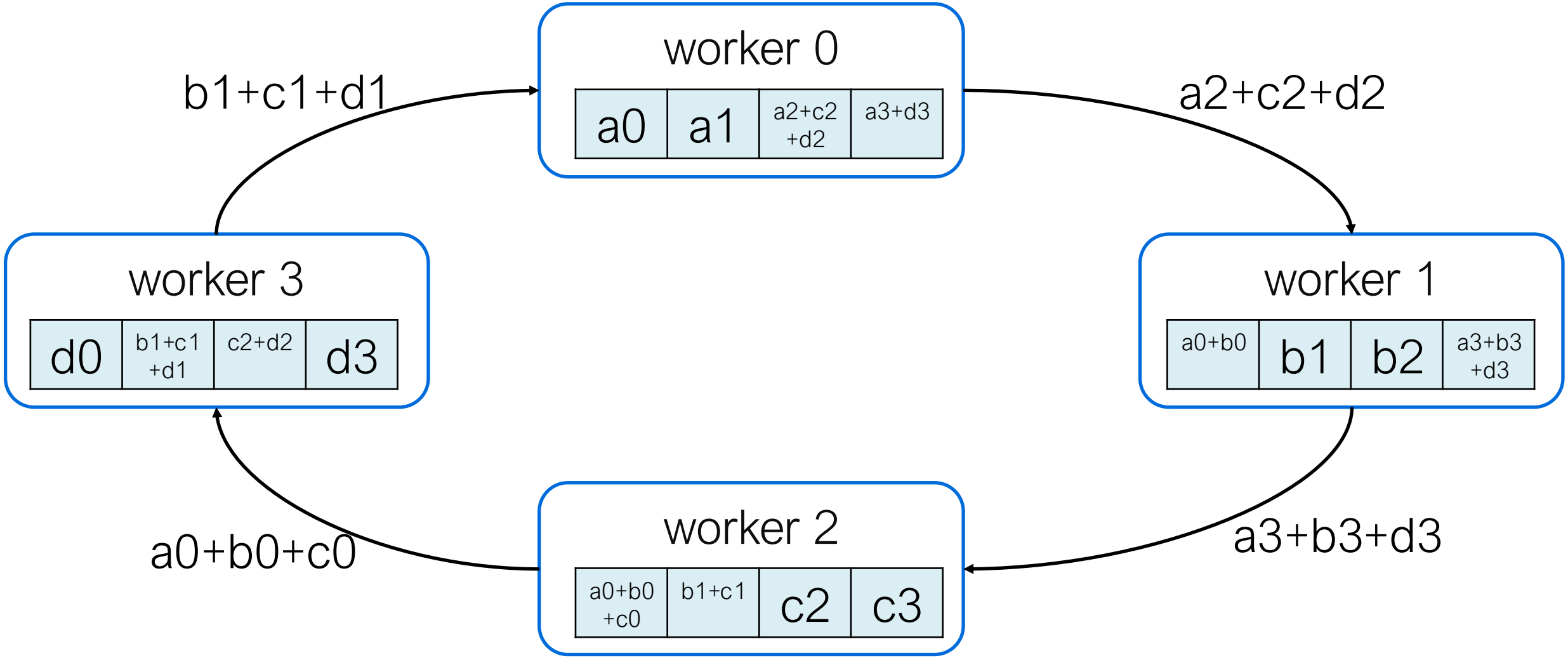
Ring AllReduce



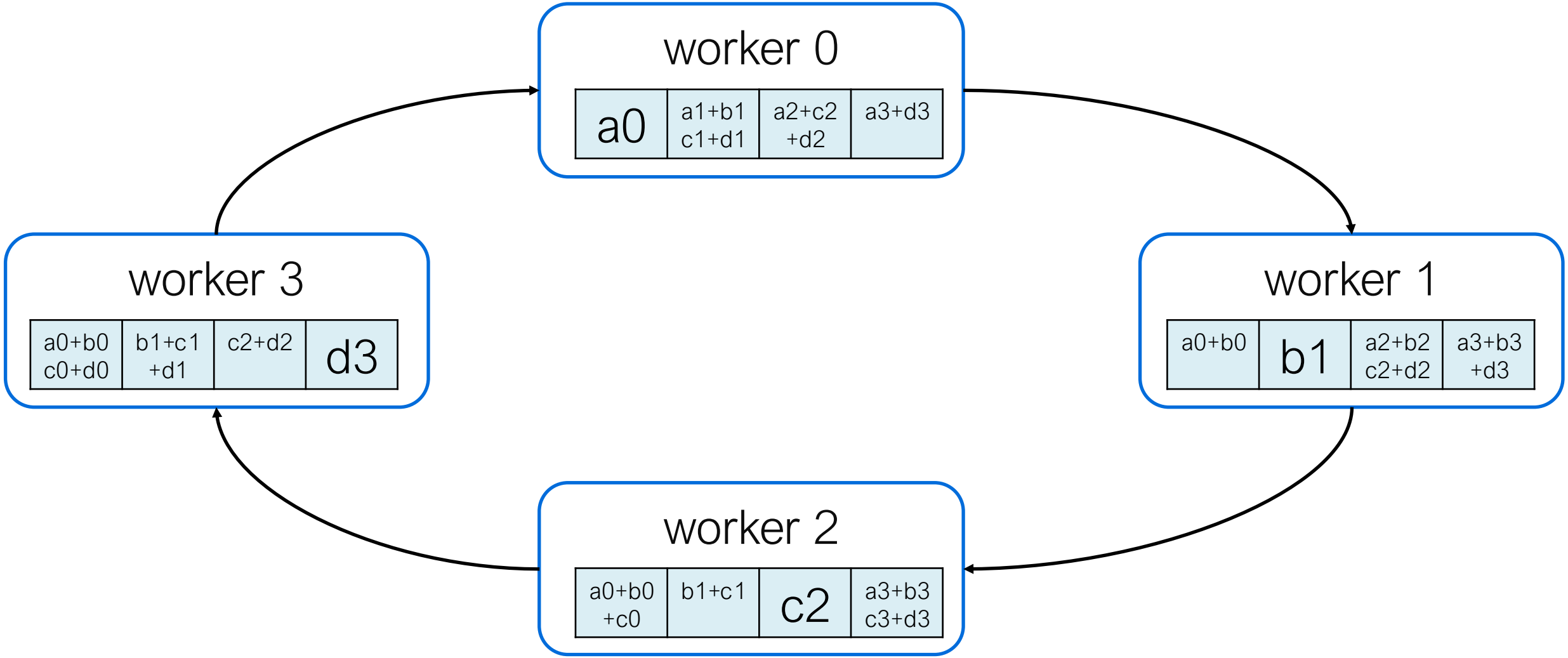
Ring AllReduce



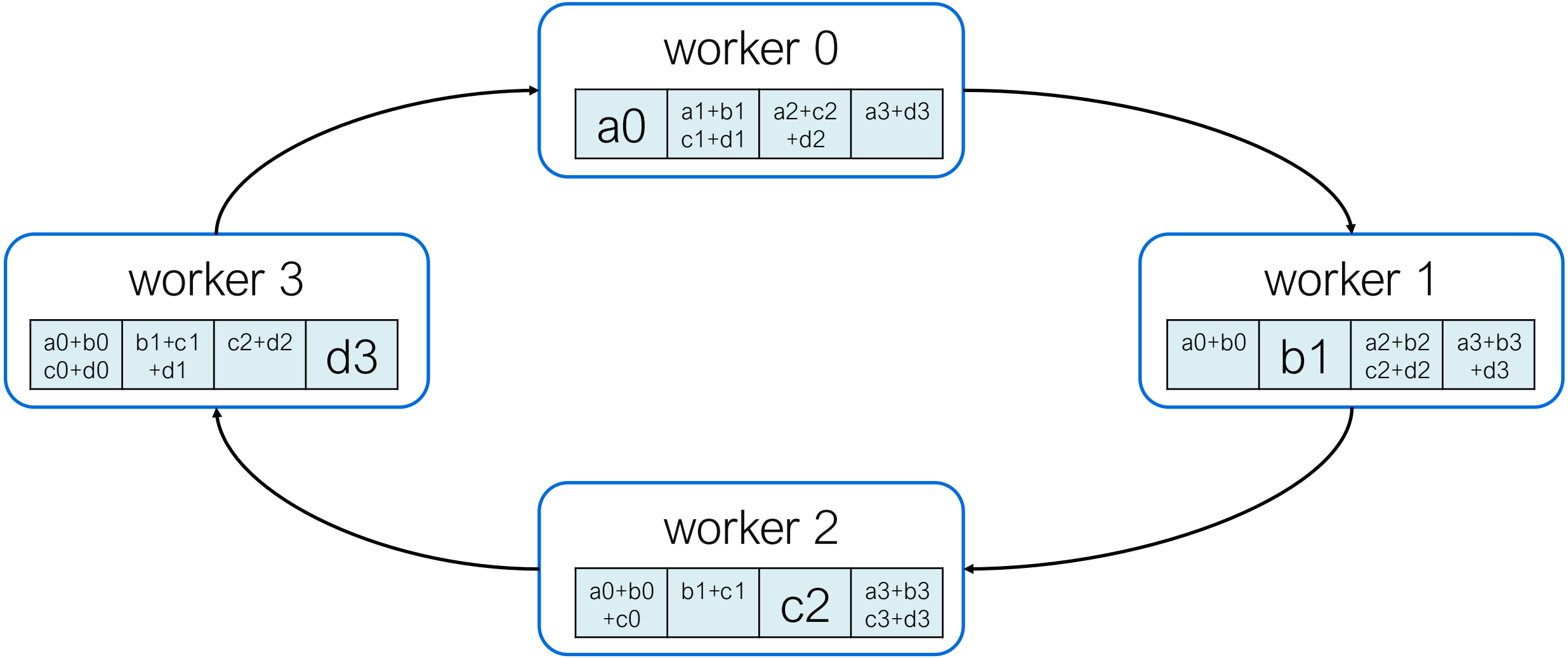
Ring AllReduce



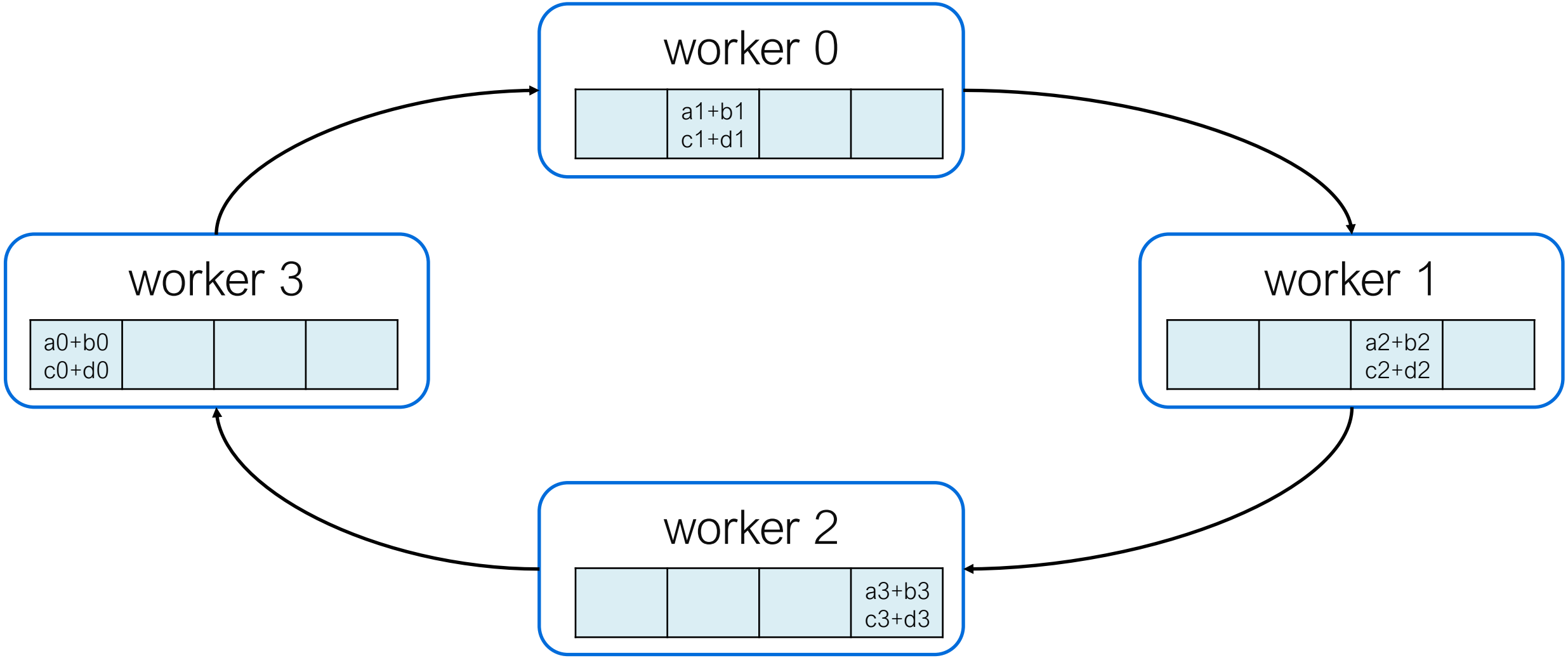
Ring AllReduce



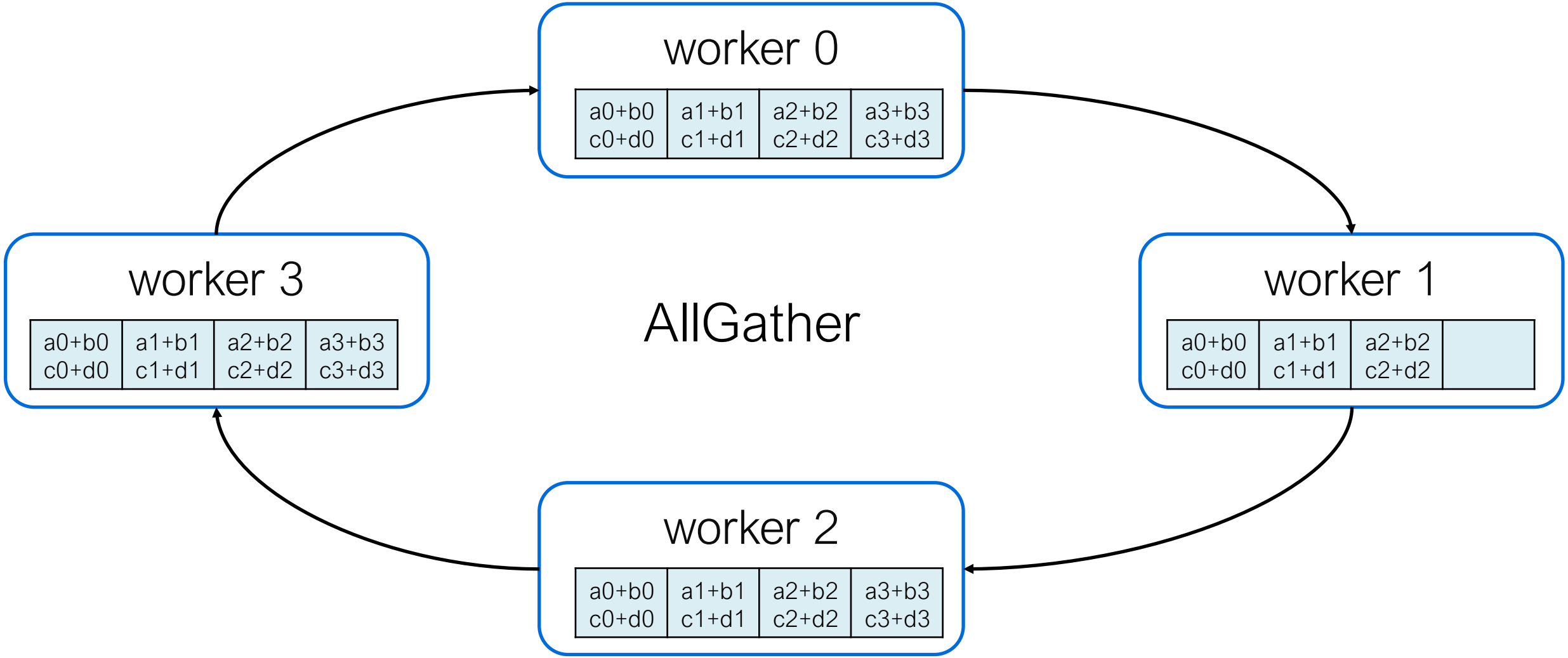
Ring AllReduce



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

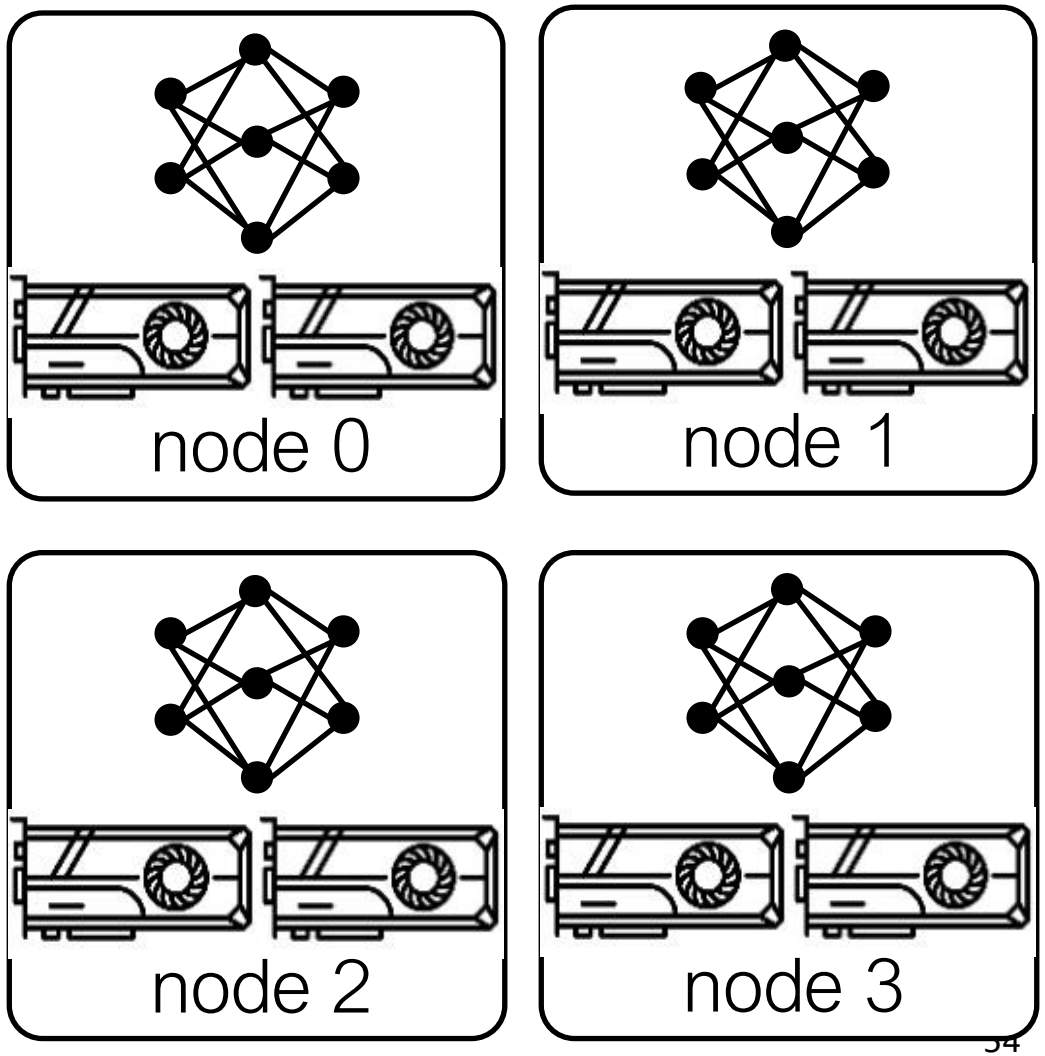
- Parameter Server
  - needs to synchronize twice (parameters and local gradients)
- AllReduce Data Parallel
  - each local worker needs to update parameters
  - redundant?
  - local updating is much faster than transferring data across gpus

Outline

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

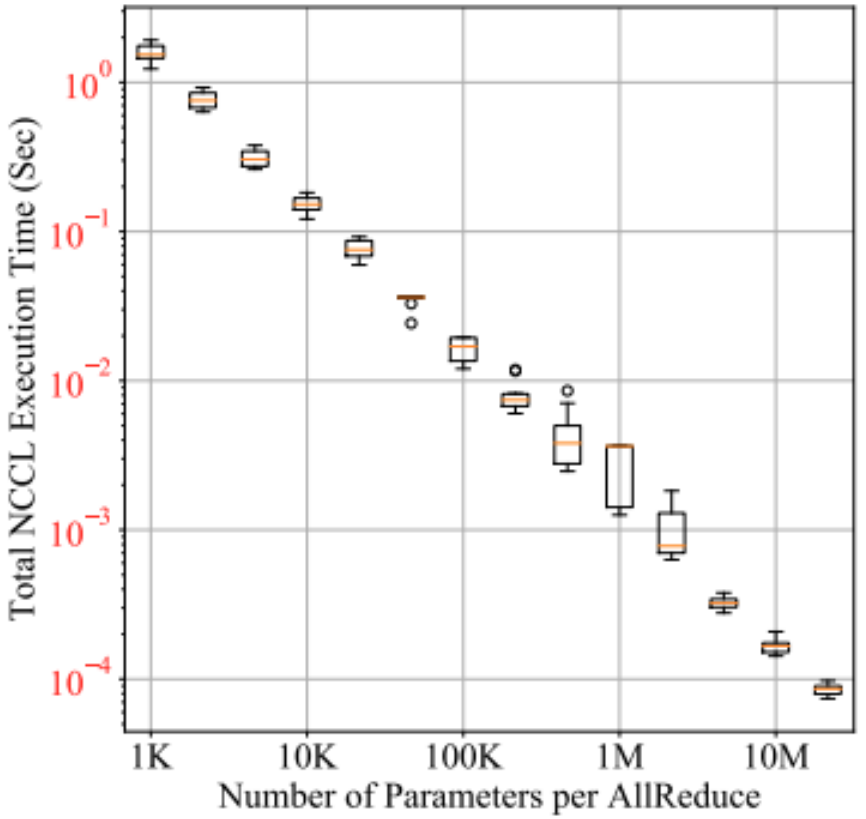
Distributed Data Parallel

- Same as Data Parallel
- multiple nodes, each with multiple GPUs
  - Create replicas of a model on multiple nodes
  - Each model performs the forward pass and the backward pass independently
  - Gather average gradients across nodes
  - Optimizers run locally (identical)



Hiding Communication with Computation in Distributed Data Parallel

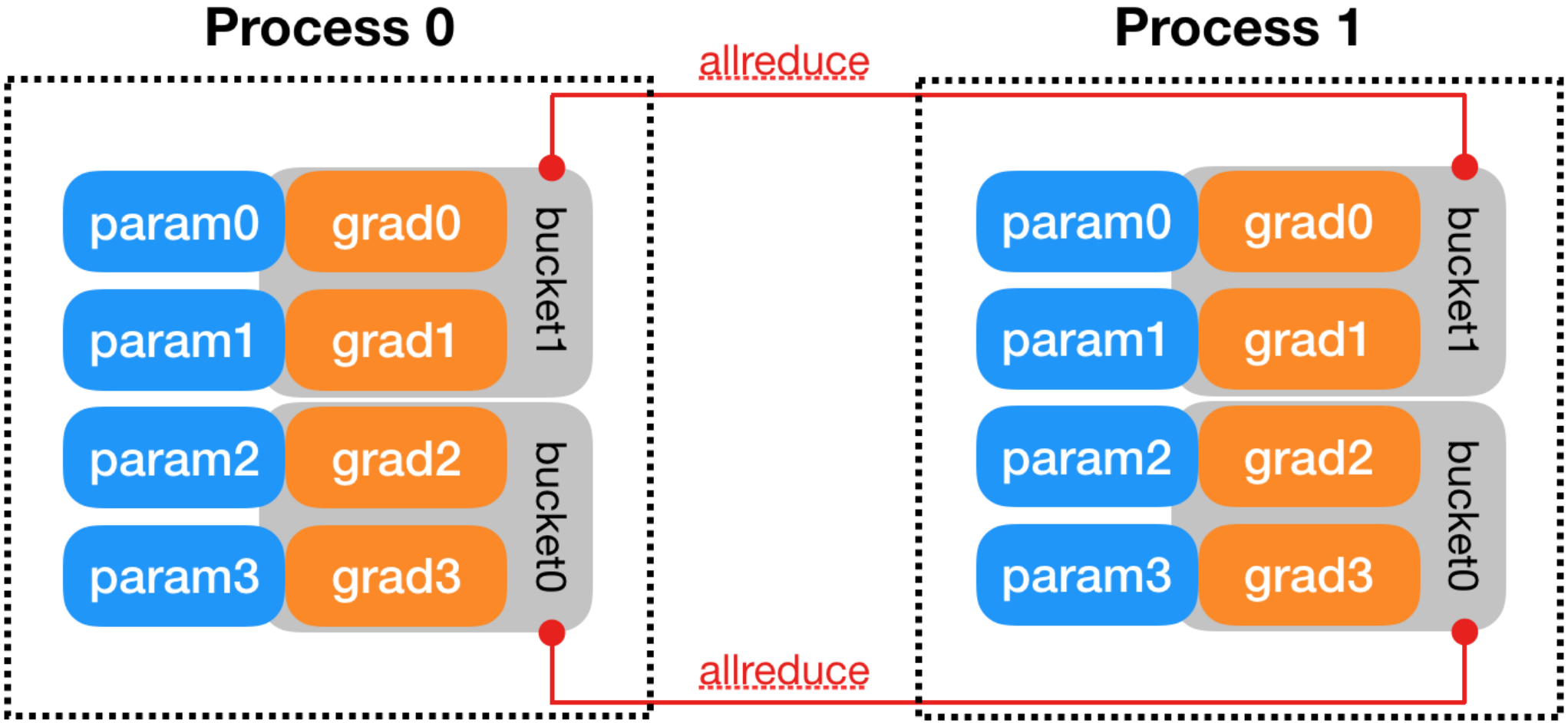
- Naïve solution: synchronize gradients after the *entire* backward pass finishes
  - We can overlap gradient computation and synchronization!
- But how often should we synchronize? Too much synchronization slows down execution
- Key idea: overlapping comm with comp



(a) NCCL

Hiding Comm by Gradient Bucketing

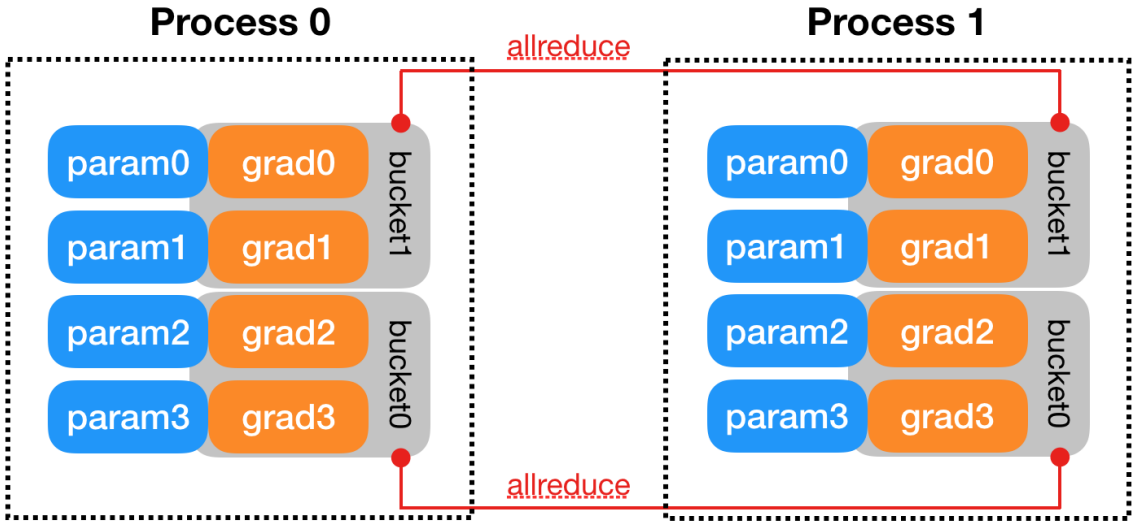
Asynchronously allreduce when a bucket of parameter grads are ready.



Shen Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.

Gradient Bucketing

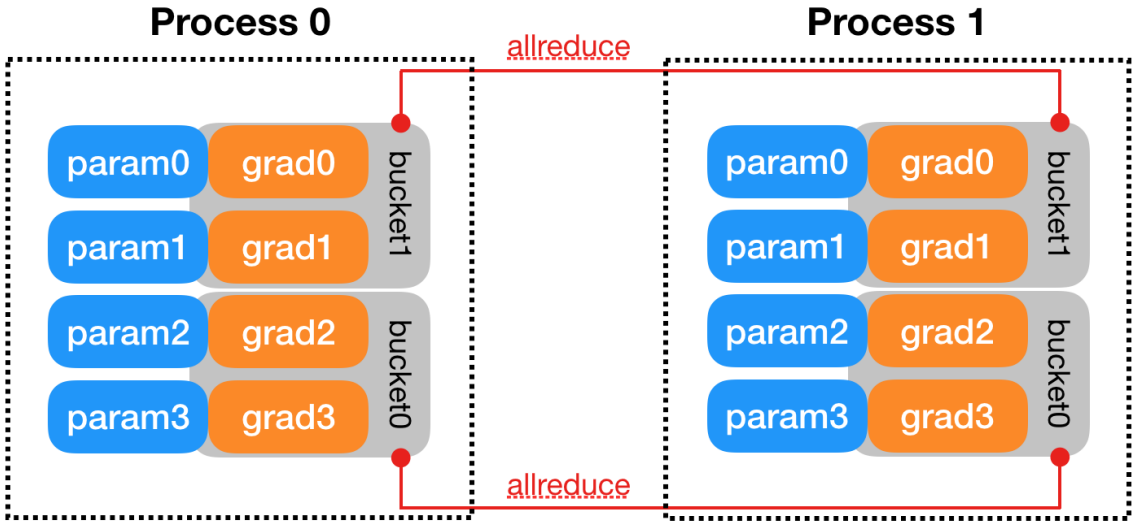
- Bucket size can be configured by setting the **bucket_cap_mb** argument in DDP constructor.
- The mapping from parameter gradients to buckets is determined at the construction time, based on the bucket size limit and parameter sizes.



Shen Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.

Gradient Bucketing

- Model parameters are allocated into buckets in (roughly) the reverse order of Model.parameters() from the given model.
- DDP expects gradients to become ready during the backward pass in approximately that order.

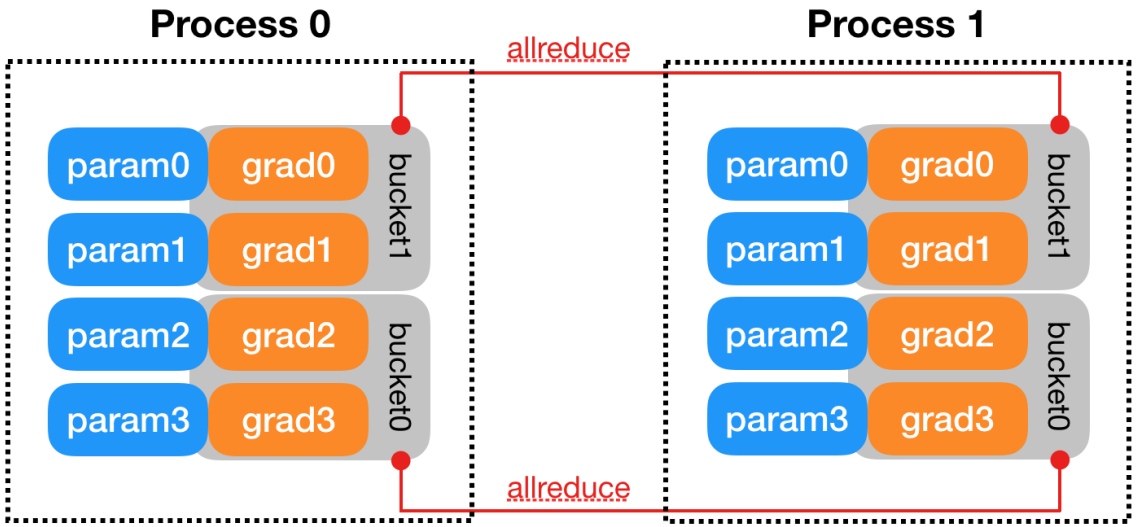


Shen Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.

Gradient Bucketing

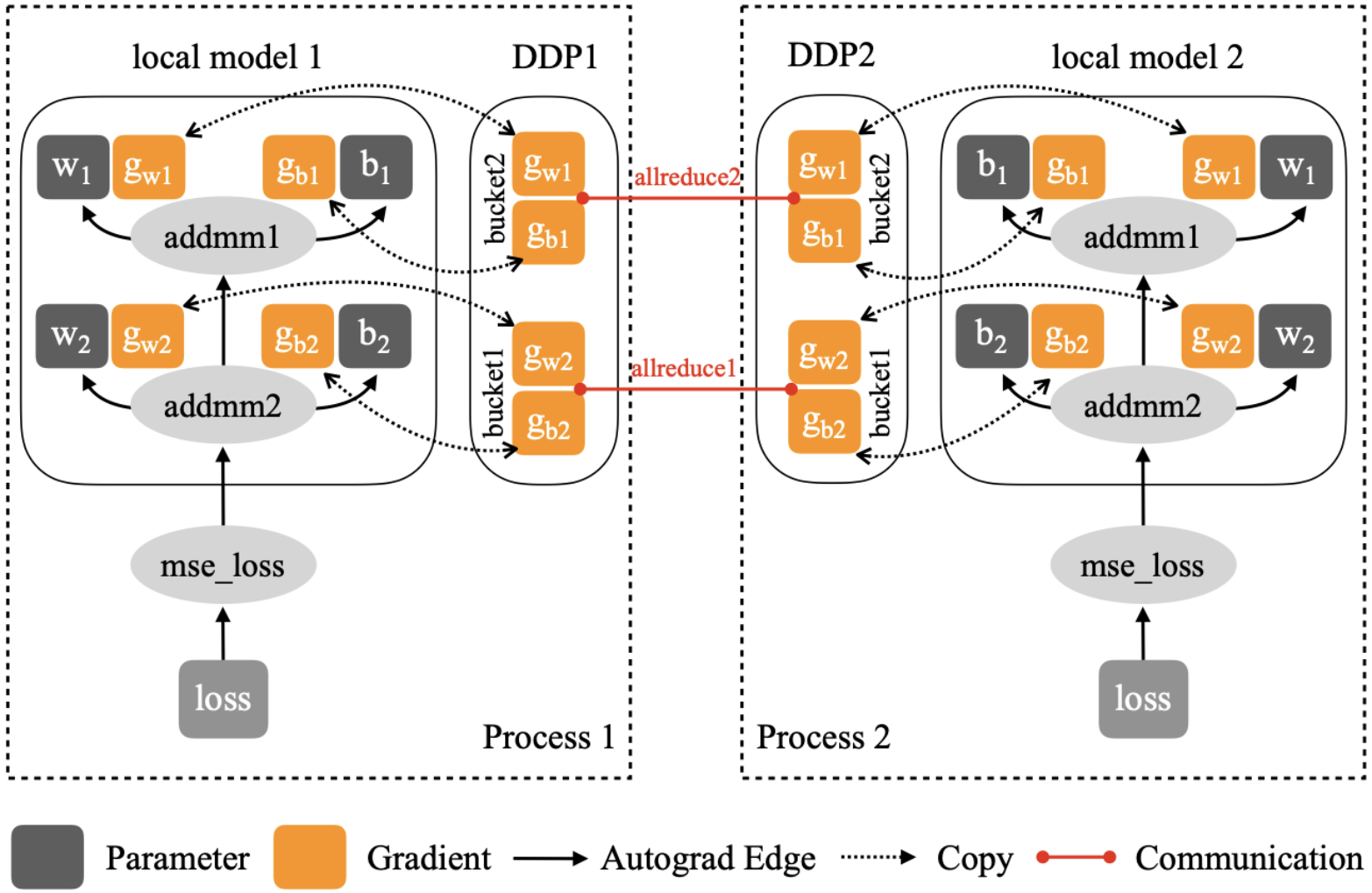
- When gradients in one bucket are all ready, the Reducer kicks off an asynchronous allReduce on that bucket to calculate average of gradients across all processes.

- Overlapping computation (backward) with communication (AllReduce)



Shen Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.

Gradient Reduction



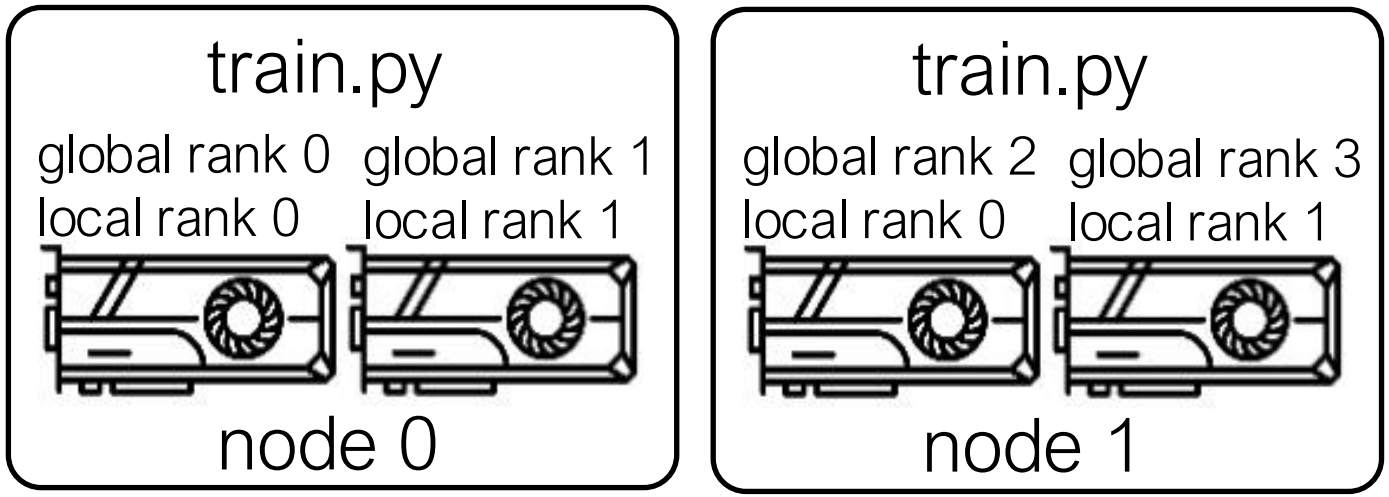
Shen Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. VLDB 2020.

DDP Implementation

```
// The function `autograd_hook` is called after the gradient for a
// model parameter has been accumulated into its gradient tensor.
// This function is only to be called from the autograd thread.
void Reducer::autograd_hook(size_t index) {
        mark_variable_ready(index);
}

void Reducer::mark_variable_ready(size_t variable_index) {
        const auto& bucket_index = variable_locators_[variable_index];
        auto& bucket = buckets_[bucket_index.bucket_index];

        if (--bucket.pending == 0) {
                mark_bucket_ready(bucket_index.bucket_index);
        }
}

void Reducer::mark_bucket_ready(size_t bucket_index) {
        for (; next_bucket_ < buckets_.size() && buckets_[next_bucket_].pending == 0; next_bucket_++) {
                num_buckets_ready_++;
                auto& bucket = buckets_[next_bucket_];
                all_reduce_bucket(bucket);
        }
}

void Reducer::all_reduce_bucket(Bucket& bucket) {
        auto variables_for_bucket = get_variables_for_bucket(next_bucket_, bucket);
        const auto& tensor = bucket.gradients;
        GradBucket grad_bucket(next_bucket_, buckets_.size(), tensor, bucket.offsets,
                bucket.lengths, bucket.sizes_vec, variables_for_bucket);
        bucket.future_work = run_comm_hook(grad_bucket);
}
```

Setting up the Distributed Process

- World size
  - total number of processes W
- Global rank
  - global process id
- Local rank
  - local process id



Launch Distributed Processes

The launch.py (torch/distributed/launch.py) will pass world size, global rank, master address, master port via env vars, and local rank as a commandline parameter to every instance

Env Vars: "MASTER_ADDR", "MASTER_PORT", "RANK", "WORLD_SIZE"

```
if __name__ == "__main__":
  parser = argparse.ArgumentParser()
  parser.add_argument("--local_rank", type=int, default=0)
  parser.add_argument("--local_world_size", type=int, default=1) args =
parser.parse_args()
  local_proc(args.local_world_size, args.local_rank)
```

Launching Local Process

```
def local_proc(local_world_size, local_rank):
  dist.init_process_group(backend="nccl")

  local_train(local_world_size, local_rank)

  dist.destroy_process_group()
```

start process group

tear down process group

```
def demo_basic(local_world_size, local_rank):
  n = torch.cuda.device_count() // local_world_size
  device_ids = list(range(local_rank * n, (local_rank + 1) * n))
  model = MyModel().cuda(device_ids[0])
  ddp_model = DDP(model, device_ids)
  loss_fn = nn.MSELoss()
  optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)
  optimizer.zero_grad()
  outputs = ddp_model(torch.randn(20, 10))
  labels = torch.randn(20, 5).to(device_ids[0])
  loss_fn(outputs, labels).backward()
  optimizer.step()
```

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
  - Ring-reduce

Summary

- Data Parallel Training via All Reduce
  - Efficient Ring AllReduce (ScatterReduce+AllGather)
- Distributed Data Parallel Training
  - gradient bucketing
  - overlay backward and AllReduce communication

Reading for next lecture

- Huang et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. 2018
- Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019
- Narayanan et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, SC 2021