Network Optimizations

Systems for Machine Learning

Chapter 7

https://www.cse.iitb.ac.in/~mythili/sysml/

Network Design for ML Workloads

Cloud vs. ML Workloads

- **Cloud computing** produces smooth and regular traffic

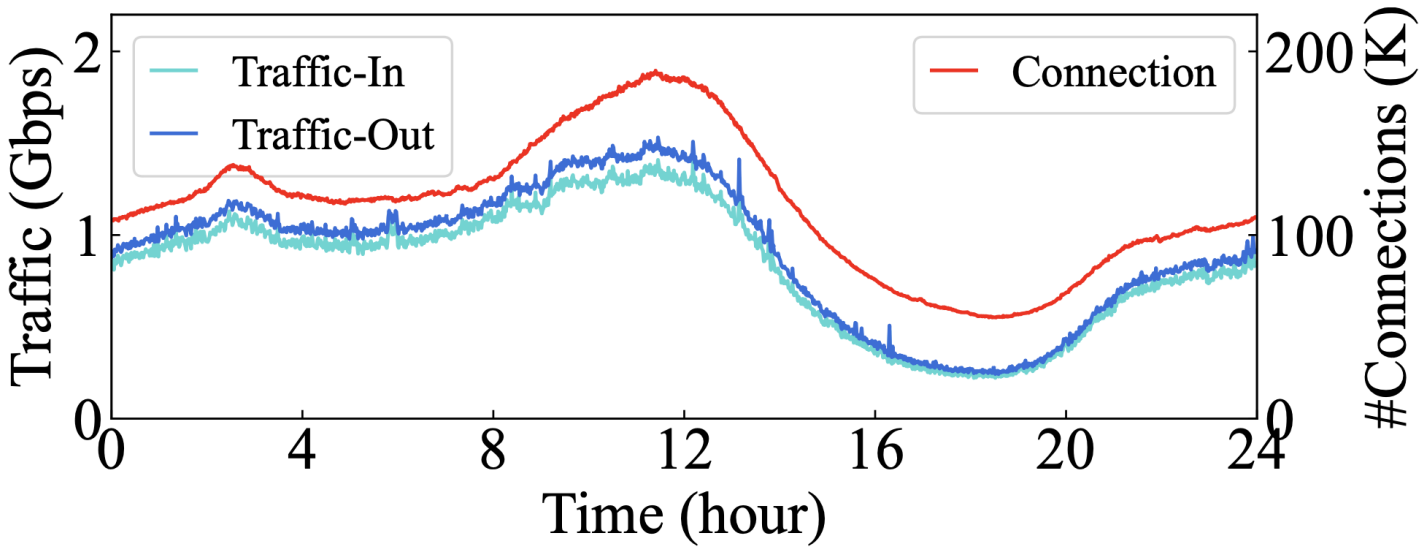


Image Credit: https://doi.org/10.1145/3651890.3672265

Bursty Workloads in ML

- Between sync points in **distributed training**, GPU busy with local computation and sends no traffic
- At sync point, sends and receives large volume of data

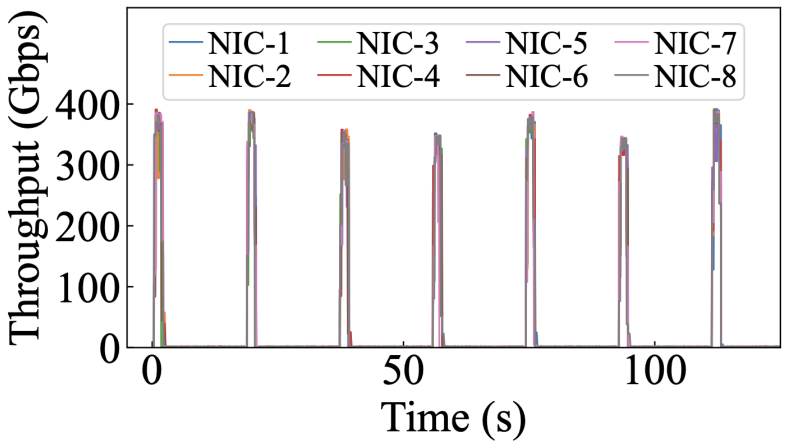


Image Credit: https://doi.org/10.1145/3651890.3672265

Low Entropy and Hash Polarization

- ML traffic has **lower entropy** than traditional traffic
- Traditional computing uses **ECMP** (equal cost multi-path) routing to spread flows evenly
- In a training cluster, number of distinct flows much smaller, harder to spread evenly

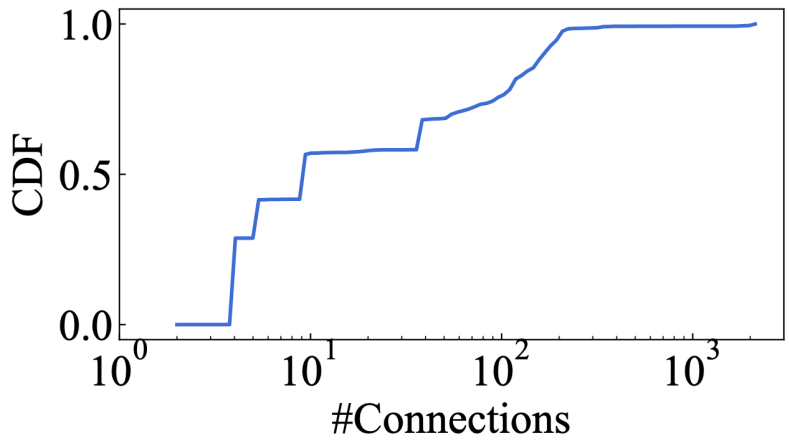
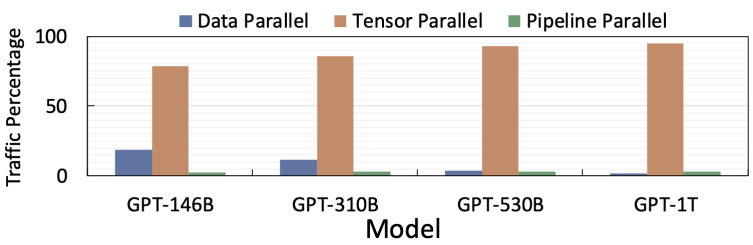


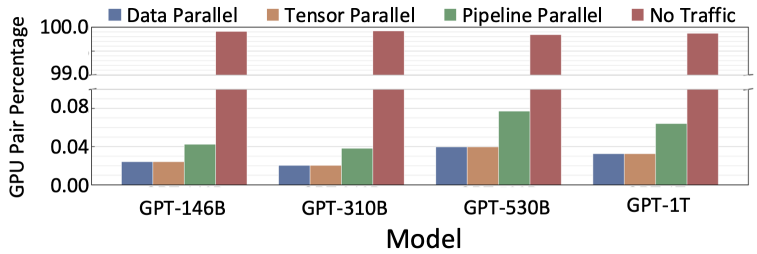
Image Credit: https://doi.org/10.1145/3651890.3672265

Parallelism-Driven Communication Patterns

- **TP** – more frequent communication, latency-sensitive, difficult to overlap with computation
- **DP**, **PP** – less frequent communication, can be overlapped



(a) Unbalanced load

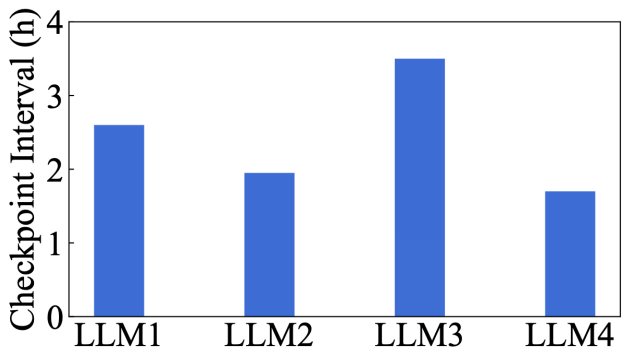


(b) Balanced load

Image Credit:
https://www.computer.org/csdl/proceedings-article/hoti/2024/560100a001/207NRC0kj1m

Failure Sensitivity

- Distributed training clusters are far more sensitive to failure than cloud deployments
- **Checkpointing** large models is expensive
- Link failures are also not rare



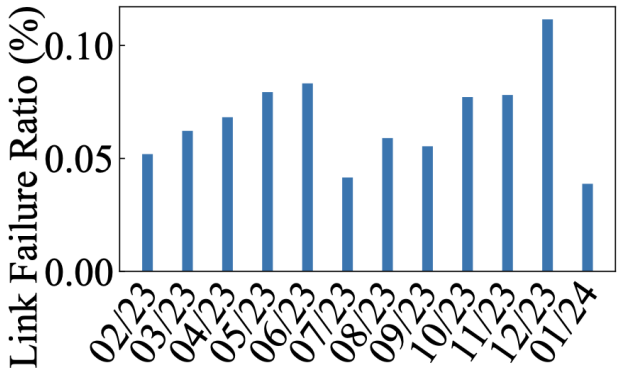


Image Credit: https://doi.org/10.1145/3651890.3672265

Data Center Networks and Interconnects for ML Training

Traditional Data Center Networks

- Top-of-Rack (**ToR**) switch aggregates the traffic of a single rack of machines
- Arranged into **Clos** topology (fat tree), uses ECMP

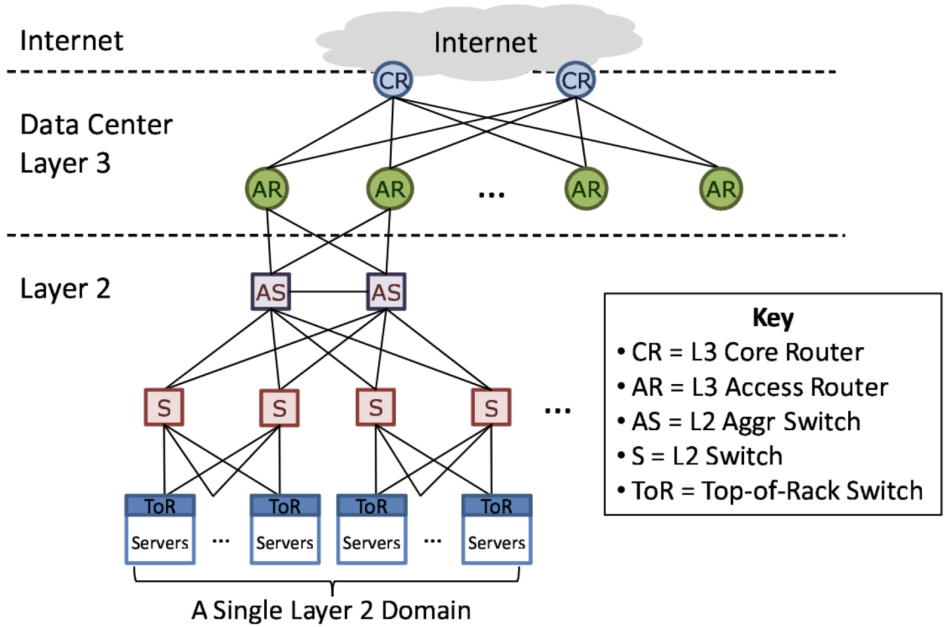


Image Credit: https://doi.org/10.1145/1592568.1592576

Remote Direct Memory Access

- **Traditional network stack** – overhead of copying and CPU cycles used
- **RDMA** – allows NIC to write into/from application buffer directly

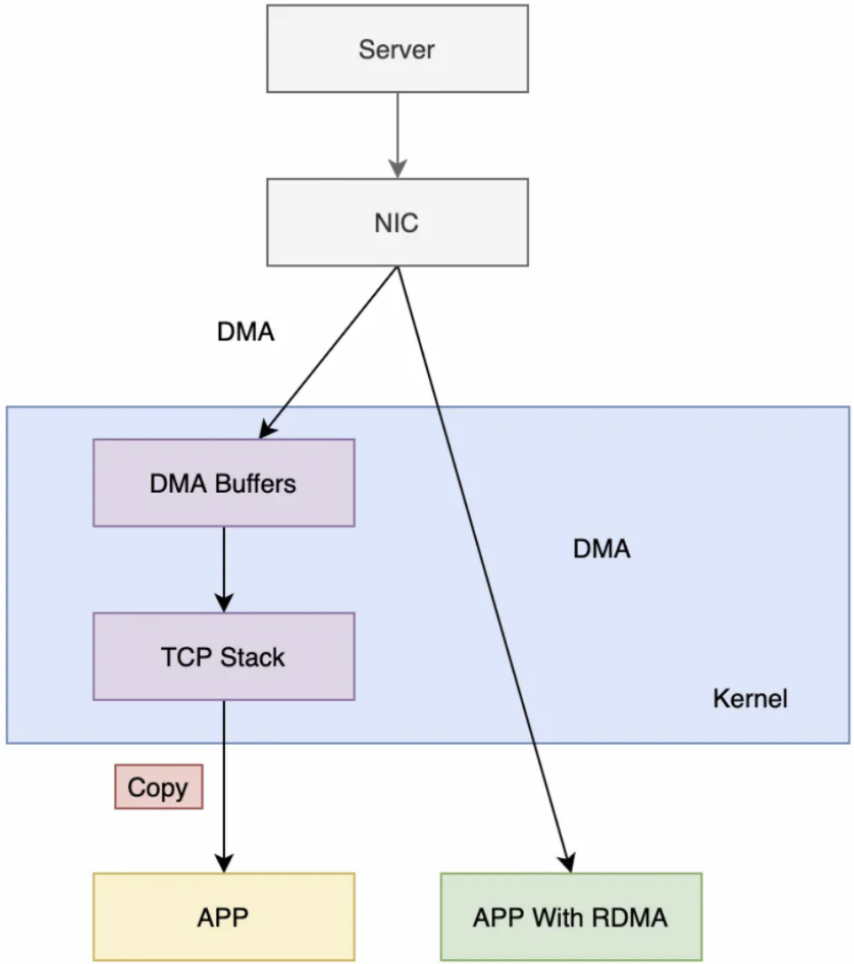


Image Credit: https://www.alibabacloud.com/blog/601598

Queue Pairs, Work Requests, and Completions

- Prepares work queue elements (**WQE**), notified via completion queue
- Transfers can be one-sided or two-sided depending if the receiver node is involved or not

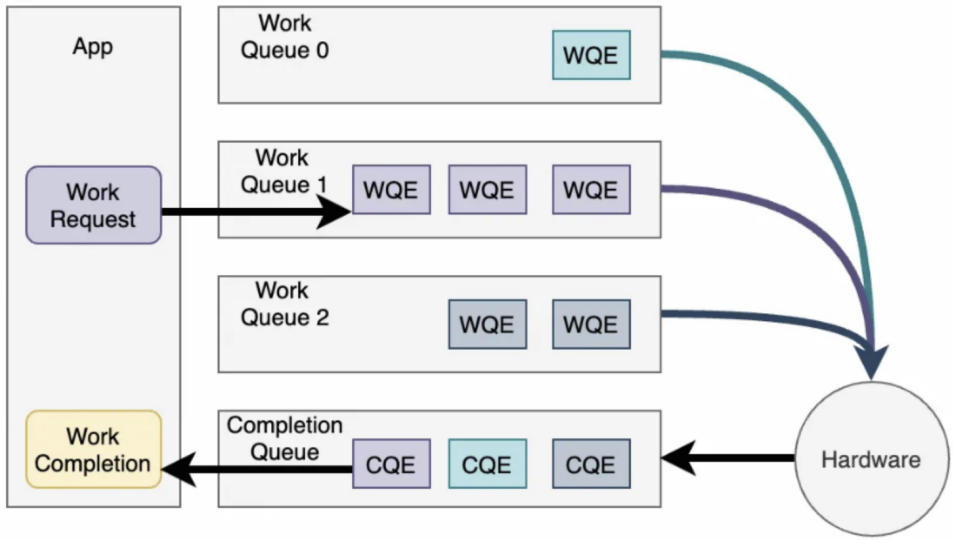


Image Credit: https://www.alibabacloud.com/blog/601598

GPUDirect RDMA

- To avoid copy between GPU memory and host memory
- NIC directly reads from and writes into GPU memory

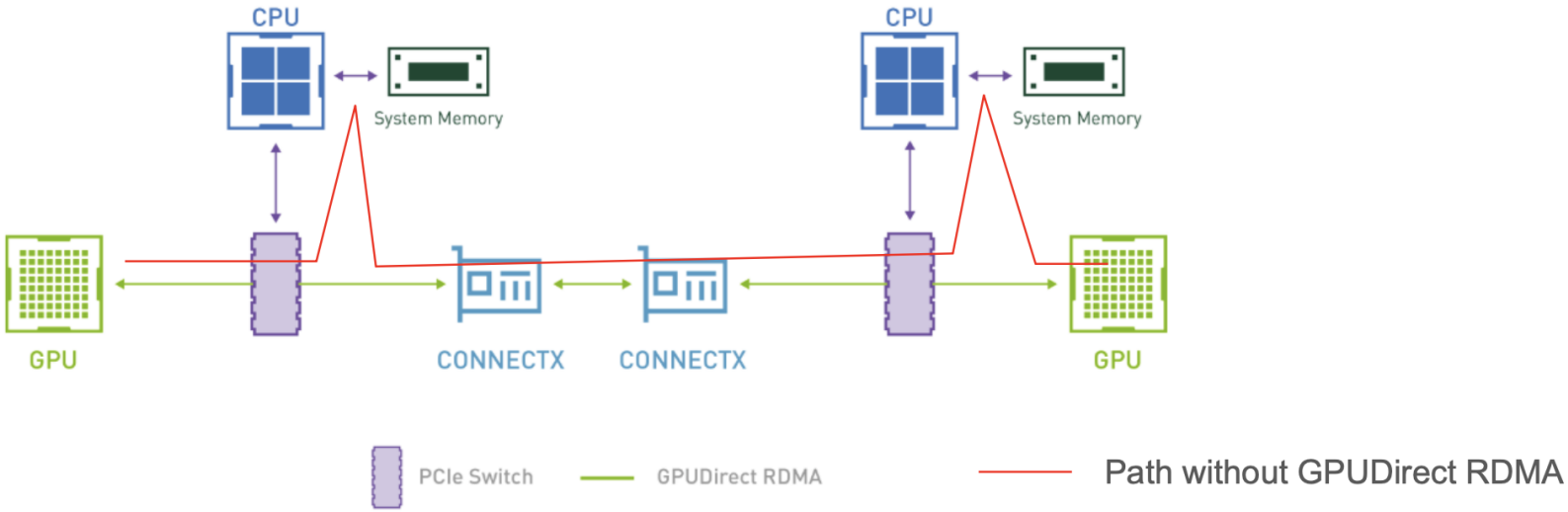


Image Credit: `https://developer.nvidia.com/gpudirect`

Connectivity Inside and Across GPU Nodes

- **NVLink** to connect GPUs within a node
- To connect to rest of the cluster, each GPU given a separate NIC to avoid contention

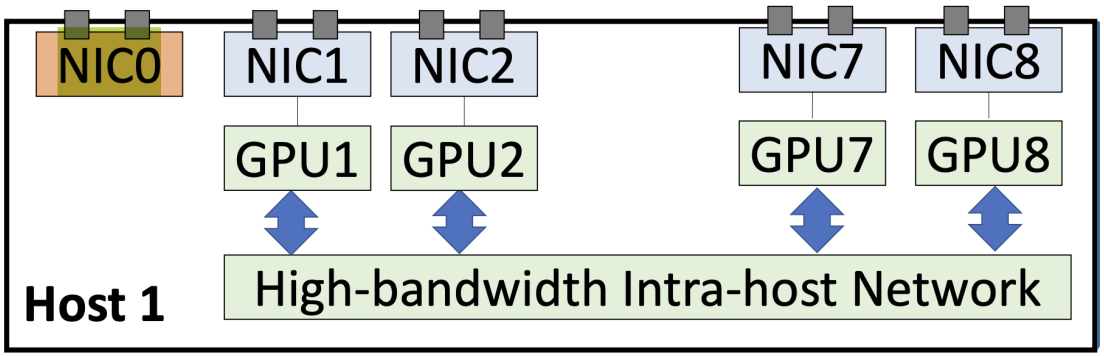


Image Credit: https://doi.org/10.1145/3651890.3672265

Intra-node Transfer

- NVLink preferred; if not available, PCIe bus used

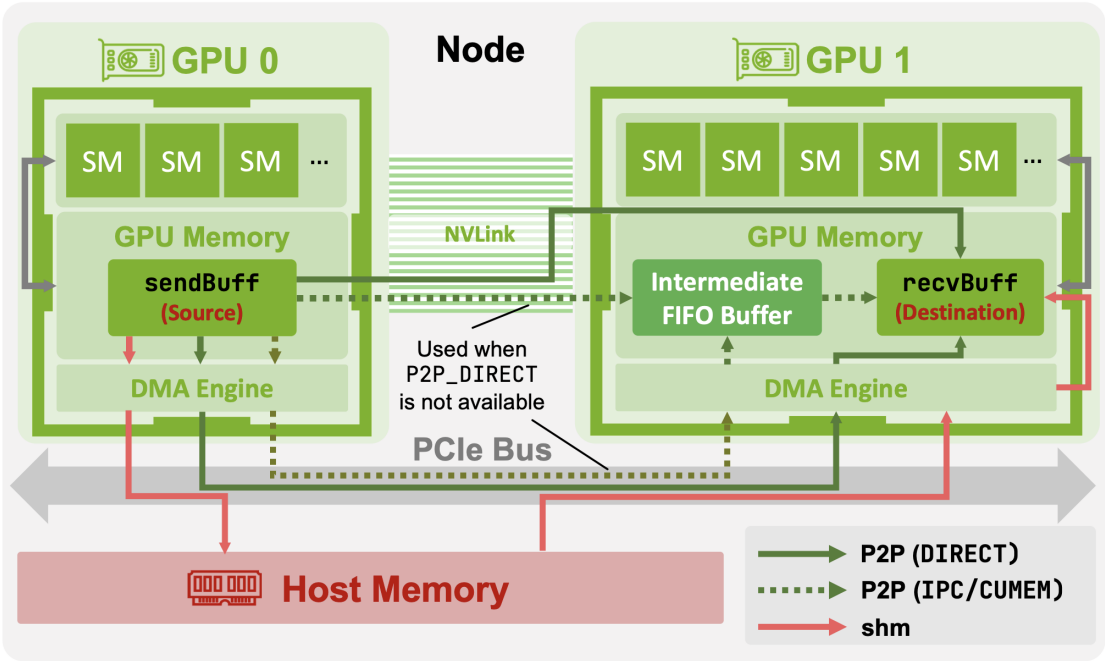


Image Credit: https://arxiv.org/abs/2507.04786

Inter-node Transfer

- If the NIC does not support RDMA, NCCL falls back to TCP socket transport via host memory

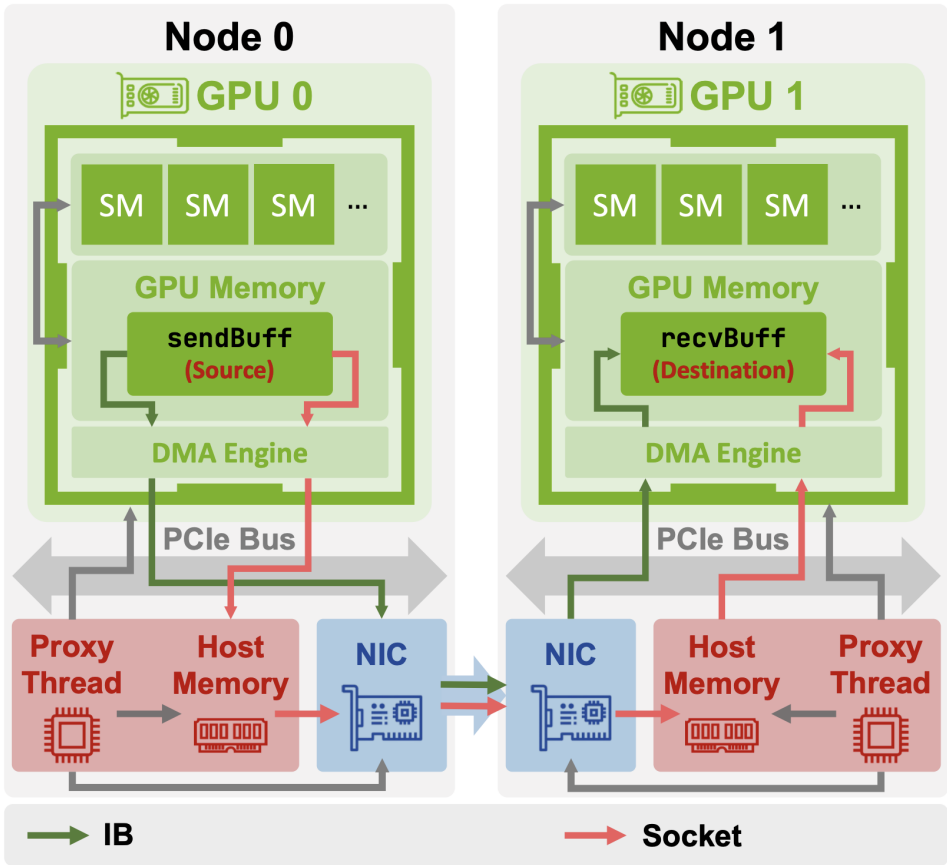


Image Credit: https://arxiv.org/abs/2507.04786

Case Study: Alibaba's HPN

- Real-world data center network designed to support large scale LLM training production workloads
- Hierarchy of ToR, aggregation, core switches

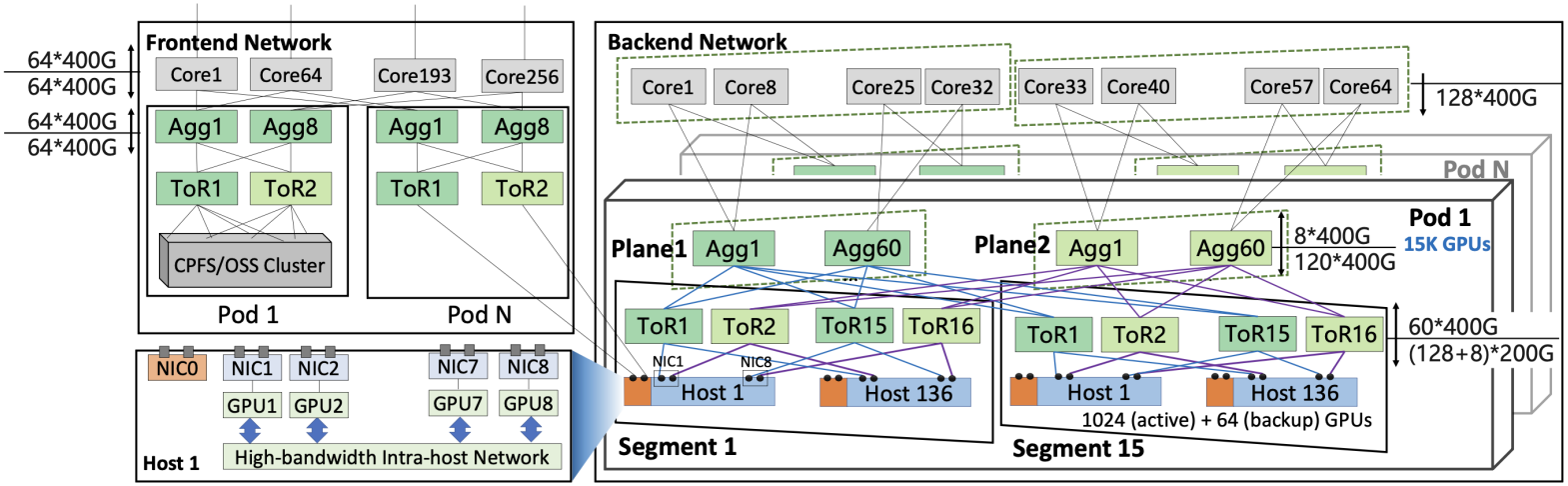


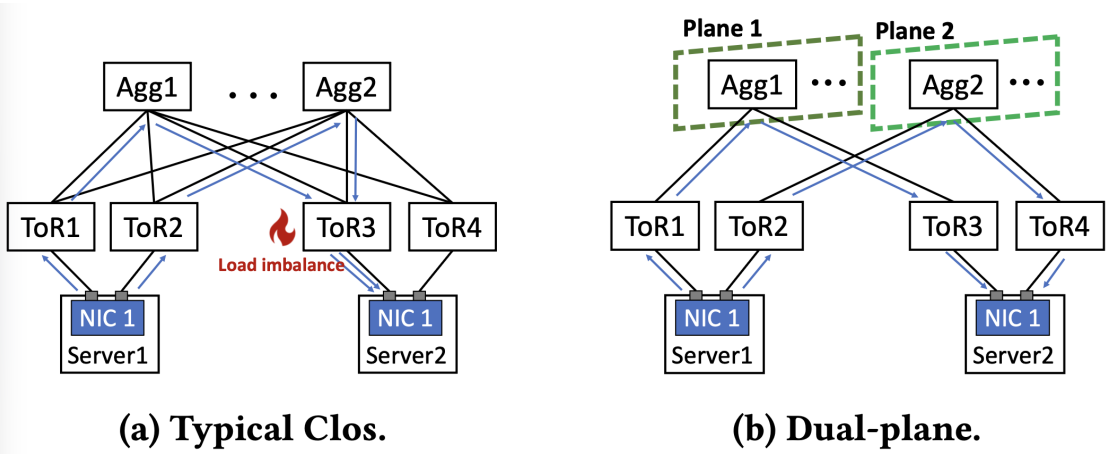
Image Credit: https://doi.org/10.1145/3651890.3672265

Non-stacked Dual-ToR

- A fault on any single link stalls the entire job
- **Dual-ToR** – give every NIC two independent uplinks to two different ToR switches
- **Stacked** – ToR switches are wired and synchronize their forwarding state
  - One down led to other's failure too
- **Non-stacked** – direct link removed, both switches coordinate through other robust means

Dual-Plane Design

- Two ToR switches of every dual-ToR pair assigned to two disjoint groups of aggregation switches (called **planes**)
- Helps in load balancing



(a) Typical Clos. **(b) Dual-plane.**

Image Credit: https://doi.org/10.1145/3651890.3672265

Rail-based Connectivity

- Only a small amount of traffic crosses a node boundary
- **Rails** – all GPUs of the same rank connect to the same ToR switch and can communicate in a single hop

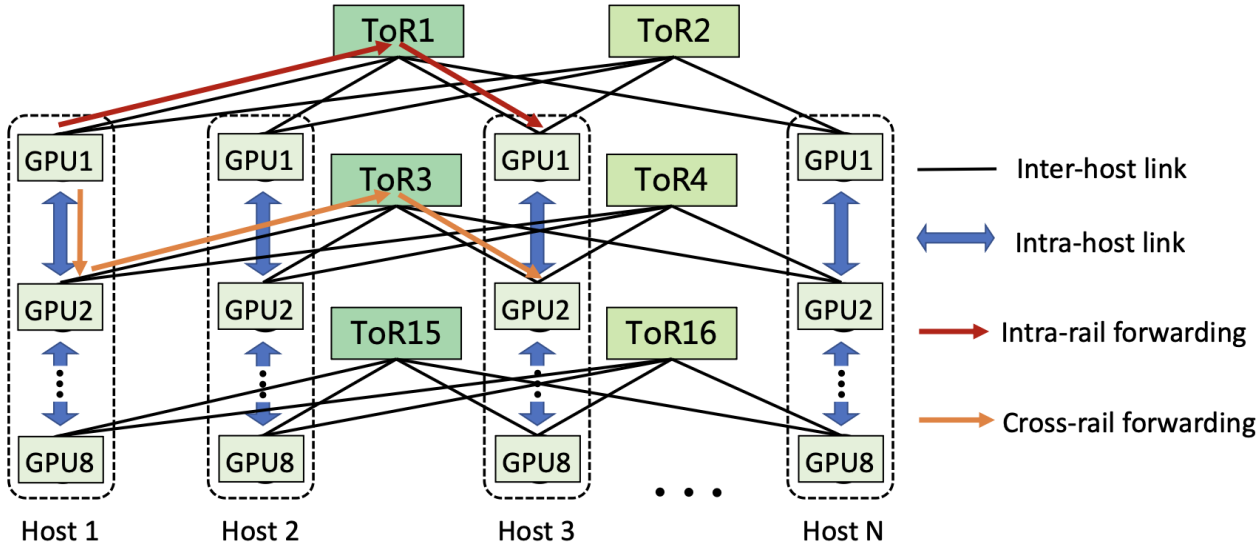
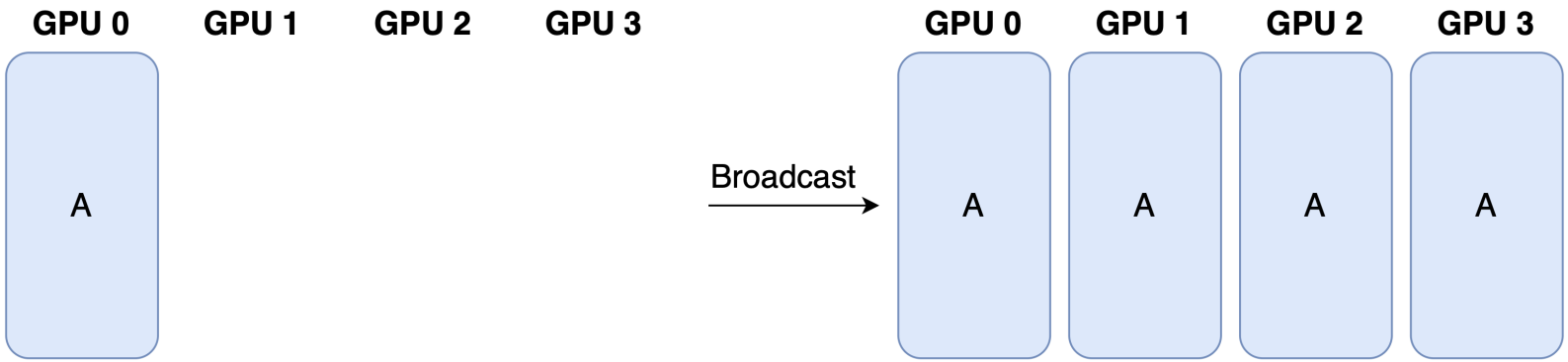


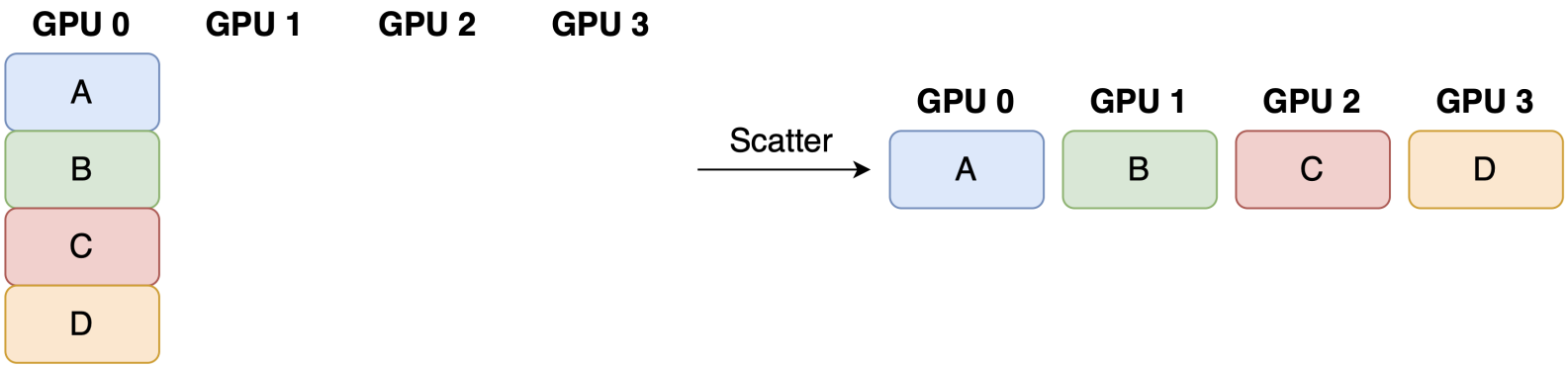
Image Credit: https://doi.org/10.1145/3651890.3672265

Collective Communication

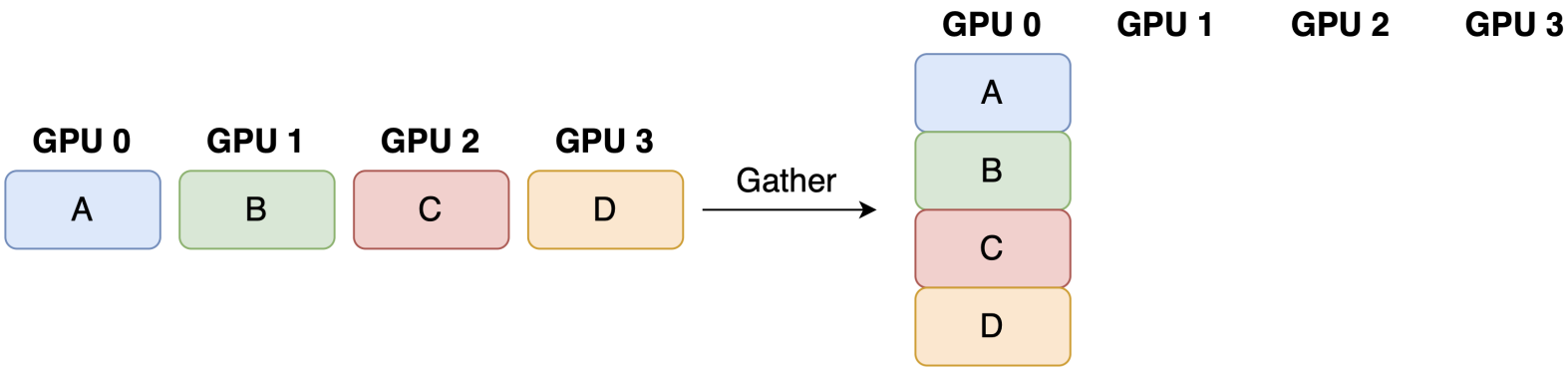
Broadcast



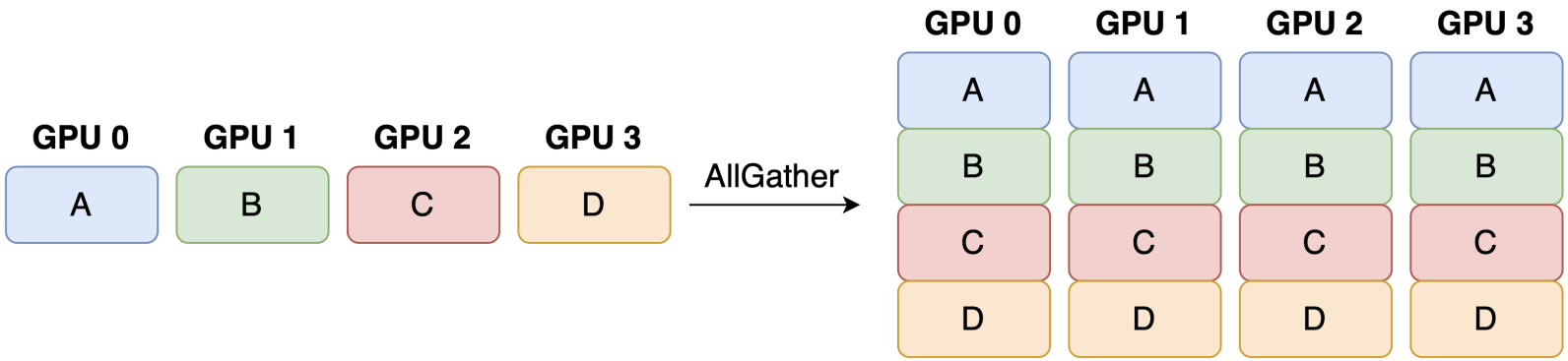
Scatter



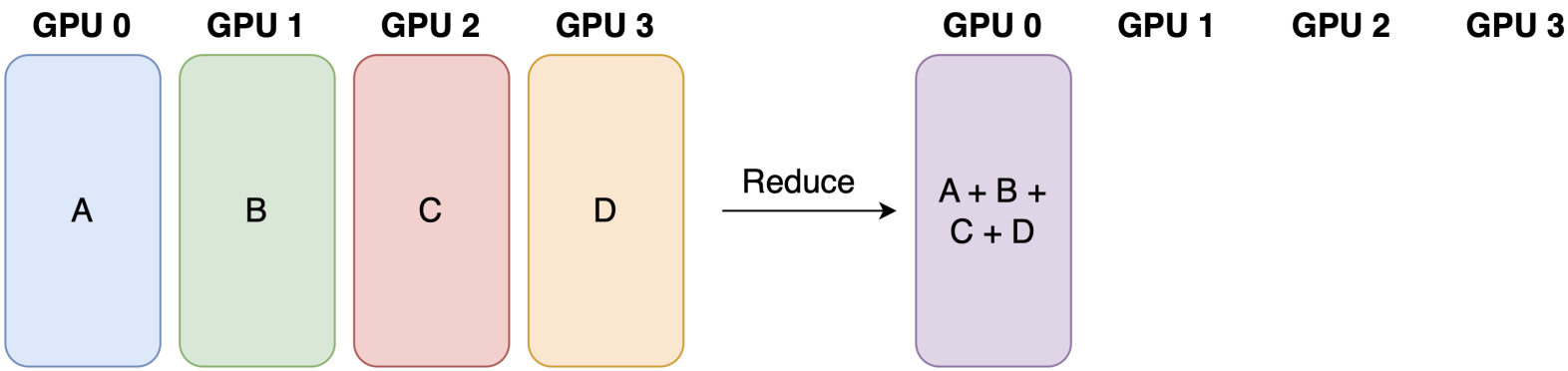
Gather



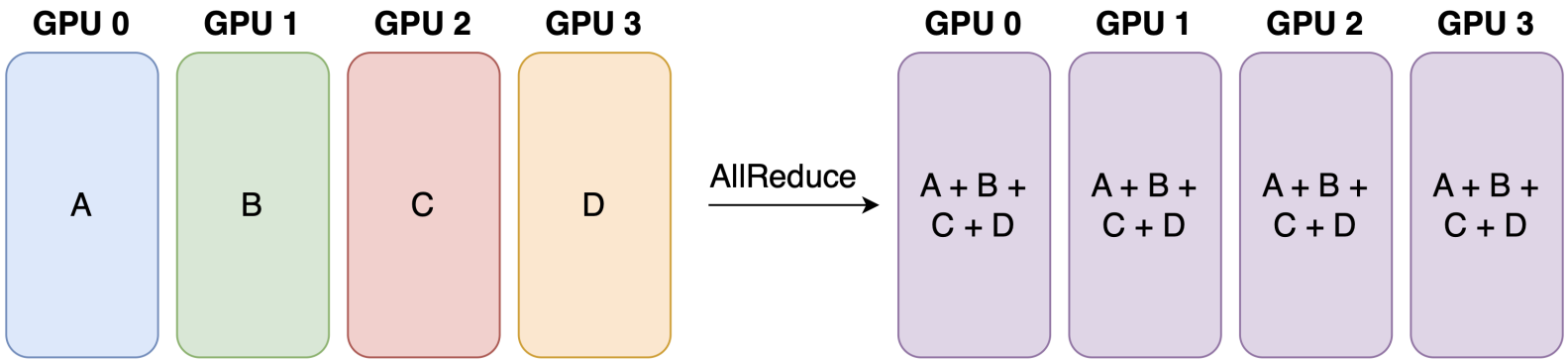
AllGather


GPU 0
GPU 1
GPU 2
GPU 3
A
B
C
D
AllGather
GPU 0
GPU 1
GPU 2
GPU 3
A
A
A
A
B
B
B
B
C
C
C
C
D
D
D
D

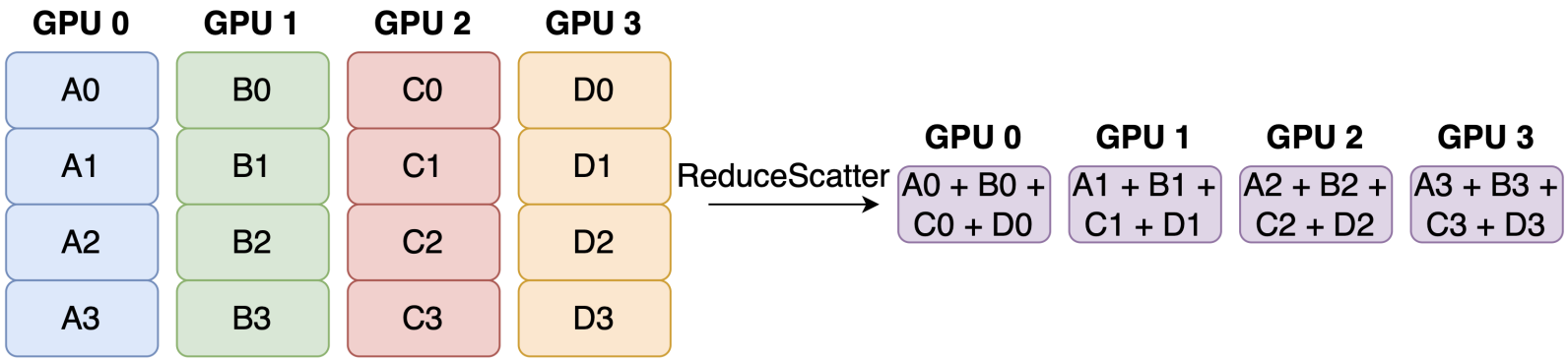
Reduce



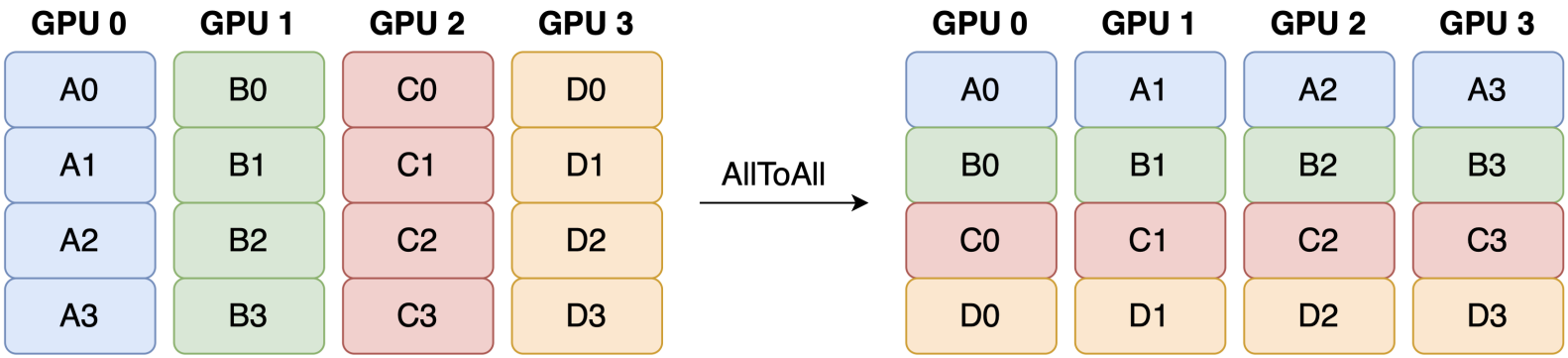
AllReduce


GPU 0
GPU 1
GPU 2
GPU 3
A
B
C
D
AllReduce
GPU 0
GPU 1
GPU 2
GPU 3
A + B + C + D
A + B + C + D
A + B + C + D
A + B + C + D

ReduceScatter



AllToAll


GPU 0
GPU 1
GPU 2
GPU 3
A0
B0
C0
D0
A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3
AllToAll
GPU 0
GPU 1
GPU 2
GPU 3
A0
A1
A2
A3
B0
B1
B2
B3
C0
C1
C2
C3
D0
D1
D2
D3

The NCCL API

Code 1: NCCL Communicator for Single Process

```
1 ncclComm_t comm[4];
2 int devs[4] = {0, 1, 2, 3};
3 ncclCommInitAll(comm, 4, devs);
```

The NCCL API

Code 2: NCCL Communicator for Multiple Processes

```
1  ncclUniqueId id;
2  if (myRank == 0) ncclGetUniqueId(&id);
3  // broadcast id to all other ranks
4
5  ncclComm_t comm;
6  cudaSetDevice(myLocalGpu);
7  ncclCommInitRank(&comm, nRanks, id,
       myRank);
```

The NCCL API

Code 3: Issuing a Collective

```
1  double *d_send, *d_recv;
2  cudaMalloc(&d_send, N * sizeof(double));
3  cudaMalloc(&d_recv, N * sizeof(double));
4  // fill d_send with this rank's local
       data
5
6  ncclAllReduce(d_send, d_recv, N,
       ncclDouble, ncclSum,
7                  comm, stream);
```

The NCCL API

Code 4: Issuing Other Primitives

```
1 ncclBroadcast(d_send, d_recv, N,
      ncclFloat, root, comm, stream);
2 ncclReduce(d_send, d_recv, N, ncclFloat,
      ncclSum, root, comm, stream);
3 ncclAllGather(d_send, d_recv, sendCount,
      ncclFloat, comm, stream);
4 ncclReduceScatter(d_send, d_recv,
      recvCount, ncclFloat, ncclSum, comm,
      stream);
```

The NCCL API

Code 5: Grouping Calls

```
1  ncclGroupStart();
2  for (int g = 0; g < nLocalGpus; g++) {
3      cudaSetDevice(g);
4      ncclAllReduce(d_send[g], d_recv[g],
           N, ncclFloat, ncclSum,
5                        comm[g], stream[g]);
6  }
7  ncclGroupEnd();
```

The NCCL API

Code 6: Send and Receive

```
1 ncclSend(d_send_chunk, chunkSize,
      ncclFloat, peer, comm, stream);
2 ncclRecv(d_recv_chunk, chunkSize,
      ncclFloat, peer, comm, stream);
```

Code 7: Destroying a Communicator

```
1 ncclCommDestroy(comm);
```

Challenges in Implementing Collectives Efficiently

- Which GPUs should communicate with which other GPUs?
- How much data to transfer in each individual step?
  - Trade-offs with small and large message sizes
- How to ensure transfers are overlapped with computation?

Communication Topologies

- **Ring** and **tree** topologies typically used
- Choice depends on message size, number of GPUs participating, interconnect characteristics

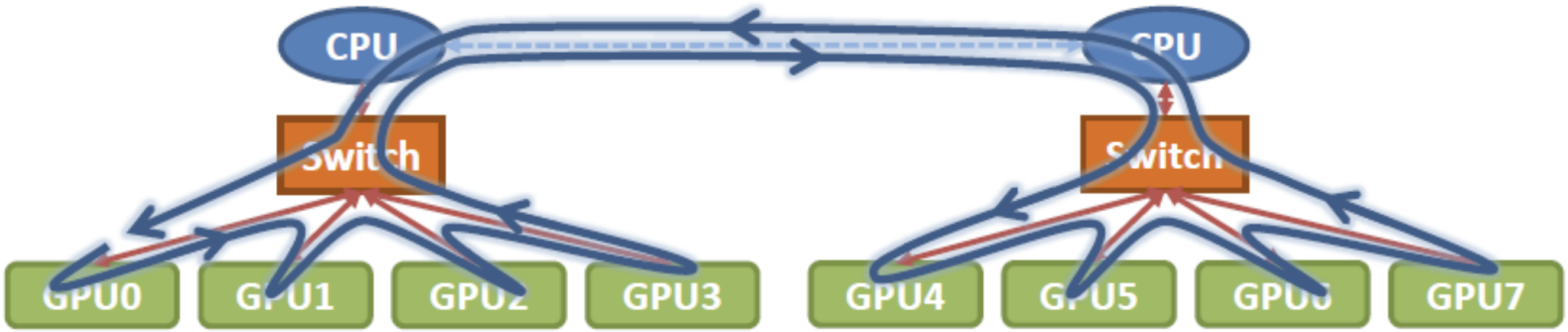


Image Credit: https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf

AllReduce over a Ring

- Composed of ReduceScatter and AllGather
- K GPUs, N buffer size, $2(K-1)$ steps, $2(K-1) \cdot N/K$ elements sent by each GPU

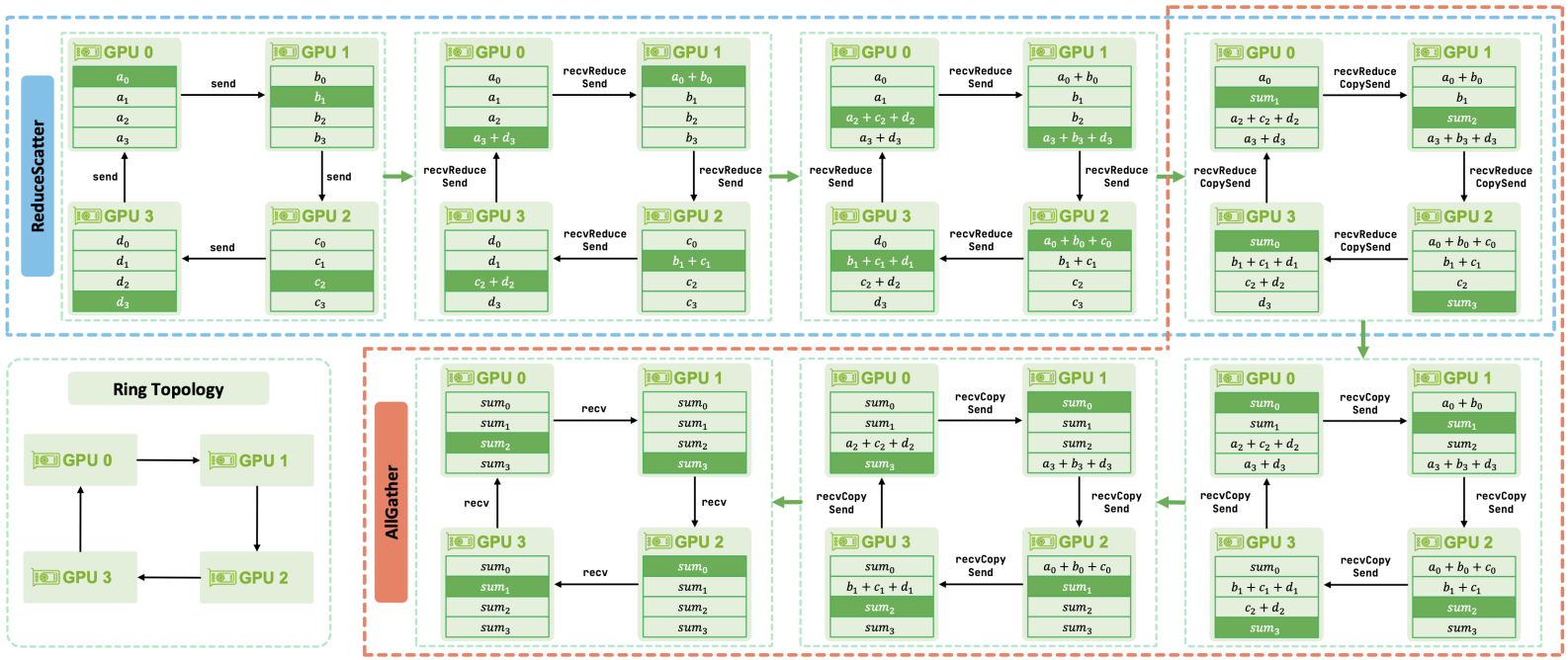


Image Credit: https://arxiv.org/abs/2507.04786

AllReduce over a Tree

- Composed of Reduce and Broadcast
- $\log K$ steps, but internal nodes are more loaded than leaf nodes

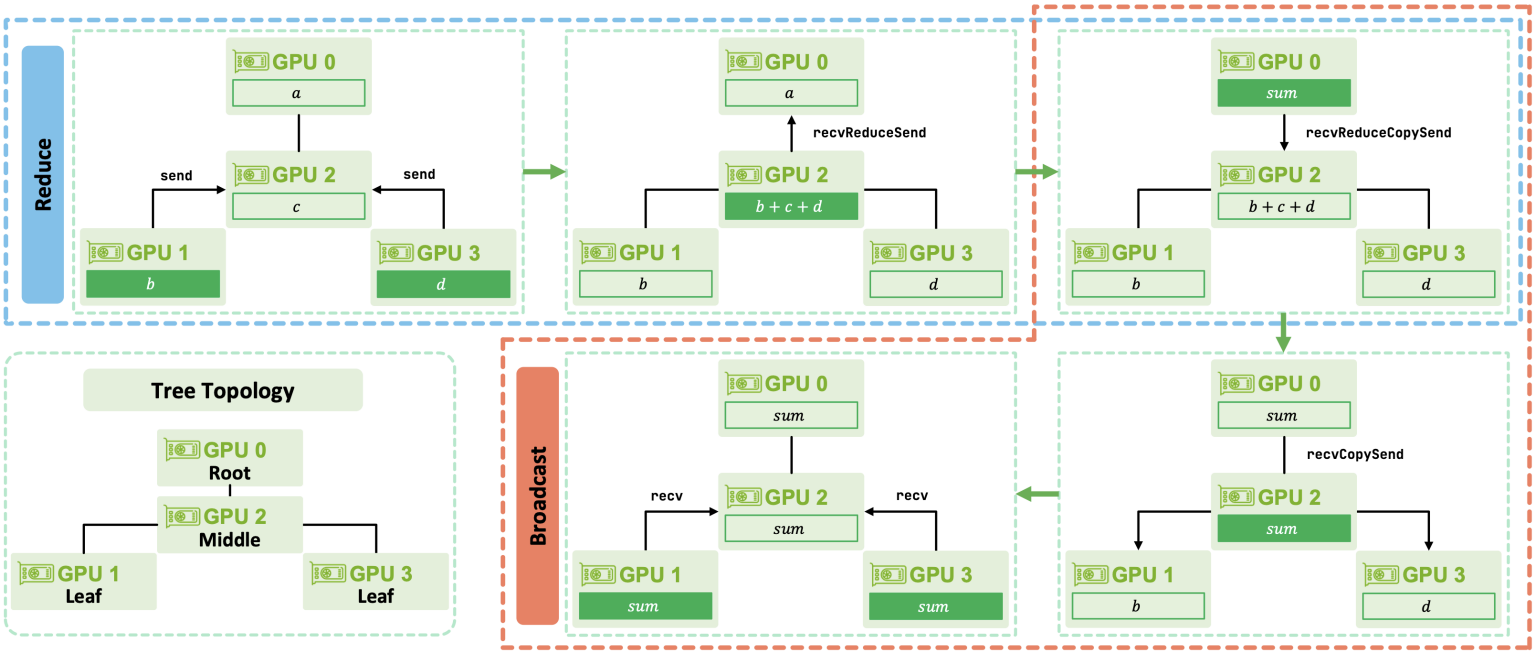


Image Credit: https://arxiv.org/abs/2507.04786

AllReduce on Multiport Direct-Connect Networks

- Ring, tree assume that GPU talks to one peer at a time
- Many clusters have multiple ports per GPU
- For AllReduce on K nodes:
  - With one port, a node can receive as much as it has, so information doubles in each step, log$K$ steps
  - A node with $2D$ ports can multiply its coverage by $2D+1$ per step
  - Most existing algorithms just use one port

Tripling the Distance in Both Directions

- **Trivance** – in step k, every node sends to two nodes at distance 3^{k-1}, one on each side
- After k steps, a node holds the sum over a group of 3^k consecutive nodes centered on itself

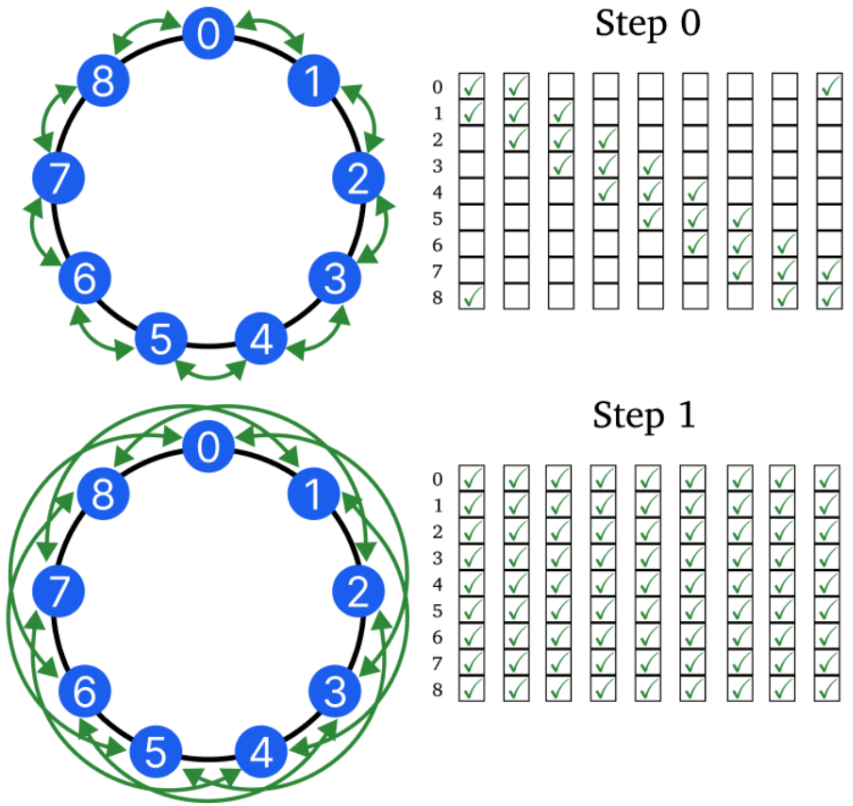


Image Credit: https://doi.org/10.1145/3789240.3829181

Transport and Communication Protocols

RDMA Transport and Communication Protocols

- **GPU** moves and reduces data, **NIC** moves bytes over the network, **CPU** coordinates this transfer
- Inside a server:
  - **P2P** – direct GPU-to-GPU access over NVLink/PCIe
  - **SHM** – stages data through shared host memory
  - **NVLS** – uses the multicast and reduction features of NVLink switches
- Between servers:
  - **NET** – uses InfiniBand, RoCE, or ordinary TCP sockets
  - **COLLNET** – uses switches that can perform part of the collective in the network

Reliable and Unreliable Modes

	Reliable Connection (RC)	**Unreliable Connection (UC)**	**Unreliable Datagram (UD)**
Reliability and retransmission in NIC	yes	no	no
Congestion control in NIC	yes	no	no
Message size	large messages	large messages	one packet
NIC splits message into packets and reassembles	yes	yes	no
Preferred use for a software transport	second choice	first choice	last resort

Congestion and Flow Control

- **Congestion control** makes senders slow down when network receives data faster than it can send
- Signals – ECN marks, RTT measurements, packet drops
- **DCQCN** used in RDMA – congested switches mark packets; receiver warns sender, who slows down
- **Flow control** stops sender when receiver can't take more, e.g., Priority Flow Control (**PFC**) sends a pause message to the device that feeds it
- Congestion and flow control in hardware in RDMA

Protocols: Simple, LL, and LL128

	Simple	**LL**	**LL128**
Design goal	high bandwidth	low latency	low latency and high bandwidth
Synchronization	memory fences	flag-based	flag-based
Payload	data chunks	4 B data + 4 B flag	120 B data + 8 B flag
Bandwidth used	near peak	25–50% of peak	about 95% of peak
Latency per hop	$\sim 6\,\mu$s	$\sim 1\,\mu$s	$\sim 2\,\mu$s

Channels

- NCCL splits a large message into **channels** (parts) and sends them in parallel to fill a fast link

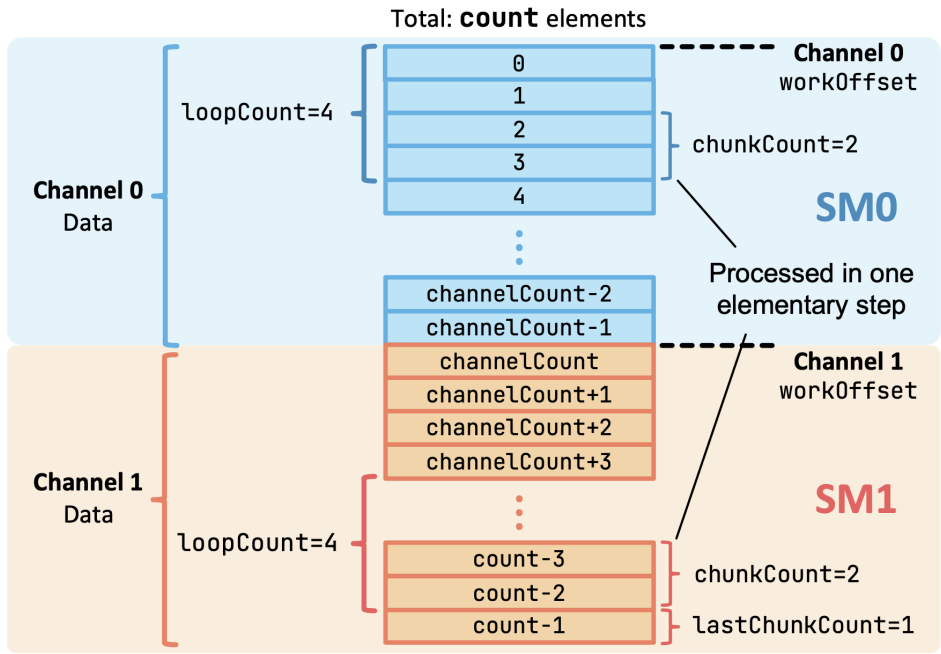
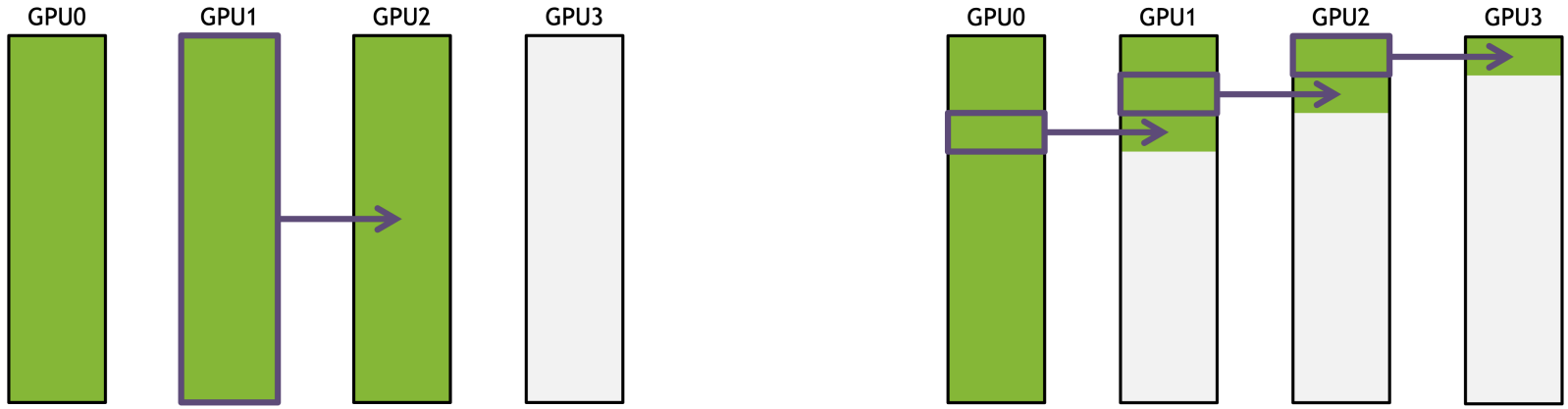


Image Credit: https://arxiv.org/abs/2507.04786

Pipelining with Slots

- **Pipelining** – don't wait for the whole message, split into small pieces, pass each piece as soon as it is received
- K GPUs, B link bandwidth, N message bytes, BROADCAST across ring
- $T_{\text{no pipeline}} = (K-1)\frac{N}{B}$
- Split the message into S equal pieces (slots), GPU forwards a slot to the next GPU immediately after receiving it
- $t_{\text{step}} = \frac{N}{SB}$; $T_{\text{pipeline}} = (S+k-2)\frac{N}{SB} = \frac{N}{B}\left(1+\frac{k-2}{S}\right)$

Pipelining with Slots



(a) Without slots

(b) With slots

Image Credit: `https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf`

Software-Based Transport

- **Hardware transport** – queue fills within microseconds, sender receives delayed warning causing overflows
- DCQCN assumes many small flows; ML has few huge, synchronized ones
- Incast overwhelms the last link; sender-driven control reacts too late
- Go-back-N and PFC: costly retransmits, no multipath
- Hardware changes take years; software transport adapts in weeks

Tolerance of ML to Losses

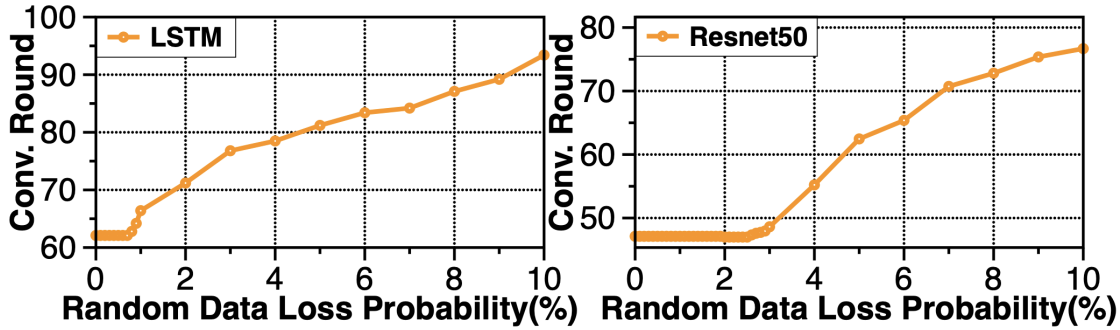
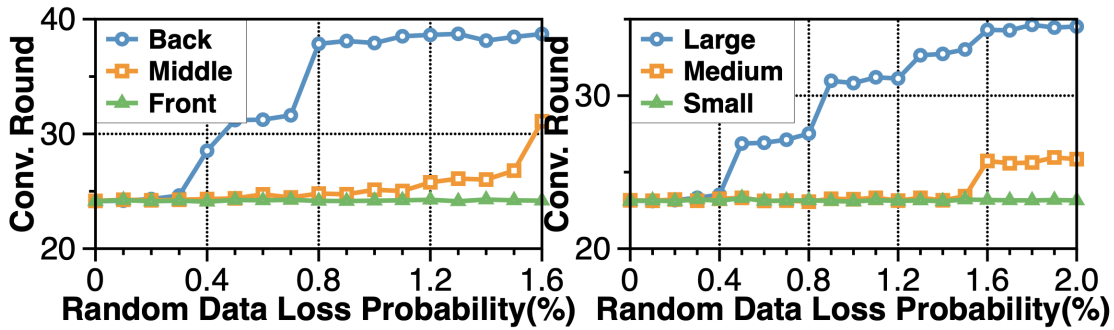


Figure: Impact of loss on model convergence



(a) Loss on different layers (b) Loss with different values

Figure: Gradient losses on different NN layers

Image Credit: https://www.usenix.org/conference/nsdi24/presentation/wang-hao

Tensor Packing

- Gradients are packed into packets together with a tensor id, a layer, and an offset
- Lost gradient packet filled with zeroes, lost parameter packet filled with old values

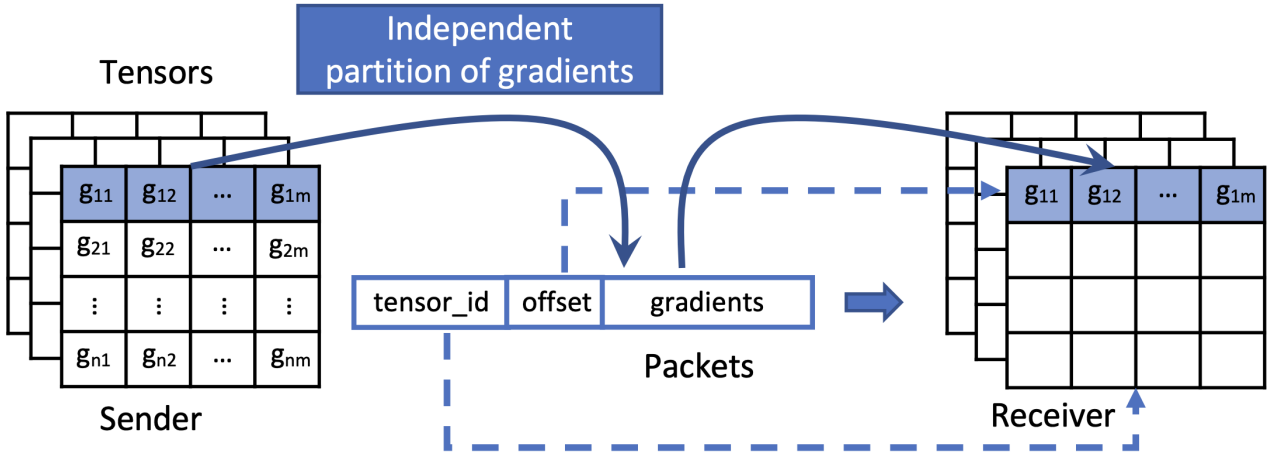


Image Credit: `https://www.usenix.org/conference/nsdi24/presentation/wang-hao`

Priority Queueing and Selective Dropping

- Each packet gets a priority based on its layer and magnitude and switches have several priority queues

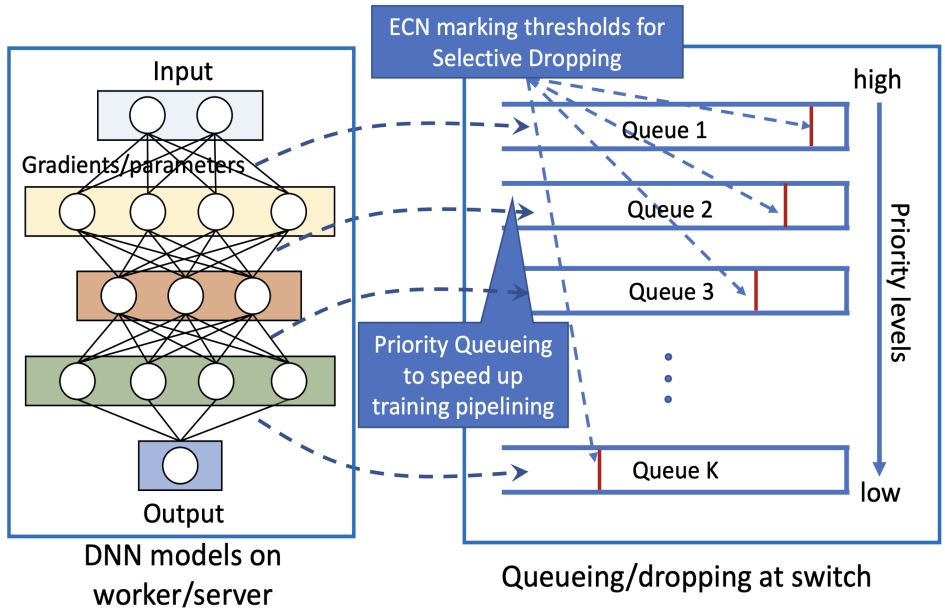


Image Credit: https://www.usenix.org/conference/nsdi24/presentation/wang-hao

UCCL-Tran

- Data path in NIC, control path in CPU, use GPUDirect

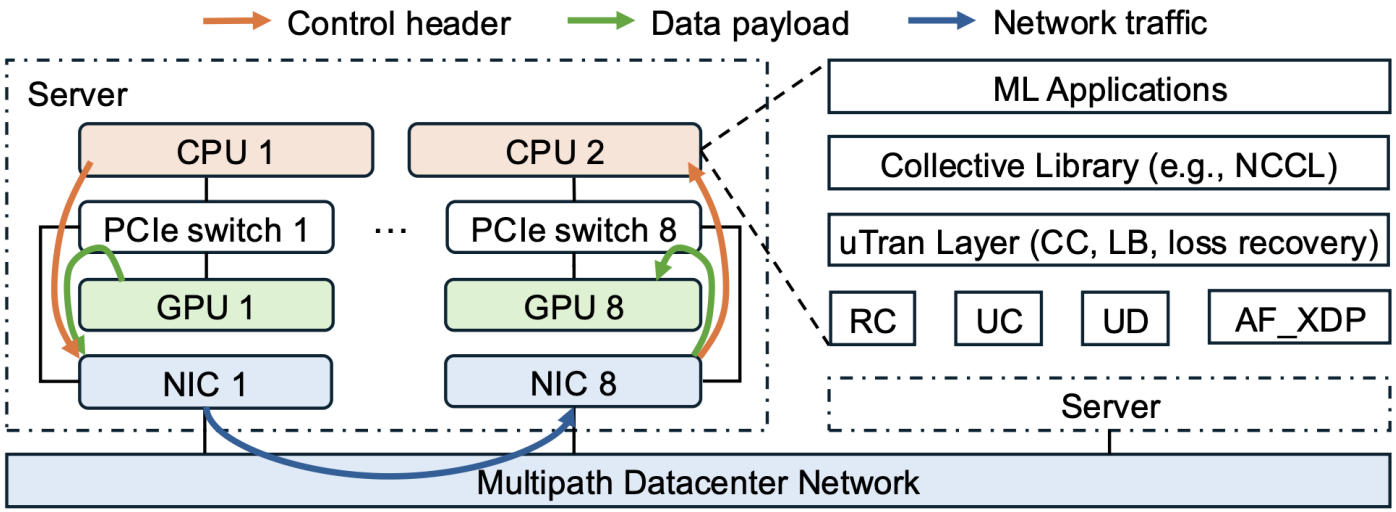


Image Credit: https://www.usenix.org/conference/osdi26/presentation/zhou-yang

UCCL-Tran

- Payload in GPU memory, control information in CPU memory
- UC mode preferred over RC and UD

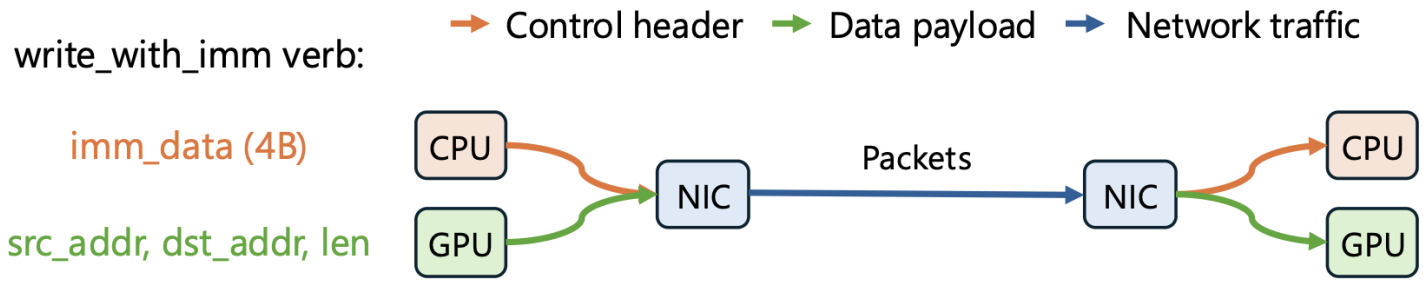


Image Credit: https://www.usenix.org/conference/osdi26/presentation/zhou-yang

Reordering Buffers

- Easier with UC and RC
- With UD, reordering and reassembly needed

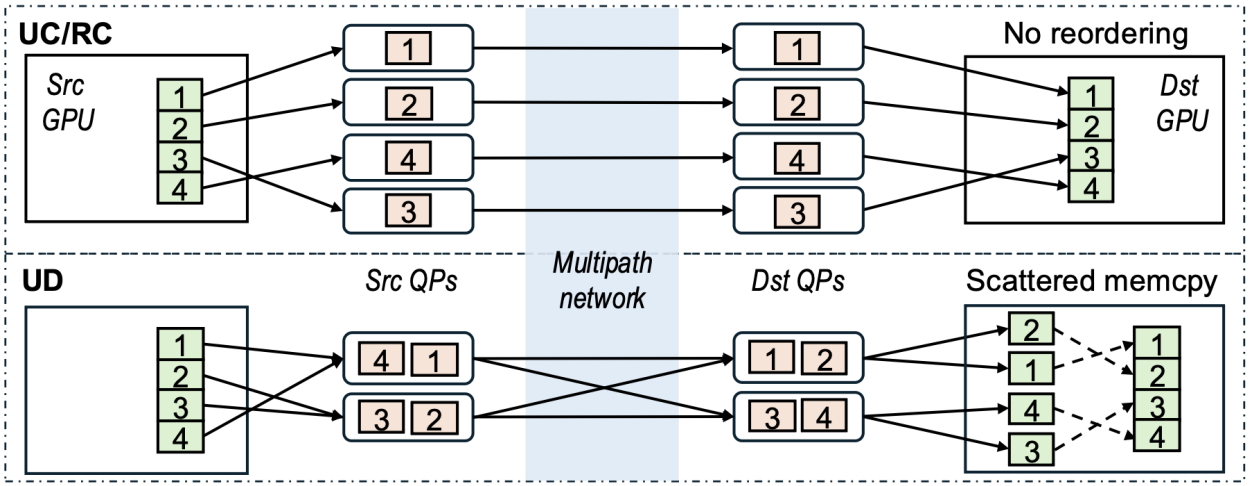


Image Credit: https://www.usenix.org/conference/osdi26/presentation/zhou-yang

Threading Model

- Load balancing and loss recovery can also work on a chunk of several packets that travel together
- Divide each connection's work into multiple **threads**

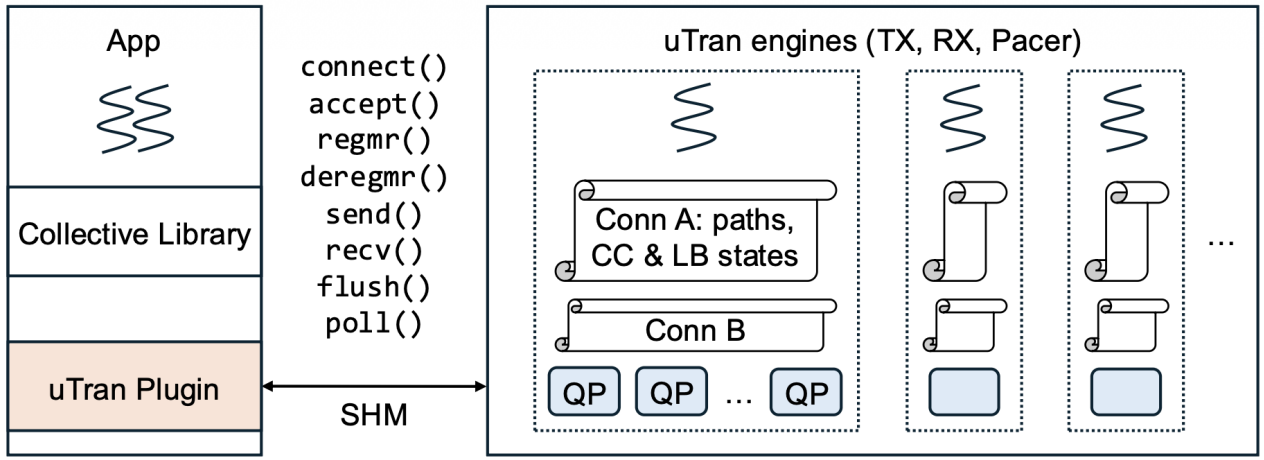


Image Credit: https://www.usenix.org/conference/osdi26/presentation/zhou-yang

Network Job Scheduling

Why Job Placement Matters for the Network?

- For scheduling jobs, existing schedulers use fairness and try to reduce network overhead, but not enough
- Training traffic is **periodic**
- Network heavily used in "Up" phases and idle in "Down" phases
- Link is overloaded when jobs collide

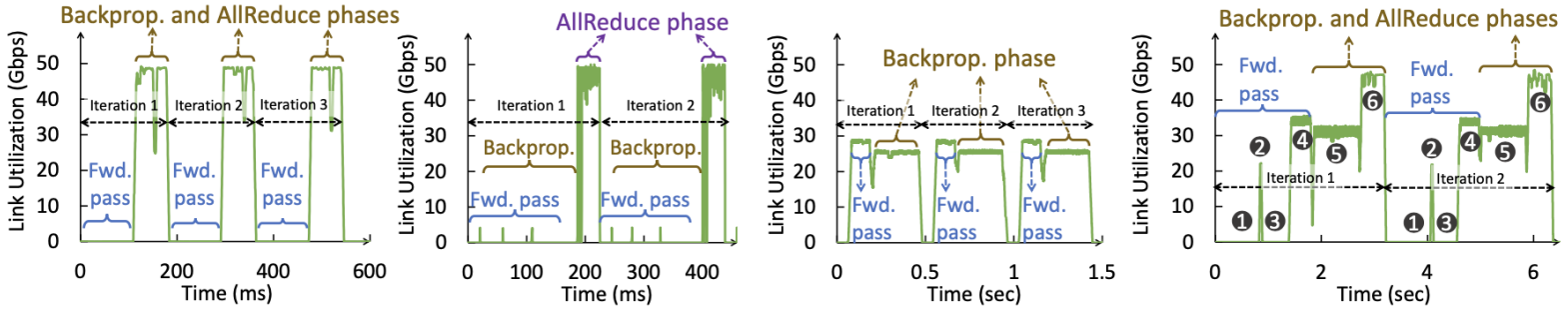
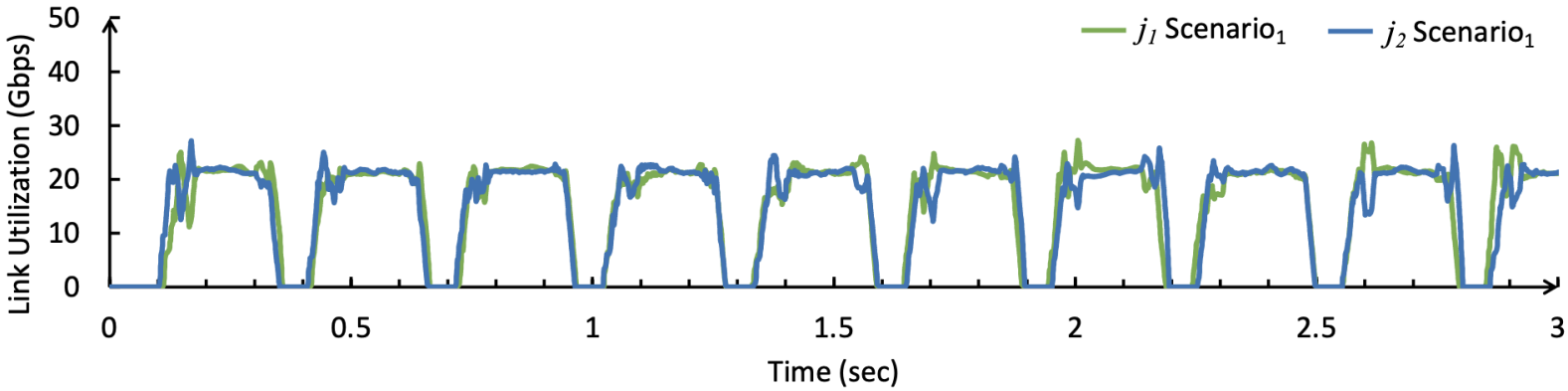
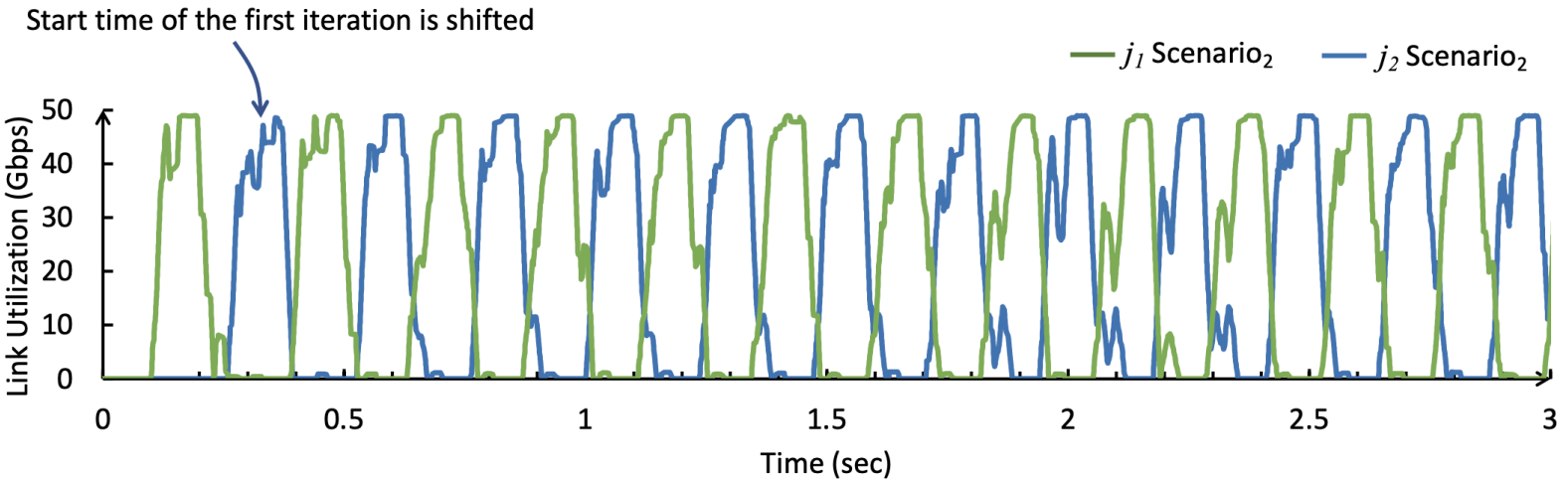


Image Credit: https://www.usenix.org/conference/nsdi24/presentation/rajasekaran

Interleaving Communication Phases

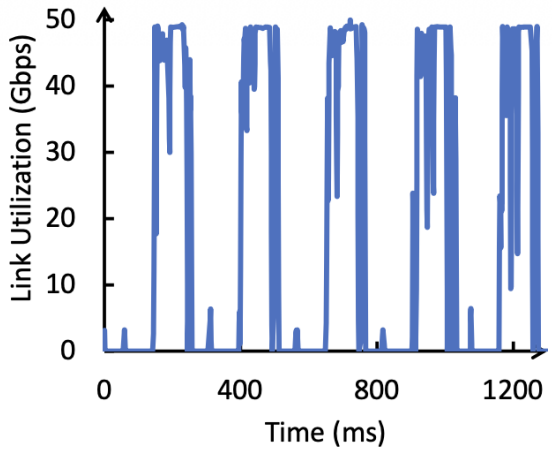
- "Up" phase of one job fit into "Down" of another



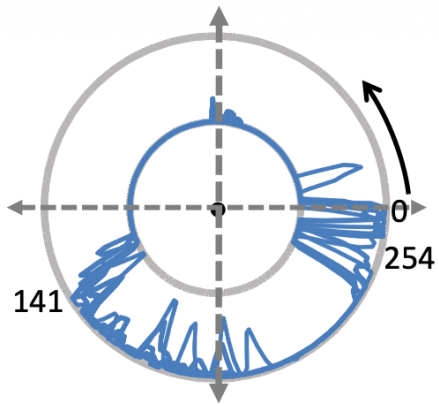


When Can Jobs Be Interleaved?

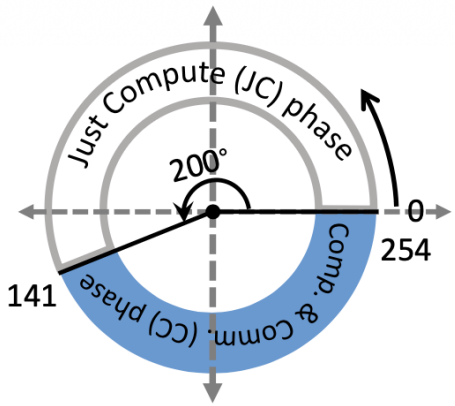
- **CASSINI** describes each job as a **circle** and searches over rotations of these circles
- Perimeter of the circle is the iteration time



(a) Network demand



(b) Network demand rolled around a circle

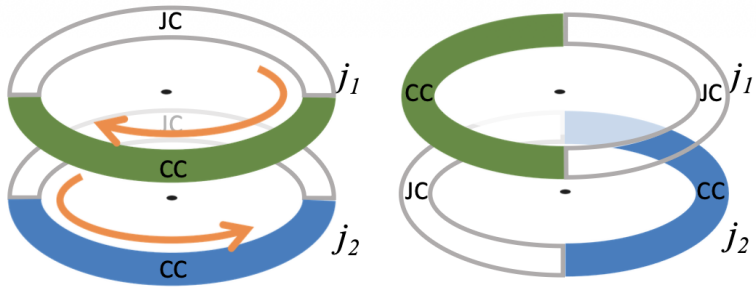


(c) Geometric abstraction

Image Credit: https://www.usenix.org/conference/nsdi24/presentation/rajasekaran

When Can Jobs Be Interleaved?

- For two jobs sharing a link, we put their circles on top of each other and add their demands at every angle
- Rotating a circle is equivalent to delaying a job
- Jobs with different iteration times share a circle with perimeter equal to LCM of iteration times



(a) CC phases are colliding

(b) CC and JC phases are interleaved

Image Credit: https://www.usenix.org/conference/nsdi24/presentation/rajasekaran