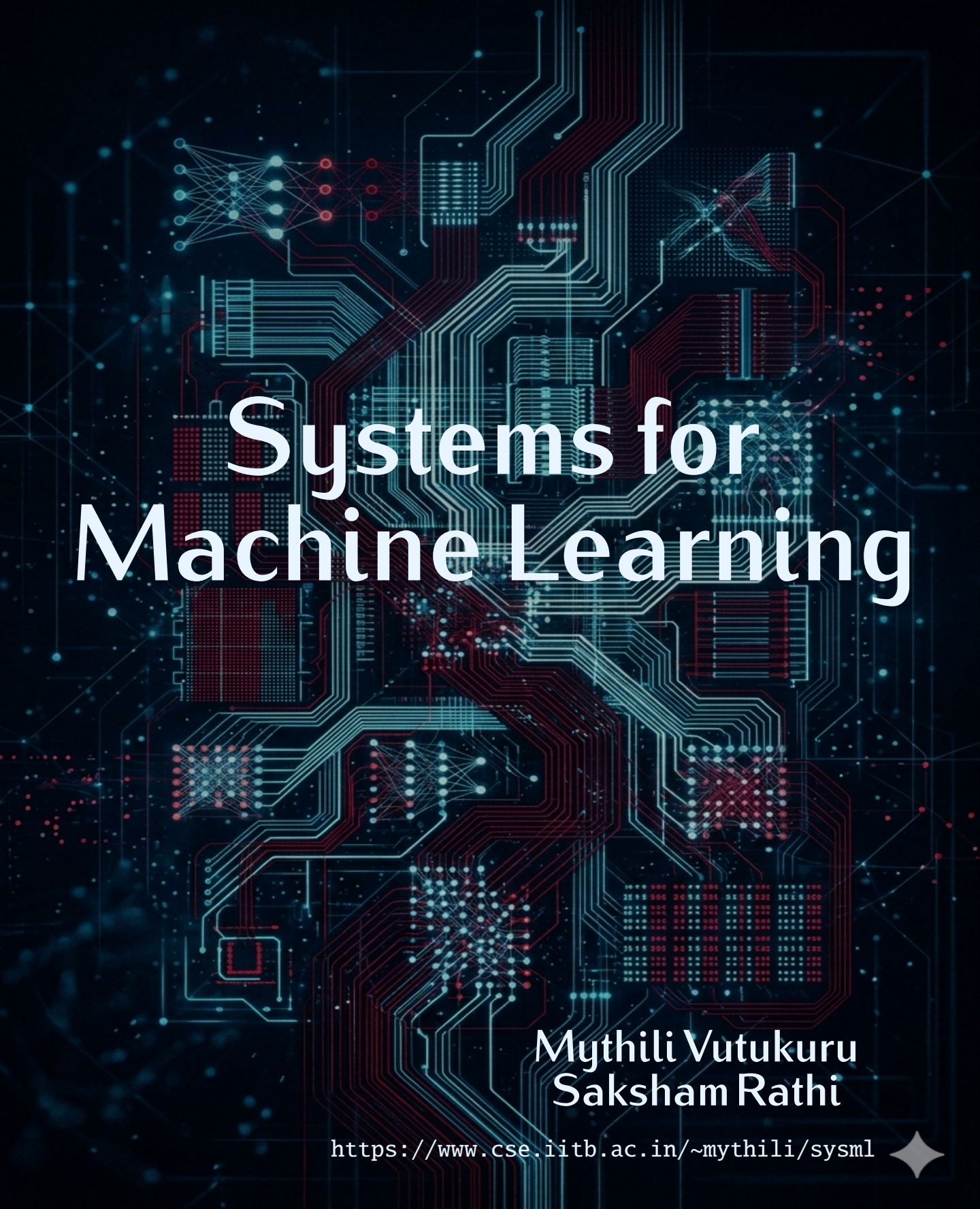

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

November 2026

7

Network Optimizations

7.1 Network Design for ML Workloads . . . 3
7.1.1 Bursty Workload . . . 3
7.1.2 Low Entropy and Hash Polarization . . . 4
7.1.3 Parallelism-Driven Communication Patterns . . . 5
7.1.4 Failure Sensitivity . . . 6
7.2 Data Center Networks and Interconnects for ML Training . . . 8
7.2.1 Traditional Data Center Networks . . . 8
7.2.2 Remote Direct Memory Access (RDMA) . . . 9
7.2.3 Connectivity Inside and Across GPU Nodes . . . 14
7.2.4 Case Study: Alibaba's HPN . . . 17
7.3 Collective Communication . . . 22
7.3.1 Collective Communication Primitives . . . 22
7.3.2 The NCCL API . . . 26
7.3.3 Challenges in Implementing Collectives Efficiently . . . 29
7.3.4 Communication Topologies . . . 30
7.3.5 AllReduce over a Ring . . . 31
7.3.6 AllReduce over a Tree . . . 33
7.3.7 AllReduce on Multiport Direct-Connect Networks . . . 35
7.4 Transport and Communication Protocols . . . 42
7.4.1 RDMA Transport and Communication Protocols . . . 42
7.4.2 Software-Based Transport . . . 49
7.4.3 Case Study: Scaling the Communication Stack to Thousands of GPUs . . . 58
7.5 Network Job Scheduling . . . 63
7.5.1 Why Job Placement Matters for the Network? . . . 63
7.5.2 Interleaving Communication Phases . . . 64

7.5.3 When Can Jobs Be Interleaved? 66
7.6 Summary 68

The previous chapter introduced distributed training as the set of techniques needed once a model, its activations, or its dataset grow too large for a single GPU to handle. We saw that training consists of a forward pass, a backward pass, and an optimizer step, and the activation memory swells during the forward pass and drains during the backward pass, while gradient memory appears only in the backward pass, and parameter and optimizer state persists for the entire run. To cope with these issues, we studied two complementary strategies for splitting work across GPUs: data parallelism, which replicates the model and partitions the data, and model parallelism, which partitions the model itself so that no single device must hold it in full. We looked closely at techniques such as pipeline parallelism, which splits the model into stages executed on different devices, and tensor parallelism, which splits individual layers across devices so that even a single matrix multiplication can be executed cooperatively. We also examined how real systems combine these strategies, through case studies of training runs that span thousands of GPUs, and we saw that virtually every one of these techniques, however it partitions memory or compute, ultimately depends on GPUs exchanging large volumes of data with one another over a network.

This chapter turns to that network itself. Every parallelism strategy we studied in the previous chapter, whether it is the AllReduce of gradients in data parallelism, the point-to-point transfers of activations in pipeline parallelism, or the AllGather and ReduceScatter operations of tensor parallelism, ultimately reduces to a communication pattern imposed on the physical network connecting GPUs. If the network cannot deliver these communication patterns efficiently, none of the memory and compute optimizations from the previous chapter matter, because the GPUs will simply sit idle waiting for data to arrive. In this chapter, we study what makes networking for machine learning workloads different from networking for traditional cloud or web applications, and how these differences motivate new choices in topology design, transport protocols, and system implementation. We begin by examining the unique characteristics of ML traffic, including its burstiness, its low entropy, and its sensitivity to failure. We then study how modern GPU clusters are physically wired together, looking at concepts such as NCCL, remote direct memory access (RDMA), and rail-optimized topologies that are purpose-built for the communication patterns of large-scale training. From there, we look at how collective communication operations, such as AllReduce and AllToAll, are actually implemented on top of these topologies.

7.1 Network Design for ML Workloads

Before we can discuss how to build or optimize a network for machine learning workloads, we must first understand why the networks that already serve traditional cloud computing are not simply good enough. At first glance, the problem posed by ML training looks similar to any other large-scale distributed systems problem — many machines need to exchange data quickly and reliably over a shared network fabric. Data center networks have been built to serve exactly this kind of need for decades, supporting web services, databases, and storage systems that also depend on fast, reliable communication between thousands of servers. However, when we look closely at the shape of the traffic generated during distributed training, several features stand out that are either absent or far less pronounced in traditional cloud workloads. These features, taken together, mean that a network designed and tuned for typical cloud traffic will perform poorly when asked to support the training of a large language model. We discuss each of these features in turn.

7.1.1 Bursty Workload

The most immediately visible difference between ML training traffic and traditional cloud traffic is its burstiness. A typical cloud computing workload, such as a web service or a database handling customer requests, produces traffic that is relatively smooth over short timescales and follows a predictable diurnal pattern over the course of a day, rising as users in a given time zone become active and falling as they go to sleep. Figure 7.1 shows this pattern from a production cloud environment, where traffic in and out of the network rises and falls gradually over a 24-hour cycle, and the number of connections active at any given moment changes slowly enough that a network built to handle the average load, plus some safety margin, comfortably handles the peaks as well.

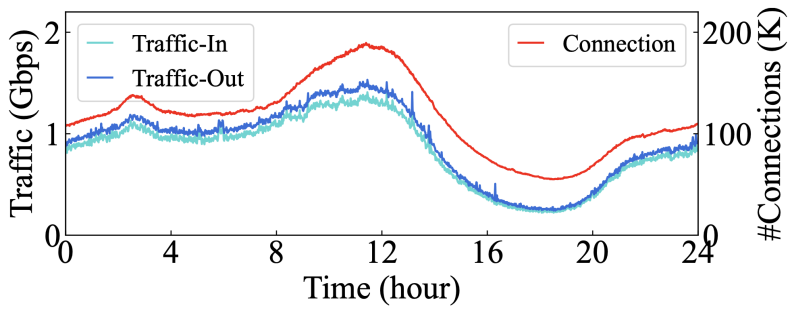


Figure 7.1: Traditional cloud computing traffic pattern; Image credit: [Qian et al., 2024]

Distributed training looks nothing like this. Recall from the previous chapter that every parallelism strategy periodically requires GPUs to synchronize with one another — data-parallel workers must average their gradients before the optimizer step, pipeline-parallel stages must exchange activations and gradients at each micro-batch boundary, and tensor-parallel workers must exchange

partial results within every layer. Between these synchronization points, each GPU is busy with local computation and sends essentially no traffic over the network. Then, when a synchronization point is reached, every participating GPU wants to send and receive a large volume of data as quickly as possible, so that computation can resume with minimal delay. The result is a atypical traffic pattern — long silences followed by short, extremely intense bursts, often saturating the full bandwidth of a network interface card (NIC) at hundreds of gigabits per second. Figure 7.2 shows this pattern measured on a production training cluster, where the egress throughput on a set of NICs jumps from near zero to hundreds of gigabits per second in a matter of seconds, holds briefly, and then falls back to zero, repeating this cycle throughout training.

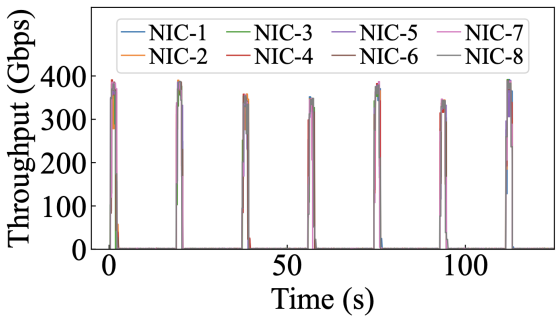


Figure 7.2: NIC egress traffic pattern during production model training; Image credit: [Qian et al., 2024]

This burstiness has real consequences for network design. Provisioning the network for peak traffic seen during a burst will prove to be impractical and expensive. On the other hand, a network provisioned for the average traffic of a training job, rather than its peak, will introduce congestion and queuing delay exactly at the moments when GPUs are waiting on each other, directly slowing down the training job as a whole. Because computation is stalled during these communication bursts, and typically cannot be usefully overlapped with unrelated work, any delay in the network is not hidden but instead adds directly to the wall-clock time of training. This is very different from a cloud service, where a slow response to one request rarely blocks progress on the rest of the system.

7.1.2 Low Entropy and Hash Polarization

A second and closely related challenge is that ML training traffic has very low entropy compared to traditional cloud traffic. A typical data center supporting cloud services carries traffic between a very large number of source-destination host pairs, often on the order of a hundred thousand distinct flows at any given time. Because there are so many flows, and each is relatively small compared to the overall link capacity, standard load-balancing mechanisms work well. For example, equal-cost multi-path (ECMP) routing hashes properties of each flow, such as its source and destination

addresses and ports, to choose among several equal-cost paths through the network. With a large number of flows, this hashing spreads traffic roughly evenly across the available paths. When two large flows happen to hash to the same path, one can end up starving the other, but with enough independent flows sharing many links, such collisions average out and are rarely a serious problem.

Distributed training breaks this assumption. Because each GPU server typically maintains only a handful of long-lived, high-bandwidth connections to its collective communication partners, the number of distinct flows in a training cluster is orders of magnitude smaller than in a general-purpose data center, and each of those flows is correspondingly larger. Figure 7.3 shows measurements of the number of connections per host in a production training cluster: the large majority of hosts maintain only a small number of connections, in sharp contrast to the hundred thousand flows typical of general cloud traffic. With so few large flows, the law of averages that makes ECMP effective in a normal data center no longer applies. A single unlucky hash collision between two large training flows can now noticeably degrade the throughput of both, and because collective communication operations require every participant to finish before training can proceed, one degraded flow can slow down the entire job, not just the two hosts directly involved. This phenomenon, sometimes called hash polarization, means that load balancing strategies which work well for traditional cloud workloads must be reconsidered, or replaced entirely, for machine learning training clusters.

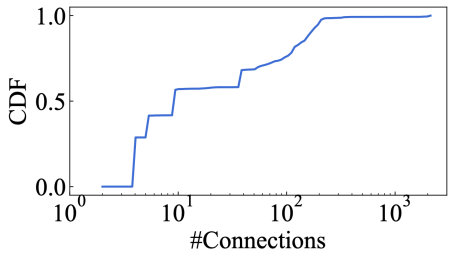
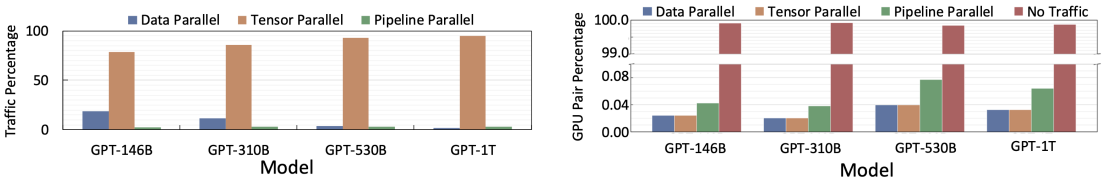


Figure 7.3: Number of connections per host; Image credit: [Qian et al., 2024]

7.1.3 Parallelism-Driven Communication Patterns

A third source of structure in training traffic comes from the fact that not all communication in a training job is equal, either in volume or in its sensitivity to latency. As we saw in the previous chapter, data parallelism, pipeline parallelism, and tensor parallelism each induce very different communication patterns. Tensor parallelism splits individual layers across GPUs and therefore requires communication after nearly every operation within a layer; this communication tends to be latency-sensitive and difficult to overlap with computation, so tensor-parallel groups are almost always kept within a small number of GPUs that are physically close together, often within the same server, connected by very high bandwidth, low latency interconnects. Data parallelism and pipeline parallelism, by contrast, communicate less frequently and can more easily tolerate being spread across

a larger number of nodes, since their communication can sometimes be overlapped with computation or pipelined across micro-batches. Figure 7.4 shows measurements from large GPT-style training runs confirming this intuition — the overwhelming majority of network traffic, by volume, is generated by tensor parallelism, and this traffic is concentrated among small groups of GPUs, while data-parallel and pipeline-parallel traffic, though they can span many more GPU pairs, contributes a much smaller share of total bytes moved. This traffic locality has direct implications for network topology, since it suggests that expensive, high-bandwidth interconnects are best concentrated within small groups of tightly coupled GPUs, while a comparatively cheaper network can suffice for the sparser traffic between these groups, a theme we return to later in this chapter.



(a) Unbalanced load

(b) Balanced load

Figure 7.4: Traffic volume and traffic type across GPU pairs for GPT-style models of varying sizes; Image credit: [W. Wang et al., 2024]

7.1.4 Failure Sensitivity

Finally, distributed training clusters are far more sensitive to failure than typical cloud deployments, and this sensitivity interacts directly with the network. Because collective communication requires every participating GPU to reach a synchronization point together, the failure of a single GPU, NIC, or network link does not merely degrade the performance of the unlucky component; it can stall the entire training job, since every other GPU will be left waiting indefinitely for a peer that will never respond. Recovering from such a failure typically means restarting the training job from the most recent checkpoint, and because checkpointing large models is itself an expensive operation that briefly pauses training to write parameter and optimizer state to persistent storage, checkpoints are usually taken only every few hours rather than continuously. Figure 7.5 shows checkpoint intervals for several representative large language model training jobs in production, most falling somewhere between one and a half and three and a half hours. A single failure occurring shortly before a checkpoint can therefore erase several hours of otherwise successful training, an outcome with no real analogue in most cloud services, where a failed request can usually simply be retried at negligible cost.

Making matters worse, link failures are not a rare occurrence even in otherwise reliable, well-managed production networks. Figure 7.6 shows the monthly link failure ratio observed in a large production training cluster over the course of a year. Even though the fraction of links failing in

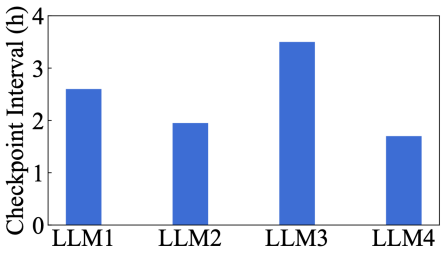


Figure 7.5: Checkpoint intervals; Image credit: [Qian et al., 2024]

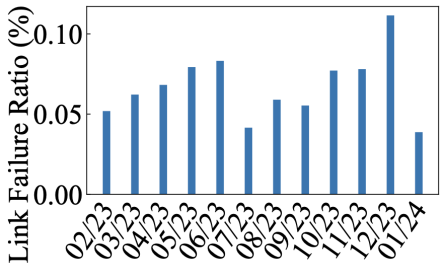


Figure 7.6: Monthly link failure ratio; Image credit: [Qian et al., 2024]

any given month is small, on a network built from many thousands of links this translates into one or two link failures affecting a typical training job every month. Combined with the long checkpoint intervals discussed above, this means that network reliability, and the ability to detect and route around failures quickly, is not a peripheral concern for machine learning training clusters but a design requirement.

Taken together, these four characteristics — extreme burstiness, low entropy with a small number of very large flows, strong locality driven by the parallelism strategy in use, and sensitivity to failure — define the network design problem that the rest of this chapter addresses. We need a network that can deliver hundreds of gigabits per second to each host during brief, synchronized bursts; that does not rely on the statistical averaging of many small flows to balance load; that concentrates bandwidth where communication is most intense while still connecting the rest of the cluster affordably; and that can tolerate, detect, and recover from failures without derailing days or weeks of expensive training. In the sections that follow, we study a few broad categories of solutions to this problem: how the physical and logical topology of the network can be designed around these traffic patterns, how collective communication operations are implemented efficiently on top of such topologies, and what other techniques and research directions are emerging to address the remaining gaps.

7.2 Data Center Networks and Interconnects for ML Training

The previous section argued, from first principles, why a network tuned for ordinary cloud traffic struggles once it is asked to carry the bursty and failure-sensitive traffic of distributed training. This section makes that argument concrete. We first review how traditional data center networks are actually built, since the vocabulary of that design (core, aggregation, and edge layers, and the Clos or fat-tree topologies underlying them) is the baseline against which every ML-specific design is compared. We then introduce Remote Direct Memory Access (RDMA), the networking primitive that almost every ML-specific network design is based on. Equipped with that background, we zoom in from the level of switches and racks down to the level of a single server, examining how the GPUs inside one machine are wired together and how a library such as NCCL actually moves bytes from one GPU's memory to another's. Finally, we zoom back out and walk through two philosophies for wiring together tens of thousands of GPUs — Alibaba's HPN, a production network that provides full any-to-any connectivity, and the Rail-only design, which asks whether that connectivity is even necessary for LLM training traffic.

7.2.1 Traditional Data Center Networks

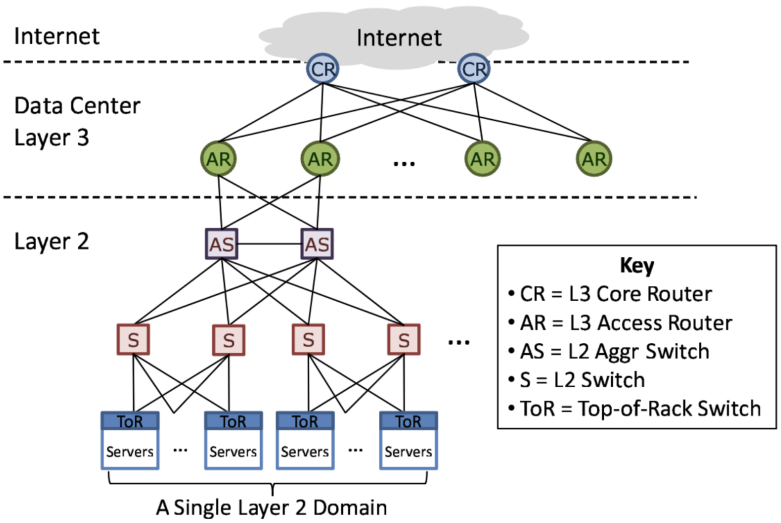


Figure 7.7: A traditional three-tier data center network; Image credit [Greenberg et al., 2009]

Most data center networks built over the last two decades, whether they support a search engine, a social network, or a database backend, are organized around the idea of a Clos network, also referred to informally as a fat-tree. Figure 7.7 shows the shape of such a network, as popularized

by designs like VL2 [Greenberg et al., 2009]. Servers sit at the bottom of the tree, each connected to a Top-of-Rack (ToR) switch that aggregates the traffic of a single rack of machines. ToR switches connect upward to a layer of aggregation switches, and aggregation switches connect further upward to a layer of core routers, which in turn connect the data center to the wider internet. The defining property of a Clos topology is that there are many parallel paths between any two servers — a packet leaving one rack can reach almost any other rack by any of several equally valid routes through the aggregation and core layers. This redundancy is what allows the network to be built from many small, identical, inexpensive switches rather than a handful of enormous, custom ones, and it is what allows the network to keep functioning even when individual links or switches fail.

Because a Clos network offers many equally good paths between any pair of endpoints, it needs some mechanism to decide, for each individual flow of traffic, which of those paths to use. The standard mechanism is Equal-Cost Multi-Path (ECMP) routing. Each switch computes a hash over a handful of fields identifying the flow, typically the source and destination IP addresses and port numbers, and uses the result of that hash to pick one of several equal-cost next hops. Because the hash is deterministic, all packets belonging to the same flow take the same path, which keeps them in order, while different flows are likely to hash to different paths and therefore spread out across the network's capacity. As we discussed in the previous section, this scheme depends on having a large number of independent flows to average over. With a hundred thousand flows sharing a link, the law of large numbers guarantees that the traffic distributes itself roughly evenly, even though no single switch has any global view of network load.

7.2.2 Remote Direct Memory Access (RDMA)

Before we can talk sensibly about how a GPU cluster's network is designed, we need to understand the mechanism almost every design in this section relies on to move bytes between hosts — Remote Direct Memory Access, or RDMA. RDMA is not specific to machine learning; it was originally developed for high-performance computing and storage. But its properties, such as extremely low latency, near-zero CPU involvement, and the ability to sustain hundreds of gigabits per second on a single connection, are exactly what large collective operations such as AllReduce need, which is why RDMA has become the default transport for the backend network of essentially every modern training cluster.

Traditional Network Stack vs. RDMA

Consider what happens when an ordinary application sends data over a conventional TCP/IP socket. At the start, the application provides a userspace buffer to the operating system kernel. The kernel copies data from that buffer into a kernel socket buffer, then runs it through the TCP/IP protocol stack (computing checksums, managing sequence numbers, handling congestion control and retransmission state), and eventually hands the resulting packets to the network interface card (NIC) for transmission. On the receiving side, the process runs in reverse — the NIC deposits incoming packets into a kernel buffer, the kernel processes them through the TCP/IP stack, and then copies the data

one more time into the buffer of the receiving application. Every one of these steps costs CPU cycles, and every one of the copies costs time and memory bandwidth. For an ordinary web request this overhead is irrelevant, but for a training job that needs to move gigabytes of gradients within a fraction of a second, hundreds of times per hour, both the copies and the CPU involvement become the bottleneck.

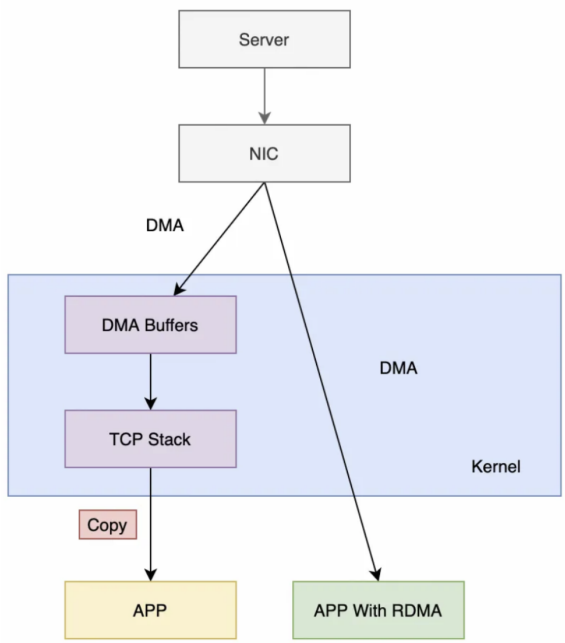


Figure 7.8: Paths with and without RDMA; Image credit [Yujing, 2024]

RDMA removes both problems by letting the network interface card write data directly into, or read data directly out of, the memory of an application without the CPU or operating system kernel being involved at all (Figure 7.8). The application still initiates the transfer, but once initiated, the RDMA-capable NIC on each end handles the data movement itself — it performs the equivalent of a hardware-managed memory copy across the network from the local application buffer to the remote application buffer. Hence, RDMA is commonly referred to as a zero-copy kernel bypass mechanism. The hardware also performs all protocol processing traditionally done inside the kernel, including congestion and flow control. Because the CPU is not copying bytes or running protocol code, RDMA transfers can achieve latencies an order of magnitude lower than conventional sockets and can saturate multi-hundred-gigabit links while leaving the CPU almost entirely free to do other work, such as running the next step of the training loop.

Queue Pairs, Work Requests, and Completions

Applications do not talk to an RDMA-capable NIC through ordinary socket calls. Instead, they use an interface known as "verbs", e.g., read/write or send/receive. The central abstraction in verbs is the queue pair (QP) — a send queue and a receive queue that together represent one endpoint of a connection. An application can have multiple work queues based on its requirements. Before RDMA transfers can begin, the application shares a region of pinned host memory with the RDMA-capable NIC. To issue an RDMA operation, the application constructs a work request describing what it wants done (for example, "write the following 4 megabytes of my local memory to the remote peer") and posts it to the send queue as a work queue element, or WQE (Figure 7.9). From that point on, the NIC's hardware takes over — it reads the WQE, performs the requested memory transfer, and, once the operation is complete, writes a completion queue element, or CQE, into a completion queue that the application can poll or wait on. This flow — application posts a work request, hardware executes it asynchronously, application later observes a completion — is the reason RDMA transfers can overlap so cleanly with GPU computation. For example, a training framework can post a send for one chunk of gradients and immediately return to other work while the transfer proceeds in the background.

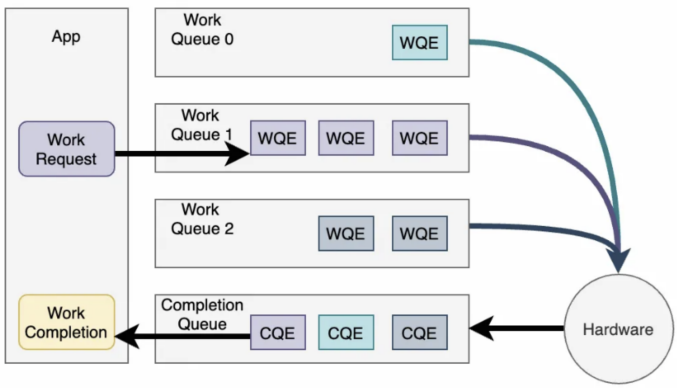


Figure 7.9: Work and completion queues; Image credit [Yujing, 2024]

RDMA transfers are of two types — one-sided operations (read/write) and two-sided operations (send/receive). With one-sided RDMA operations, the remote node's application is not involved in the critical path of the data transfer. The two applications exchange the information needed to reach each other's memory (buffer address, remote key, and queue pair number) out of band, typically over an ordinary TCP connection, before the data transfer begins. So when an application initiates a one-sided RDMA write, the local NIC directly deposits the data into the buffers of the remote application, without any involvement of the application on the remote CPU, and then notifies the local application of the completion. Now, because the remote application is not involved in any way, it does not get

notified when data arrives, and must enable some lightweight RDMA signalling mechanisms (e.g., RDMA write with immediate data notification) if it wishes to process arriving data with minimal latency. In contrast, for two-sided RDMA, both the sender and receiver applications must actively participate in the data transfer. For example, the remote application allocates buffers in which to receive incoming data, and monitors the completion queue actively for data arrival notifications from its RDMA-capable NIC. The networking stacks in modern ML applications use both flavours of RDMA. The two-sided RDMA mode is typically preferred for control plane operations like topology setup and handshakes, while the faster one-sided RDMA is preferred for data plane operations like transferring large gradients.

A simplified sketch of what an application does when issuing an RDMA write with the verbs API is shown in the code below. The exact function names come from the widely used `libibverbs` library. The code below shows the sender side of a one-sided RDMA write using `libibverbs`. Steps 1–4 are one-time setup that both peers perform — open the device, register memory, create the queue pair and completion queue, and exchange connection details. Step 5 is the data path and is executed only by the sender; the receiver posts no work request and runs no code when the data arrives.

Code 7.1: RDMA

```
// ---- 1. Open the NIC; create a protection domain and a completion queue ----
struct ibv_device **devs = ibv_get_device_list(NULL);
struct ibv_context *ctx  = ibv_open_device(devs[0]);
struct ibv_pd *pd = ibv_alloc_pd(ctx);                      // scopes all
    resources below
struct ibv_cq *cq = ibv_create_cq(ctx, 16, NULL, NULL, 0);  // CQEs will appear
    here

// ---- 2. Register (pin) a memory region so the NIC can access it directly ----
void *buffer = aligned_alloc(4096, buffer_size);
struct ibv_mr *mr = ibv_reg_mr(pd, buffer, buffer_size,
                               IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE);
// mr->lkey is used locally; mr->rkey is given to the peer so it may write here

// ---- 3. Create the queue pair (send queue + receive queue) ----
struct ibv_qp_init_attr qp_attr = {
    .qp_type = IBV_QPT_RC,                 // reliable connection
    .send_cq = cq,                         // completions of sends go to cq
    .recv_cq = cq,
    .cap = { .max_send_wr = 16, .max_recv_wr = 16,
             .max_send_sge = 1, .max_recv_sge = 1 },
};
```

```
struct ibv_qp *qp = ibv_create_qp(pd, &qp_attr);

// ---- 4. Exchange connection info out of band (e.g., over TCP), then connect ----
struct conn_info local = { .addr = (uintptr_t)buffer, .rkey = mr->rkey,
                           .qpn = qp->qp_num, .lid = port_lid };
struct conn_info remote;
exchange_over_tcp(&local, &remote);        // sends 'local', receives peer's info
connect_qp(qp, &remote);                   // ibv_modify_qp: INIT -> RTR -> RTS

// ---- 5. Sender only: one-sided RDMA WRITE into the peer's memory ----
struct ibv_sge sge = { .addr = (uintptr_t)buffer, .length = len, .lkey = mr->lkey };
struct ibv_send_wr wr = {
    .wr_id      = 1,
    .sg_list    = &sge,
    .num_sge    = 1,
    .opcode     = IBV_WR_RDMA_WRITE,       // one-sided: remote CPU not involved
    .send_flags = IBV_SEND_SIGNALED,       // request a CQE when done
    .wr.rdma    = { .remote_addr = remote.addr, .rkey = remote.rkey },
}, *bad_wr;
ibv_post_send(qp, &wr, &bad_wr);           // enqueue WQE; returns immediately

// ... do other useful work (e.g., GPU compute) while the NIC transfers ...

struct ibv_wc wc;
while (ibv_poll_cq(cq, 1, &wc) == 0) { }   // wait for the CQE
// wc.status == IBV_WC_SUCCESS and wc.wr_id == 1 means the write completed
```

InfiniBand and RoCE

RDMA was first popularised on InfiniBand, a network technology with its own switches, cabling, and link-layer protocol designed for lossless, low-latency transport. InfiniBand remains common in high-performance computing and in some GPU clusters, but it requires a dedicated network fabric, distinct from the commodity Ethernet most data centres are already built on. To avoid this vendor and fabric lock-in, the industry developed RoCE (RDMA over Converged Ethernet), which implements the same verbs programming model and largely the same wire protocol on top of ordinary Ethernet hardware. RoCE requires the Ethernet fabric to be configured for lossless operation (typically using Priority Flow Control) so that the RDMA transport, which assumes packets are rarely dropped, performs well. From the perspective of the application, and from the perspective of everything we discuss for the rest of this chapter, InfiniBand and RoCE are interchangeable — both expose the verbs interface

described above, and a training framework such as NCCL does not need to know or care which one is underneath.

GPUDirect RDMA

Ordinary RDMA solves the problem of moving data between two hosts without involving their CPUs, but on a GPU-equipped server, there is still a gap — the data to be sent usually lives in GPU memory, not host memory. Without further support, a transfer would need to first copy the data from GPU memory into host memory (across the PCIe bus, using the CPU), before the NIC could pick it up and send it, and symmetrically on the receiving side. That extra copy reintroduces exactly the CPU and bandwidth overhead RDMA was meant to eliminate.

NVIDIA's GPUDirect RDMA closes this gap by allowing an RDMA-capable NIC to read from, or write directly into, GPU memory, completely bypassing host memory and the CPU (Figure 7.10). This works because a modern server's GPUs and NICs, when attached to the same PCIe switch, can address each other's memory directly over PCIe; GPUDirect RDMA simply extends the RDMA verbs machinery so that the memory region registered with the NIC can be a region of GPU memory rather than host memory. The practical effect is a genuinely direct path — GPU memory, across PCIe, into the NIC, out onto the network, into the remote NIC, across PCIe, into the remote GPU's memory, with the CPU on neither end touching a single byte of the actual payload. This is the mechanism that makes it possible for a server to sustain multiple terabits per second of aggregate network throughput.

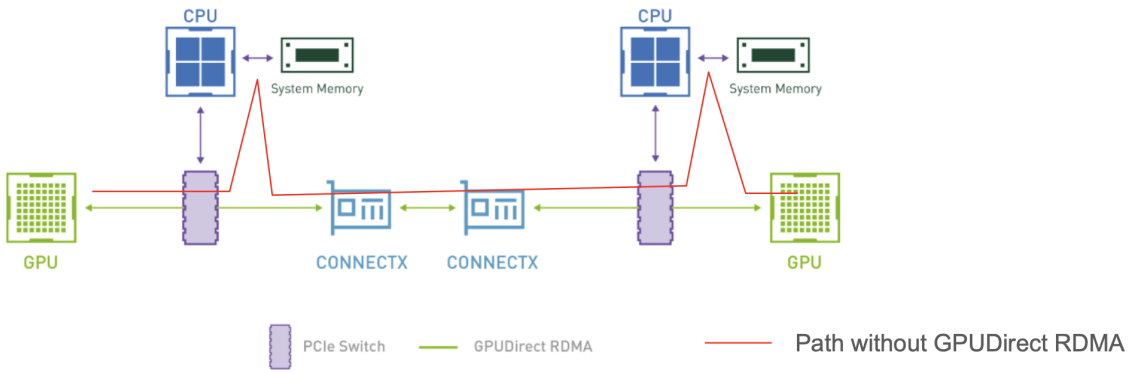


Figure 7.10: GPUDirect RDMA; Image credit [NVIDIA Corporation, 2023]

7.2.3 Connectivity Inside and Across GPU Nodes

With RDMA and GPUDirect RDMA established as the underlying transport, we can now describe how the GPUs inside one server are wired together, and how a collective communication library such as NCCL decides, for any given pair of communicating GPUs, which physical path their data

should take.

A GPU node, as described in Chapter 3, is a single server containing several GPUs connected to one another by a dedicated, very high bandwidth interconnect internal to the server. The canonical example is NVIDIA's NVLink, which, depending on the GPU generation, provides several hundred gigabytes to several terabytes per second of bidirectional bandwidth between GPUs in the same server, typically an order of magnitude more than what a single network of the server can offer. AMD's equivalent, Infinity Fabric, plays the same role. Whatever the exact interconnect, what matters is that a small group of GPUs within a GPU node (eight in the common DGX-style server, up to 256 in the largest NVSwitch-based systems) shares a pool of bandwidth so much larger than the network outside the server that, for design purposes, it can be treated as an almost free resource, while the network connecting different GPU nodes is treated as the scarce one.

To connect a GPU node to the rest of the cluster, each GPU is typically given its own dedicated network interface card. Rather than sharing a small number of NICs across all eight GPUs in a server, which would force GPUs to compete for network access, one NIC is wired to one GPU, so that GPU's outgoing traffic never has to contend with another GPU on the same host for the same physical port. Figure 7.11 shows this arrangement as deployed in Alibaba's network: eight of the host's nine NICs are dedicated one-to-one to the eight GPUs and carry backend training traffic, while a ninth NIC is set aside for frontend management traffic such as job scheduling, dataset loading, and checkpoint I/O.

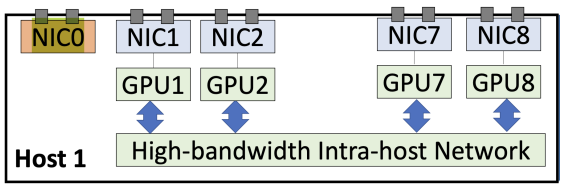


Figure 7.11: Connectivity within a GPU node; Image credit [Qian et al., 2024]

The library responsible for actually orchestrating GPU-to-GPU data movement in almost every modern training framework is NVIDIA's Collective Communication Library, NCCL. A recent systematic study of NCCL's internals [Hu et al., 2026] gives us a precise picture of how it chooses a physical path for any given transfer, and the picture differs sharply depending on whether the two communicating GPUs sit in the same server or in different servers.

Intra-node Transfer

When both GPUs are in the same host, NCCL's first preference is always a direct, hardware-mediated path that never touches host memory. If the two GPUs are connected by NVLink, NCCL lets one GPU's DMA engine copy data directly into the other GPU's memory over NVLink. If NVLink is unavailable, NCCL falls back to using the PCIe bus instead, which is slower but still avoids host

memory. Figure 7.12 illustrates this. When a mode called P2P_DIRECT is available, data moves from the source buffer on GPU 0, through GPU 0's DMA engine, straight into the destination buffer on GPU 1. If this direct mode is not available, the data is copied through a small intermediate FIFO staging buffer on GPU 1. Only when direct GPU-to-GPU access is unavailable or found to perform poorly, for example, between GPUs attached to different CPU sockets, does NCCL fall back to routing data through a shared memory segment in host memory, a slower path since it requires the data to cross PCIe twice and be handled by the host's memory system in between.

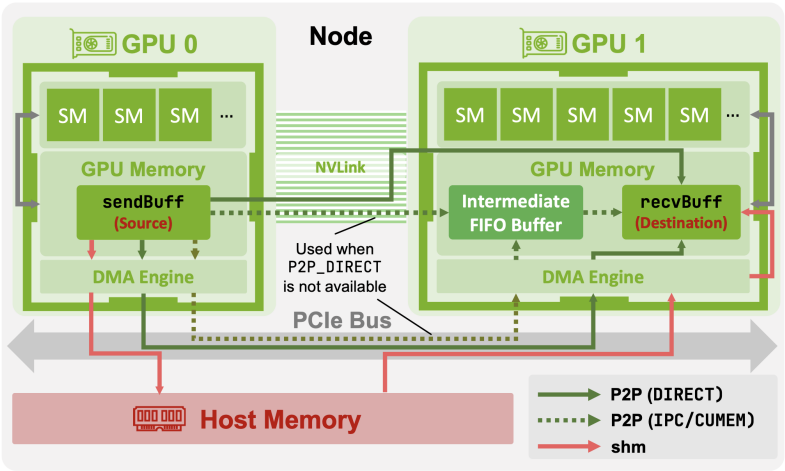


Figure 7.12: Intra-node data transfer paths in NCCL; Image credit [Hu et al., 2026]

Inter-node Transfer

When the two GPUs sit in different hosts, the data has to cross the physical network, and NCCL picks between two transports depending on what the NIC supports. If the NIC does not support RDMA, NCCL falls back to a plain TCP socket transport: data is copied from the GPU into a pinned host-memory buffer, sent over an ordinary socket, received into a host buffer on the far end, and copied into the destination GPU. Every one of those copies costs time, which is precisely why production training clusters equip every NIC with RDMA support. When RDMA with GPUDirect support is available, NCCL uses the IB verbs transport to move data directly from the GPU memory into the remote application's memory in a zero-copy manner, without involving host CPU or memory at all. Figure 7.13 shows both these cases — the green RDMA path goes straight between GPU memory and the NIC, while the red socket path detours through host memory and a CPU-managed proxy thread on both ends. Note that NCCL never manages this transfer entirely from GPU kernel code; a CPU-side proxy thread posts and monitors the RDMA work requests on behalf of the GPU, coordinating between the GPU's kernel, the NIC, and the network.

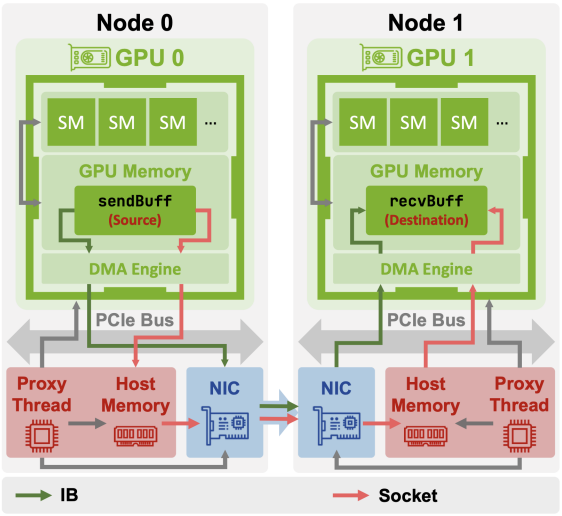


Figure 7.13: Inter-node data transfer paths in NCCL; Image credit [Hu et al., 2026]

7.2.4 Case Study: Alibaba's HPN

Alibaba's HPN (High Performance Network) is an example of a real-world data center network designed to support large scale LLM training production workloads. HPN had three concrete design goals — the network must scale to at least 15,000 GPUs in a single cluster (matching what Alibaba observed as the practical ceiling for a single training job, with headroom toward 100,000 GPUs as models grow further); it must minimize the number of network hops and hash operations a flow experiences, both to cut latency and to shrink the path-selection search space; and it must be resilient to network failures to the extent possible. To achieve these goals, the network is designed in a hierarchical manner, as shown in Figure 7.14. Hosts with 8 GPUs each are connected to a pair of ToR switches in a fault tolerant manner. The ToR switches are then wired to the aggregation layer switches in a way that allows load balancing of flows. The GPUs are grouped into segments and then into pods hierarchically, using a three-tier network comprising of ToR switches, aggregation switches, and core switches. Finally, the heavy-duty backend network that carries training traffic is separated from the light-weight frontend network that carries control plane traffic. We now discuss in detail the several interesting ideas that underpin this network design.

Non-stacked Dual-ToR

As we saw before, LLM training is sensitive to single-point failures. Because every GPU in a training job must reach the same collective operation before any of them can proceed, a fault on any single link stalls the entire job rather than merely degrading it, the way a single failure would in an

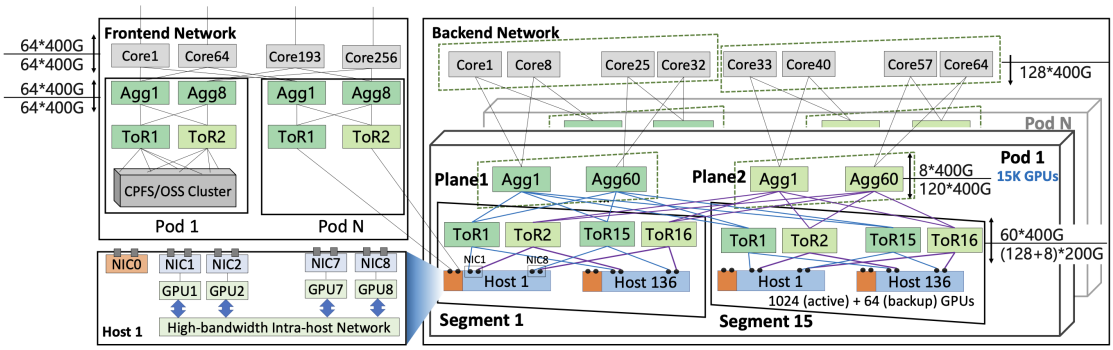


Figure 7.14: Overview of the HPN backend and frontend networks; Image credit: [Qian et al., 2024]

ordinary web service. Alibaba's production statistics quantify just how costly this is: a failure during LLM training costs roughly 20 times what an equivalent failure costs in general cloud computing, because the training job has to roll back to its last checkpoint, and checkpoints are taken only every few hours. Note that checkpointing itself is expensive, costing around 100 seconds and 30 gigabytes of storage per GPU each time it runs, so it cannot be done every iteration. Their monitoring further showed that although the aggregation and core layers of the network have ample redundant links, every NIC connects to the network through a single link to a single Top-of-Rack (ToR) switch, so a ToR failure alone can silently take down dozens to hundreds of GPUs at once.

The most direct way to remove ToR as a single point of failure is to give every NIC two independent uplinks to two different ToR switches, an arrangement generally called dual-ToR. The two ports of the NIC are configured with the same MAC and IP addresses and run in an active-active mode, so if one ToR or its uplink fails, traffic simply continues flowing through the other port without breaking any active flow, since both ports of the NIC share the same RDMA queue-pair context. The obvious way to build dual-ToR, and the one already shipped by switch vendors, is what Alibaba calls the stacked design. Here the two ToR switches are directly wired to each other so that they can synchronize their forwarding state (MAC tables, ARP entries) and agree, through a primary-secondary negotiation, on how to present themselves to the host as a single logical switch. This works well until the synchronization link itself becomes a liability. Alibaba observed a recurring failure mode in which one ToR silently stopped forwarding (for example, due to a hardware buffer overflow) while its control kept reporting itself as healthy over an independent management network. The healthy ToR, seeing that it could no longer synchronize state with its partner over the (now broken) direct link, concluded it could not guarantee consistent forwarding and shut itself down as a safety measure, taking every host under that dual-ToR pair offline. A second recurring failure mode came from software upgrades — whenever the two ToRs in a pair briefly ran different firmware versions during a rolling upgrade, mismatched fields in the state-synchronization messages between

them could crash the pair outright.

HPN's fix, called non-stacked dual-ToR, is to remove the direct synchronization link between the two ToRs entirely, so that each switch operates completely independently, and to solve the resulting coordination problem. Two problems have to be solved to make this work. First, the host's protocol module still expects to see a single logical switch, which normally requires the two physical switches to agree on a shared system identifier and on distinct port identifiers derived from that shared negotiation. HPN sidesteps the need for negotiation altogether: both switches are pre-configured with the same fixed, reserved virtual-router MAC address (rather than discovering it through the now-absent direct link), and each computes its own port identifiers by applying a different, fixed offset to its physical port numbers, so that the two switches produce a consistent identity without ever talking to each other. Second, the two switches need to keep a consistent view of which hosts are reachable through which ToR without a synchronization link to exchange that information directly. HPN solves this by folding host reachability into BGP, which already runs across the whole fabric. Every ARP entry a ToR learns for its directly attached hosts is redistributed as a host route in BGP. If a NIC-to-ToR link fails, the corresponding ARP entry and its host route are withdrawn, and the rest of the network automatically reconverges onto the surviving ToR's route, exactly as BGP is designed to do when any route disappears.

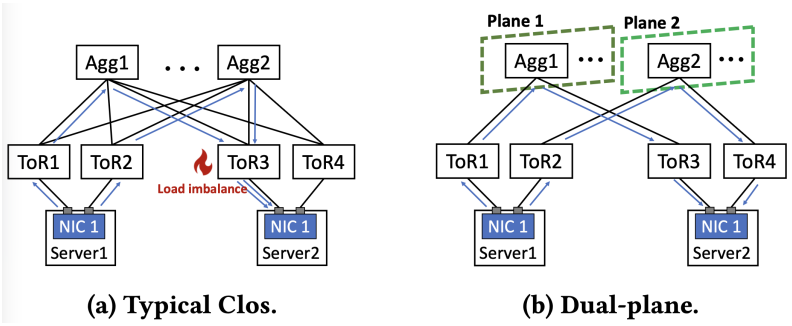


Figure 7.15: Dual-plane tier2 design; Image credit [Qian et al., 2024]

Dual-Plane Design

LLM training clusters need to scale to tens of thousands of GPUs while keeping the network's path-selection problem tractable. The more tiers a topology has, and the more switches participate in load balancing at each tier, the larger the search space becomes for finding good, disjoint paths for a handful of big flows. Therefore, most networks stick to a three-tier topology, consisting of the ToRs, aggregation switches, and core switches. Wiring dual-ToR segments to aggregation switches in the ordinary way, however, reintroduces exactly the hash-polarization problem the rest of the design tries to avoid. Because each host's traffic is split across only two ToRs, while dozens of aggregation

switches sit above them, the downstream direction sees a heavy many-to-two convergence of traffic (Figure 7.15). HPN's answer is a dual-plane topology, in which the two ToR switches of every dual-ToR pair are assigned to two disjoint groups of aggregation switches, called Plane 1 and Plane 2, and each ToR only ever links upward into its own plane. Once a flow enters one of the ToR's uplinks, its entire path through the pod is now fully determined rather than subject to another round of ECMP hashing at the aggregation layer, because there is only one plane it can possibly traverse from that point on. Dual-plane brings a second, more structural benefit beyond load balancing, because each aggregation switch's ports are now split between two planes rather than all serving one flat pool of ToRs, each plane's aggregation switches free up capacity that lets a pod cover many more segments.

Independent Frontend Network

Once the traffic characteristics of distributed training are taken seriously, it becomes clear that a single, undifferentiated network cannot serve both the needs of routine cluster operations and the needs of training. Modern ML data centers therefore typically split their network into two physically separate networks — a frontend network and a backend network. The frontend network behaves much like a traditional cloud data center network. It carries management traffic, orchestrates job scheduling, loads datasets and container images, and writes and reads model checkpoints to and from a shared storage cluster. Because this traffic looks statistically like ordinary cloud traffic, a conventional three-tier Clos topology is a perfectly good fit for it. The backend network, by contrast, exists for one purpose only: to carry the enormous, bursty, synchronized flows of gradients, activations, and parameters that GPUs exchange with one another during training.

Rail-based Connectivity

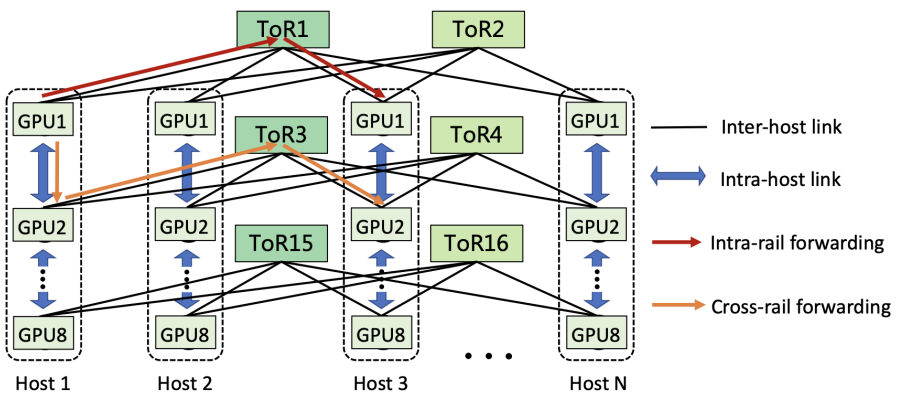


Figure 7.16: Rail-optimized network under dual-ToR, showing intra-rail and cross-rail forwarding; Image credit [Qian et al., 2024]

Since a single pod of ToR and aggregation switches already connect close to 15,000 GPUs and most of the training traffic, HPN deliberately keeps the third tier of core switches somewhat thin. Most traffic generated by tensor or data parallelism usually resides within this pod itself, and only small amounts of traffic traverses the core, e.g., the point-to-point send/receive traffic of pipeline parallelism when a job's pipeline-parallel stages are mapped onto different pods. The one pattern that breaks this locality is the AllToAll communication of Mixture-of-Experts (MoE) models under expert parallelism, where tokens may route to any expert on any GPU, producing dense, rank-agnostic traffic, exactly what a spine layer seems built for.

In order to further optimize traffic flow within a pod, HPN adopts the concepts of "rails" when connecting GPUs within a pod. In the rail-based network design popularized by NVIDIA, all GPUs of the same rank, e.g., the GPU0 in all nodes, connect to the same ToR switch, and can communicate directly with each other over a single hop, without traversing multiple hops over the slower data center network. This ensures a high-bandwidth path between a subset of GPUs belonging to the same rank on different nodes, in addition to the high-bandwidth path that already exists between GPUs of the same node. The rest of the system is then designed to exploit such high-bandwidth paths effectively. For example, if GPU0 on node0 needs to send data to GPU1 on node1, the data can travel on the rail of GPU0 to GPU0 on node1, and then hop to GPU1 over the fast intra-node link (or take the intra-node hop first and then the rail). Either way, the slow part of the path stays on a single rail. Figure 7.16 illustrates both intra-rail and cross-rail forwarding.

7.3 Collective Communication

As we have seen so far, training and serving large machine learning models rarely involves a single accelerator working in isolation. Gradients must be summed across data-parallel replicas, activations must be exchanged across tensor-parallel shards, and experts must receive tokens routed to them from across the cluster. Almost all of this GPU-to-GPU communication is implemented not through raw point-to-point sends and receives, but through collective communication libraries (CCLs). A CCL is a piece of software that efficiently implements a small set of well-defined communication patterns, so that application developers do not need to hand-roll these patterns themselves on top of raw networking primitives.

The reason CCLs matter so much in machine learning systems is that these patterns appear over and over again, and a naive implementation of any of them can easily leave most of the available network bandwidth unused and lead to poor application throughput. A good CCL implementation instead reasons carefully about the underlying hardware topology, chooses an appropriate algorithm for the size of the transfer, and overlaps communication with computation wherever possible. The most widely used CCL for GPU clusters is NVIDIA's NCCL (pronounced "Nickel"), though the same collective patterns are implemented by other libraries, such as oneCCL, RCCL, and MSCCL, and the design of these libraries remains an active area of systems research. Before discussing how CCLs are implemented efficiently, it is worth reviewing what each of these standard collective primitives actually does.

7.3.1 Collective Communication Primitives

Let us recall the collective communication primitives introduced in Chapter 6. Each collective operation below is defined over a group of participating processes, or ranks, each of which typically corresponds to one GPU.

In a Broadcast (Figure 7.17), a single sender rank holds a piece of data, and that same data is copied to every other rank in the group. Broadcast is used, for instance, to distribute initial model weights from one GPU to all other GPUs at the start of training, ensuring every replica starts from an identical state.

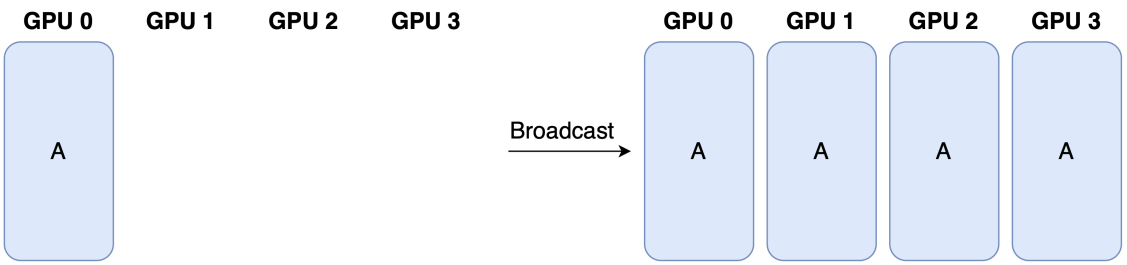


Figure 7.17: Broadcast

In a Scatter (Figure 7.18), one rank holds a buffer that is logically divided into equal chunks, one chunk per rank, and each chunk is sent to its corresponding rank. Unlike Broadcast, every receiver ends up with a different piece of the original data rather than a full copy. Scatter is useful when a single process has partitioned a dataset or a set of parameters, and needs to hand out one partition to each worker.

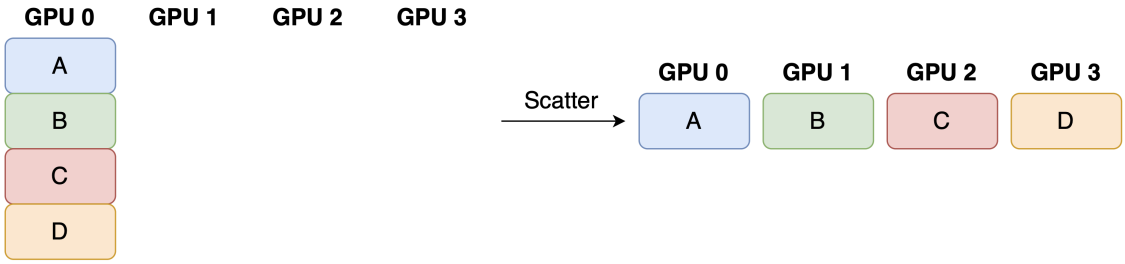


Figure 7.18: Scatter

Gather (Figure 7.19) is the inverse of Scatter — every rank holds a chunk of data, and all of these chunks are collected into a single buffer on one designated rank. Gather is commonly used when results computed independently on each device, such as partial statistics or validation outputs, need to be assembled in one place for logging or further processing.

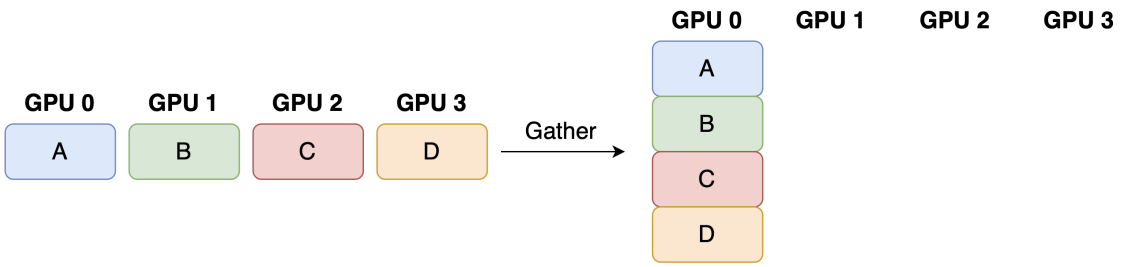


Figure 7.19: Gather

AllGather (Figure 7.20) behaves like a Gather, except that the concatenated result ends up on every rank rather than on just one. Each rank contributes its own chunk, and every rank receives the full concatenation of all chunks from all ranks. AllGather is central to techniques such as fully sharded data parallelism, where each GPU holds only a shard of a weight tensor and must reconstruct the full tensor before use.

A Reduce (Figure 7.21) operation takes a value from every rank, applies an associative reduction operator such as sum, max, or product element-wise across all of these values, and writes the single reduced result to one designated rank. Reduce is useful whenever a global quantity, such as a total loss, needs to be computed from per-device partial values but only needs to be known by one process.

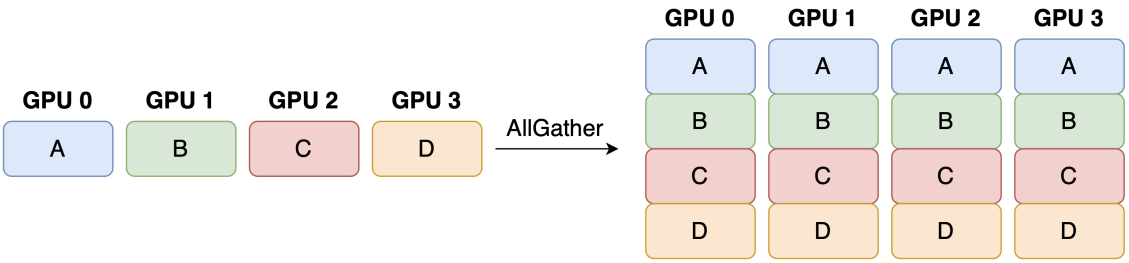


Figure 7.20: AllGather

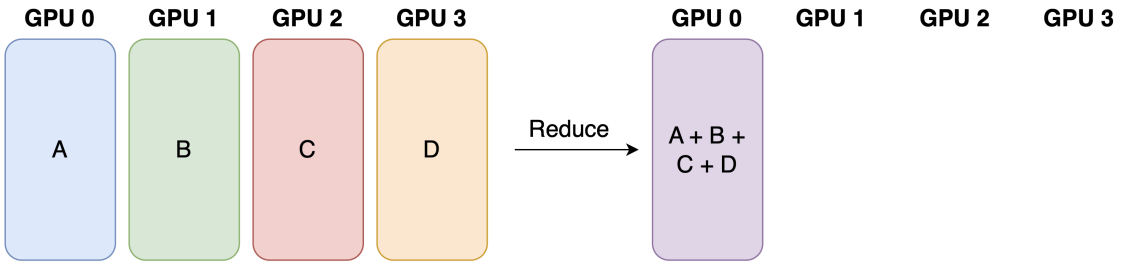


Figure 7.21: Reduce

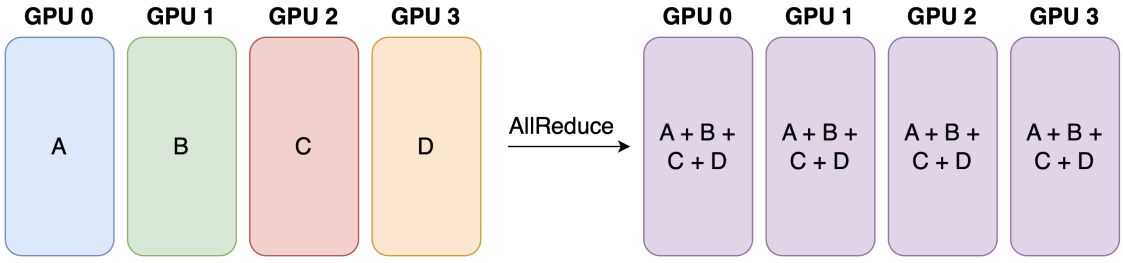


Figure 7.22: AllReduce

AllReduce (Figure 7.22) applies the same element-wise reduction as Reduce, but the resulting reduced value is delivered to every rank rather than only one. This is the single most important collective operation in data-parallel training — after each device computes gradients on its own shard of data, an AllReduce sums those gradients across all devices and ensures every device ends up with the identical summed gradient, so that all replicas apply the same parameter update.

ReduceScatter (Figure 7.23) combines a reduction with a Scatter. Conceptually, the input buffers on all ranks are reduced element-wise as in an AllReduce, but instead of every rank receiving the full reduced result, each rank receives only its own chunk of the reduced output. ReduceScatter is also used directly in sharded training schemes such as ZeRO, where each GPU only needs to retain the slice of the reduced gradient corresponding to the parameters it owns.

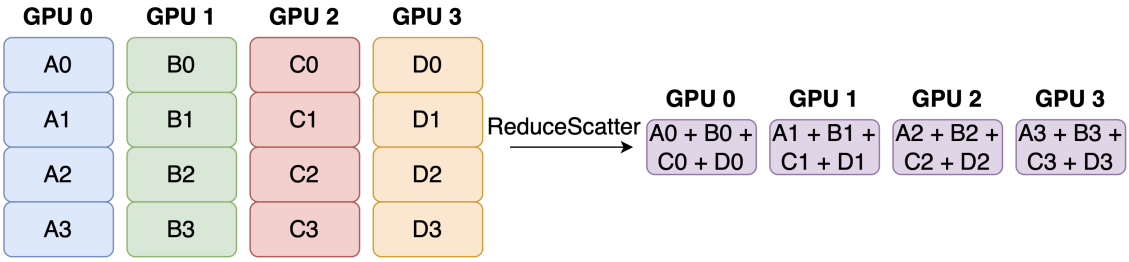


Figure 7.23: ReduceScatter

In an AllToAll (Figure 7.24), every rank has a separate chunk of data prepared for every other rank, and after the operation, every rank ends up with the chunks that all other ranks prepared for it. It is the most general of these collectives, since every pair of ranks exchanges a distinct piece of data. AllToAll is the workhorse of Mixture-of-Experts models, where tokens computed on one device must be routed to whichever device hosts the expert they were assigned to, and the resulting expert outputs must later be routed back.

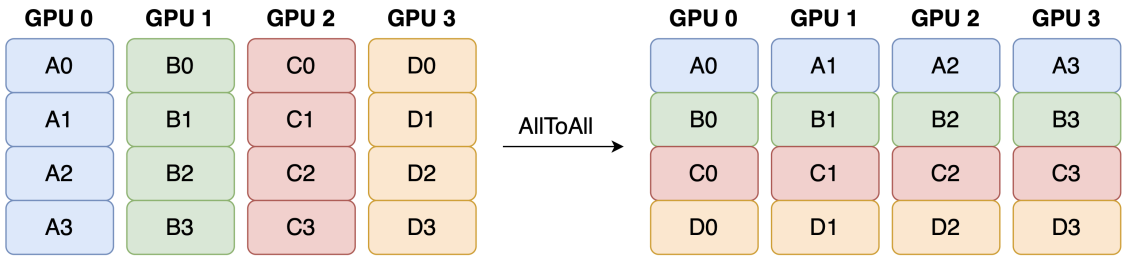


Figure 7.24: AllToAll

7.3.2 The NCCL API

Having described what these collectives compute, we now look at how a programmer actually invokes them. NCCL's design goal is to make using a collective as simple as allocating memory for the data to be sent and received, and then invoking the corresponding API call.

Setting up Communicators

Every NCCL call operates on a communicator, an opaque handle representing one GPU's membership in a group of ranks that will participate together in future collective operations. Before any data can move, every GPU that will take part in a job must construct such a handle. NCCL supports two ways of doing this, and the choice between them depends on whether all participating GPUs are visible to a single host process or are spread across multiple processes, possibly on different machines.

When a single process can see all the GPUs it wants to use, for instance, a single-node job with all GPUs attached to one host, the simplest option is `ncclCommInitAll`, which constructs every communicator for the entire group in one call.

Code 7.2: NCCL Communicator for Single Process

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

This single call internally creates four communicator objects, one per device listed in `devs`, all of which know about each other and belong to the same communication group. It is convenient, but it only works when one process owns every GPU involved, which is not the case in most multi-node clusters, where each node typically runs its own process, or even one process per GPU.

In the more general and far more common case, each participating GPU is driven by a separate process, and these processes may not even share memory or run on the same machine. Here NCCL splits communicator construction into two steps. First, exactly one process generates a unique identifier for the group and distributes it to every other participating process. Once every process holds the same `ncclUniqueId`, each process independently calls `ncclCommInitRank`, passing in the total number of ranks in the job, the shared identifier, and its own rank number.

This call is blocking in the sense that it will not return on any process until every rank in the group has called it and NCCL has established the underlying connections between all of them, whether over NVLink, PCIe, or the network fabric. After this call returns, every process holds a communicator handle that refers to its own GPU's membership in the group, and this handle is what every subsequent collective call will take as an argument. In practice, this initialization happens exactly once at the start of a job and the resulting communicator is reused for the job's entire lifetime, since rebuilding communicators is comparatively expensive and only necessary if the set of

participating ranks changes.

Code 7.3: NCCL Communicator for Multiple Processes

```
ncclUniqueId id;
if (myRank == 0) ncclGetUniqueId(&id);
// broadcast id to all other ranks

ncclComm_t comm;
cudaSetDevice(myLocalGpu);
ncclCommInitRank(&comm, nRanks, id, myRank);
```

Issuing a Single Collective

Once a communicator exists, invoking a collective follows a uniform pattern regardless of which specific operation is being performed — allocate a send buffer and a receive buffer on the GPU, fill the send buffer with whatever local data is being contributed, and call the corresponding NCCL function, passing the communicator and a CUDA stream on which the operation should be enqueued.

Code 7.4: Issuing a Collective

```
double *d_send, *d_recv;
cudaMalloc(&d_send, N * sizeof(double));
cudaMalloc(&d_recv, N * sizeof(double));
// fill d_send with this rank's local data

ncclAllReduce(d_send, d_recv, N, ncclDouble, ncclSum,
              comm, stream);
```

Here, `ncclAllReduce` takes the send buffer (`d_send`), the receive buffer (`d_recv`), the number of elements (`N`), the data type of those elements (`ncclDouble`), the reduction operator to apply (`ncclSum`), the communicator identifying which group of ranks participates (`comm`), and the stream on which to enqueue the operation (`stream`). Because this call is enqueued onto a CUDA stream rather than executed synchronously on the host, it returns almost immediately. The actual data movement and reduction happen asynchronously on the GPU, and the result is only guaranteed to be valid in `d_recv` once the stream has completed the execution, typically checked with a subsequent `cudaStreamSynchronize` or by enqueuing dependent kernels on the same stream. This asynchronous, stream-based execution is what allows NCCL communication to be overlapped with unrelated computation happening concurrently on the same GPU.

Every other collective follows this exact same calling convention, differing only in which buffers

are required and what arguments are meaningful for that operation. A Broadcast, for example, needs to know which rank is the root of the operation, since only that rank's buffer contains meaningful data at the start.

Code 7.5: Issuing Other Primitives

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

A plain Reduce is similar to AllReduce but additionally takes a root argument (`root`), since only the root rank's receive buffer will hold a valid result. AllGather takes a send buffer holding this rank's own chunk and a receive buffer large enough to hold the concatenation of every rank's chunk, with no root argument at all, since every rank ends up with an identical, complete result. ReduceScatter is the mirror image — the send buffer must be large enough to hold contributions covering the entire output space, while the receive buffer only needs to be large enough to hold this rank's own chunk of the reduced result. NCCL does not expose a single dedicated call for Scatter, Gather, or AllToAll in the way it does for the operations above. Instead, these patterns, along with any other custom communication pattern, are built out of NCCL's point-to-point primitives, `ncclSend` and `ncclRecv`.

Code 7.6: Send and Receive

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

Grouping Calls Together

NCCL calls are asynchronous with respect to the host — they enqueue work on a CUDA stream and return. The catch is that a collective needs every rank to participate before any data can move. When a single thread drives several GPUs, it must issue one call per device, and a call for one device may have to wait on the other devices' calls (for example, during communicator setup or when NCCL internally synchronizes the ranks). If the other calls are issued later by the same thread, this wait never ends — the first call waits for calls that the thread cannot issue until the first call returns. This is a deadlock. Even when it does not deadlock, NCCL may serialize the launches device by device instead of running them together.

`ncclGroupStart` and `ncclGroupEnd` fix this by changing when NCCL acts. Inside a group, each call only records the operation. Nothing is launched until `ncclGroupEnd`, at which point NCCL has seen the calls of all devices and launches them together. The same idea applies to point-to-point

operations — the sends and receives of an AllToAll must be grouped, since a `ncclSend` cannot finish until its matching `ncclRecv` is posted.

Code 7.7: Grouping Calls

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

Within this block, each `ncclAllReduce` only registers the operation. Once `ncclGroupEnd` is reached, NCCL launches all of them together, so the devices run concurrently.

Destroying Communicators

Once a job no longer needs a communicator, whether at the end of training or before rebuilding communicators for a different group of ranks, the handle should be released with `ncclCommDestroy`, which frees the internal state and connections associated with that communicator.

Code 7.8: Destroying a Communicator

```
ncclCommDestroy(comm);
```

Taken together, these four pieces, communicator initialization through `ncclCommInitAll`, single collective calls, grouping through `ncclGroupStart` and `ncclGroupEnd` for multi-GPU or multi-call coordination, and cleanup through `ncclCommDestroy`, make up essentially the entire surface that an application developer needs to know to use NCCL correctly. What remains hidden behind this simple interface, and what we discuss next, is how NCCL decides, for a given collective, message size, and hardware topology, exactly which sequence of point-to-point transfers to schedule underneath these calls in order to make full use of the available interconnect.

7.3.3 Challenges in Implementing Collectives Efficiently

While the API surface of NCCL is deliberately simple, implementing these collectives efficiently on real hardware is far from trivial, and the bulk of the engineering and research effort in this space goes into answering three interrelated questions.

The first question is which GPUs should communicate with which other GPUs, and in what topology. A naive implementation of AllReduce might have every GPU send its data directly to

every other GPU, but this is rarely necessary or efficient. Depending on how GPUs are physically interconnected, whether through NVLink within a server, PCIe, or a network fabric across servers, the CCL must decide on a communication topology, such as a ring, a tree, or a hierarchical combination of the two, that makes good use of the available links without creating contention.

The second question is how much data to transfer in each individual step of the algorithm. Sending very small chunks minimizes the time before the first useful bytes arrive at their destination, which matters for latency-sensitive, small-message collectives, but it also means that per-transfer fixed costs, such as kernel launch and network header overheads, are paid many times over. Sending very large chunks amortizes these fixed costs and drives up achieved bandwidth, but delays the arrival of the first useful data and can make it harder to pipeline computation with communication. Practical CCLs therefore choose the algorithm by message size — for small, latency-bound messages they prefer a tree, which needs only about $\log K$ steps, while for large, bandwidth-bound messages they prefer a ring, which keeps every link busy at the cost of more steps.

The third question is how to ensure that these data transfers are fully overlapped with computation and never become a blocking bottleneck. Because collectives are enqueued on CUDA streams alongside other kernels, a well-designed CCL implementation pipelines communication so that, for example, the gradients for early layers of a network can begin their AllReduce while gradients for later layers are still being computed, rather than waiting until the entire backward pass has finished before any communication starts (as we saw in Chapter 6). Achieving this kind of overlap requires careful coordination between the collective's internal chunking, the order in which the underlying kernels execute, and the scheduling of compute kernels issued by the application.

7.3.4 Communication Topologies

The three questions raised at the end of the previous section, namely which GPUs talk to which other GPUs, how much data moves per step, and how communication overlaps with computation, are answered differently by different CCL implementations, but almost all of them begin by imposing some fixed topology on the set of participating GPUs rather than letting every GPU talk to every other GPU directly. Two topologies dominate in practice — the ring and the tree, and understanding why each one is used, and when, is essential to understanding how collectives such as AllReduce are actually implemented underneath the simple NCCL API described above.

In a ring topology, every GPU is logically arranged into a single cycle, so that each GPU has exactly one predecessor and one successor, and data only ever flows from a GPU to its immediate neighbour in the ring (Figure 7.25). The ring can be unidirectional, where data only ever flows in one direction around the cycle, or bidirectional, where two independent rings are formed running in opposite directions, effectively doubling the usable bandwidth at the cost of some additional bookkeeping. Because every GPU only ever sends to and receives from one other GPU at a time, a ring makes extremely efficient use of point-to-point link bandwidth and never creates contention at any single device, but it comes at the cost of latency, since information originating at one GPU must hop through every other GPU in the ring before the collective can complete.

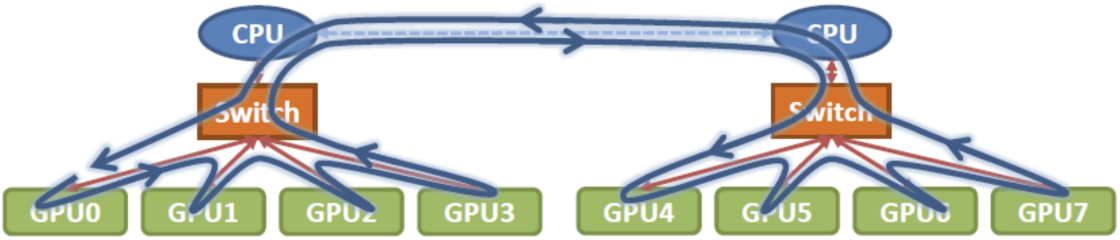


Figure 7.25: Ring topology; Image Credit: [Woolley, 2015]

A tree topology instead arranges GPUs into a hierarchy, where each GPU other than the leaves has one or more children below it and a single parent above it, up to a designated root at the top of the tree. Unlike a ring, where data must pass through every device sequentially, a tree allows the depth of the topology to grow only logarithmically with the number of participating GPUs, since each level of the tree can double, or more generally multiply, the number of devices reached. This makes trees substantially more attractive for latency-bound collectives, particularly when the number of participating ranks is large or when the message being communicated is small, since the number of hops from any leaf to the root scales as the logarithm of the number of ranks rather than linearly with it. NCCL, in fact, often constructs two trees simultaneously, sending half the data down one tree and half down another in the opposite orientation, so that every link in the underlying hardware topology is used in both directions and no GPU sits idle while it is not acting as a relay.

The choice between a ring and a tree, or some hierarchical combination of the two that uses rings within a node and trees across nodes, is not arbitrary. It depends on the size of the message being communicated, the number of ranks participating, and the physical characteristics of the interconnect joining them. Modern CCL implementations, including NCCL, therefore do not commit to a single topology for every call. Instead, they measure or estimate the cost of several candidate algorithms for a given message size and rank count, and dynamically select whichever topology minimizes the expected completion time for that particular collective invocation. Having introduced the two topologies at a high level, we now examine in detail how each one is used to implement the single most important collective in data-parallel training, AllReduce.

7.3.5 AllReduce over a Ring

The most common way of implementing AllReduce on GPU clusters is to decompose it into two simpler collectives executed back to back over a ring: first a ReduceScatter, and then an AllGather. This decomposition is not merely a convenient way of describing the algorithm, but it is in fact exactly how NCCL and most other CCLs implement AllReduce internally, because it turns out to be bandwidth optimal.

To see how this works, suppose there are K GPUs arranged in a ring, and each GPU begins

with a local buffer of data that is logically divided into K equal chunks. Figure 7.26 shows this setup for four GPUs, each holding four chunks, labeled so that GPU 0 holds chunks a_0, a_1, a_2, a_3, GPU 1 holds chunks b_0, b_1, b_2, b_3, and so on. The goal of the overall AllReduce is for every GPU to end up holding the element-wise sum of all four buffers, that is, every GPU should end up with $a_0 + b_0 + c_0 + d_0$ in its first slot, $a_1 + b_1 + c_1 + d_1$ in its second slot, and so forth.

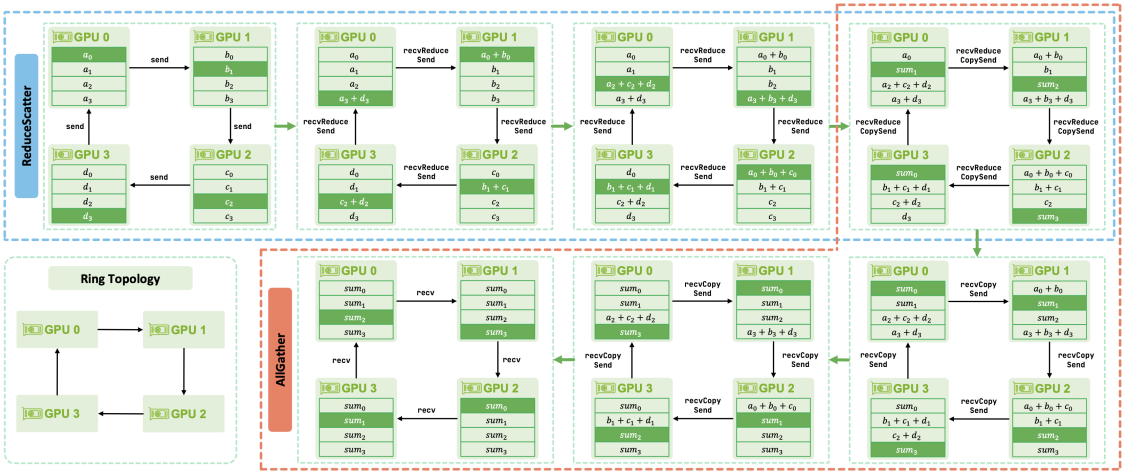


Figure 7.26: AllReduce over a ring, decomposed into a ReduceScatter phase followed by an AllGather phase; Image Credit: [Hu et al., 2026]

The first phase, the ReduceScatter, proceeds in a sequence of $K-1$ steps. At every step, each GPU sends one chunk of its buffer to its successor in the ring, and simultaneously receives a chunk from its predecessor. Crucially, whenever a GPU receives a chunk, it does not simply store it, but rather adds the incoming chunk to the corresponding chunk it already holds, and then forwards this partially accumulated sum onward to its own successor at the next step. In the first step, GPU 0 sends its chunk a_0 to GPU 1, GPU 1 sends b_1 to GPU 2, and so on around the ring, so that after this step, for instance, GPU 1 holds $a_0 + b_0$ in the slot that originally held b_0. In the second step, this partially reduced value is forwarded again, so GPU 2 receives $a_0 + b_0$ from GPU 1 and adds it to its own c_0, producing $a_0 + b_0 + c_0$. This process of receiving a chunk, reducing it into the locally held chunk, and sending the result onward, commonly referred to as a `recvReduceSend` step, continues for a total of $K-1$ steps. After exactly $K-1$ such steps, every GPU ends up holding one chunk that has accumulated contributions from every GPU in the ring, that is, one chunk of the buffer is now fully reduced on each GPU, while the remaining chunks on that GPU are still only partially reduced. This is precisely the result that ReduceScatter is defined to produce — each rank ends up with a different, fully reduced chunk of the output, and no rank has the complete reduced result.

Once the ReduceScatter phase has completed and each GPU holds exactly one fully reduced

chunk, the second phase, the AllGather, begins. This phase also takes $K-1$ steps and also proceeds around the same ring, but now each GPU simply forwards whatever fully reduced chunk it holds to its successor, without performing any further reduction, and the successor copies this chunk into the appropriate slot of its own buffer rather than adding to it. This step is commonly referred to as a `recvCopySend`. After $K-1$ such copy-and-forward steps, every fully reduced chunk that started out on some GPU has now propagated all the way around the ring and has been copied into the corresponding slot of every other GPU's buffer. Figure 7.26 shows this complete two-phase process end to end for a four GPU ring, culminating in every GPU holding an identical buffer containing the complete elementwise sum $\text{sum}_i = a_i + b_i + c_i + d_i$ for every chunk index i.

It is worth pausing to count exactly how much data moves during this algorithm, since this is what makes the ring decomposition attractive in the first place. Each of the two phases takes $K-1$ steps, so the entire AllReduce takes $2(K-1)$ steps in total, and at every step, every GPU sends and receives a chunk of size N/K, where N is the total size of the buffer being reduced. This means that every GPU sends a total of $2(K-1)\cdot N/K$ elements over the course of the algorithm, which for large K approaches $2N$, independent of the number of GPUs participating. This is the key property that makes the ring algorithm bandwidth optimal — as more GPUs are added to the ring, the amount of data each individual GPU must send does not grow, but rather converges to a constant multiple of the original buffer size. So the algorithm scales well to very large numbers of devices as long as the ring itself can be embedded efficiently in the underlying hardware topology, for instance, by following NVLink connections within a node and network links across nodes.

The cost of this bandwidth efficiency is latency. Because information must physically pass through every one of the K GPUs in the ring before the algorithm can complete, the total number of steps grows linearly with K, and each step incurs some fixed per-step overhead regardless of how small the chunk being transferred is. For a very large number of ranks, or for collectives operating on very small messages where this fixed per-step cost dominates the actual data transfer time, this linear dependence on K can become the bottleneck. It is precisely this weakness of the ring algorithm that motivates the alternative tree-based approach discussed next.

7.3.6 AllReduce over a Tree

Where the ring algorithm decomposes AllReduce into a ReduceScatter followed by an AllGather, the tree algorithm instead decomposes it into a plain Reduce followed by a plain Broadcast, both of which are executed over a tree-shaped topology rather than a ring. The motivation for this alternative decomposition is to address the weakness of the ring algorithm identified above, namely that the number of communication steps in a ring grows linearly with the number of participating GPUs. A tree, by contrast, has a depth that grows only logarithmically with the number of GPUs, so the number of sequential steps required is significantly smaller once the rank count becomes large, at the cost of not achieving quite the same asymptotic bandwidth efficiency as the ring for very large messages.

Consider the four GPU example shown in Figure 7.27, arranged into the tree topology, where

GPU 0 is the root, GPU 2 is a middle node, and GPU 1 and GPU 3 are leaves attached to GPU 2. Each GPU begins holding its own local value, denoted a, b, c, and d for GPUs 0 through 3 respectively, and as with the ring example, the goal is for every GPU to end up holding the sum of all four values.

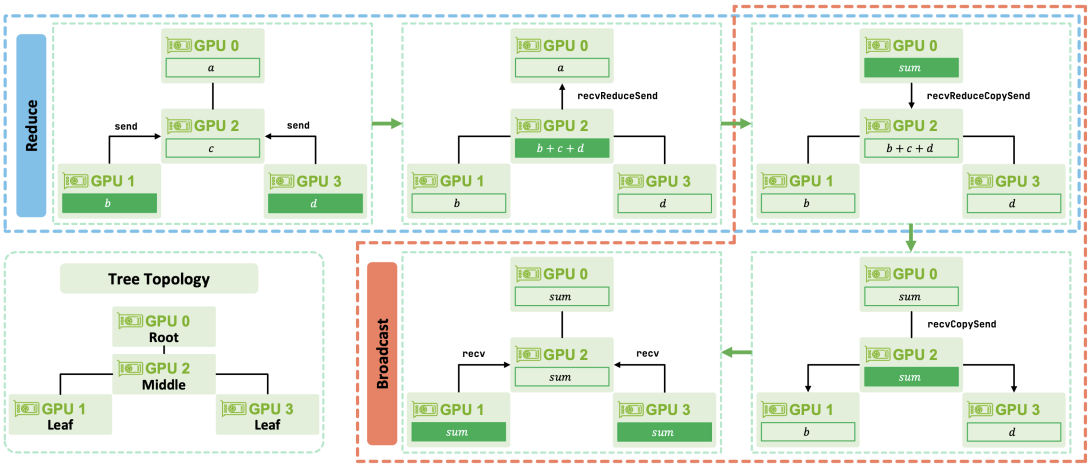


Figure 7.27: AllReduce over a tree, decomposed into a Reduce phase that accumulates values up to the root, followed by a Broadcast phase that distributes the result back down to every leaf; Image Credit: [Hu et al., 2026]

The first phase, the Reduce, proceeds upward from the leaves of the tree toward the root. In the example, GPU 1 and GPU 3, being leaves, each send their local value directly to their parent, GPU 2, which receives both incoming values and adds them to its own, producing the partial sum $b + c + d$ at GPU 2. This is exactly analogous to the `recvReduceSend` step from the ring algorithm, except that here a single GPU may receive from, and reduce contributions from, multiple children at once rather than from a single predecessor. Once GPU 2 has accumulated this partial sum from its own subtree, it forwards the result upward to its own parent, the root GPU 0, which adds it to its own local value a, producing the fully reduced sum $a + b + c + d$ at the root. At the end of this phase, the complete reduced result exists at exactly one GPU, the root, while every other GPU in the tree still holds only its own original, unreduced value or a partial sum. This is exactly the result that a plain Reduce collective is defined to produce.

The second phase, the Broadcast, reverses this process, sending the now fully reduced sum back down from the root to every other GPU in the tree. The root, GPU 0, sends the reduced sum down to its child, GPU 2, which simply copies this value, since no further reduction is needed at this stage, and then forwards it onward to its own children, GPU 1 and GPU 3. Once this phase completes, every GPU in the tree, from the root down to every leaf, holds an identical copy of the fully reduced sum, which is exactly the combined effect of a Reduce followed by a Broadcast, and therefore the

result that AllReduce is defined to produce.

The key advantage of this approach over the ring becomes clear when considering how the number of communication steps scales with the number of participating GPUs. In a ring of K GPUs, both the ReduceScatter and AllGather phases require $K-1$ sequential steps each, so the total step count grows linearly with K. In a tree, however, the number of sequential steps required for either the Reduce or Broadcast phase is bounded by the depth of the tree, and a tree can always be constructed with depth proportional to $\log K$ rather than K itself, for instance by making the tree binary or by allowing each internal node to have several children. This logarithmic scaling is what makes tree-based algorithms attractive in the regime where the ring algorithm struggles — when the number of participating GPUs is large, or when the message being reduced is small enough that the fixed per-step latency, rather than achieved bandwidth, dominates the total completion time.

This latency advantage does not come for free. Unlike the ring, where every GPU is always either sending or receiving exactly one chunk at every step and no link ever sits idle, a tree concentrates traffic near the root, since the root and other internal nodes must handle data flowing from or to every GPU in their subtree, while leaf nodes are comparatively lightly loaded. For very large messages, this imbalance means a tree typically cannot sustain the same aggregate bandwidth utilization as a ring. This is why NCCL does not use a single fixed algorithm for every AllReduce call, but instead selects between ring-based and tree-based implementations, and sometimes hierarchical combinations of the two, based on the message size and the number of ranks involved, using rings when large messages make bandwidth the binding constraint, and trees when a large rank count or a small message makes the number of sequential communication steps the binding constraint instead.

7.3.7 AllReduce on Multiport Direct-Connect Networks

The ring and tree algorithms above assume that a GPU talks to one peer at a time. Many large accelerator clusters do not look like that. They are built from direct-connect networks, where every chip is wired straight to a few neighbors with no switch in between, and each wire is a separate port that can send and receive at the same time. A good algorithm for such a network should use all of these ports in every step. In this subsection we look at why this is hard and at one recent idea, called Trivance [Juerss et al., 2026], that shows how much a collective can gain by using both directions of a link at once.

Direct-Connect Topologies and the Cost of Distance

The most common direct-connect design is the torus. A one-dimensional torus is simply a ring in which every node has a left and a right neighbor. A two-dimensional torus arranges nodes in a grid whose rows and columns each wrap around, so every node has four neighbors, and in general a D-dimensional torus gives every node $2D$ ports. As we saw in Chapter 3, Google's TPUv4 systems are an example — chips are connected into a three-dimensional torus, and optical switches can reconfigure which chips belong to the same torus.

Tori are cheap and simple to build, which is why they are attractive for machine learning clusters.

Their weakness is that they offer less bisection bandwidth than a switched Clos network. Two nodes that are far apart in the torus must send data across many intermediate links, and these links are shared with other messages. For this reason the distance between communicating peers matters a lot. A message sent over a distance of four hops occupies four links, and if several such messages overlap, they slow each other down. An AllReduce algorithm for a torus must therefore care about two things at once — how many steps it takes, and how far its messages travel in each step.

A Cost Model with Congestion

To compare algorithms, we need a simple model of time. The classic choice is the Hockney model, in which sending m bytes between two nodes costs

$$C(m) = \alpha + m\beta,$$

where α is a fixed startup latency and β is the time per byte, the inverse of the link bandwidth. This model has no notion of sharing, so we extend it with a congestion factor. Suppose an algorithm runs in several steps, and in step t, each node sends a message of size m_t, while in the worst case, c_t messages share the same link. The completion time is then

$$T = \text{steps} \cdot \alpha \; + \; \beta \sum_t m_t \, c_t$$

The first term is the latency cost and grows with the number of steps. The second term is the transmission delay. It depends not only on how much data is sent, but also on how many messages collide on a link. Two algorithms that send exactly the same number of bytes can have very different transmission delays if one routes its messages over long, overlapping paths.

This gives us three separate properties to ask about. An algorithm is latency optimal if it uses the fewest possible steps, bandwidth optimal if each node sends the fewest possible bytes, and transmission-delay optimal if the congestion-weighted sum in the above equation is as small as possible. The ring algorithm of the previous section is bandwidth optimal and, on a ring, also transmission-delay optimal, but it needs $2(K-1)$ steps. The tree reduces steps but is not bandwidth optimal. In general it is hard to win on all three at once, and which one matters most depends on the message size.

How Few Steps Are Possible?

Let us ask how quickly an AllReduce can finish on K nodes. At the start, each node knows only its own data. At the end, each node must hold the sum of the data of all K nodes. To measure progress, we define the coverage of a node as the number of nodes whose data are already included in its current value. The coverage starts at 1 and must reach K.

In each step, a node receives values from its peers and adds them to its own value. A received value covers at most as many nodes as the node itself does, because every node has taken part in the same number of steps. Suppose a node has one port, so it can receive from only one peer per

step. Then its coverage can at most double in each step. After s steps the coverage is at most 2^s, and we need $2^s \geq K$, so at least $\log_2 K$ steps are required. For $K = 81$ we have $2^6 = 64 < 81 \leq 128 = 2^7$, so at least 7 steps are needed. Classic algorithms such as recursive doubling reach this bound.

A node on a ring has two ports, one for each direction, and can receive from its left peer and its right peer in the same step. It then adds two received values to its own, so the coverage can at most triple in each step. After s steps the coverage is at most 3^s, and a ring of K nodes needs at least $\log_3 K$ steps. For $n = 81$ we have $3^4 = 81$, so 4 steps are enough, compared to 7 with one port. In general, a node with $2D$ ports can multiply its coverage by $2D + 1$ per step.

Most existing algorithms do not use this. They are designed for one port, and the second port is used by running a copy of the same algorithm in the opposite direction on half of the data. The messages become smaller, but the number of steps stays the same. Other algorithms reach the smaller step count, but they send all data in one direction, so one direction is overloaded while the other is idle. We want both properties together: few steps and an even load on all links.

Tripling the Distance in Both Directions

We first show how Trivance works on a ring of nine nodes, numbered 0 to 8 (Figure 7.28). Initially, each node holds only its own value. In the first step, the distance is 1. Node 0 receives from nodes 8 and 1 and adds both to its own value, so it holds the sum of $\{8, 0, 1\}$. Every other node does the same, so each node now holds the sum of three consecutive nodes centered on itself. In the second step, the distance is 3. Node 0 receives from nodes 3 and 6, which hold the sums of $\{2, 3, 4\}$ and $\{5, 6, 7\}$. Together with $\{8, 0, 1\}$, these cover all nine nodes exactly once. Node 0 now holds the total sum, and so does every other node. Two steps suffice, whereas one port would need $\lceil \log_2 9 \rceil = 4$.

The example generalizes to a simple rule — in step k, every node exchanges its current value with the two nodes at distance 3^{k-1}, one on each side, so the distances are $1, 3, 9, \ldots$. After k steps, a node holds the sum over a group of 3^k consecutive nodes centered on itself. The two peers at distance 3^k hold groups of the same size that start exactly where the node's own group ends, one on each side. The three groups neither overlap nor leave a gap, so their sum covers 3^{k+1} nodes with no data counted twice. A ring of 3^s nodes is therefore finished in s steps. Every node sends the same amount in both directions, so all links carry the same load. Trivance needs no extra wiring — each node stays connected only to its direct neighbours, and a message to a node at distance 3^{k-1} is forwarded hop by hop through the intermediate nodes.

Adding Both Messages Together

In the two-copy approach, each copy receives one message per step and adds it to its own partial result. The two copies never combine their results. In Trivance, the two messages that arrive in the same step are both added to the node's own value, as in the example above, and the single combined value is sent onward in the next step in both directions. This is what allows the coverage to triple instead of double. This has a cost and a benefit. The cost is that a node must wait for both messages before it can start the next step, so one slow message delays the whole node. The benefit is that the

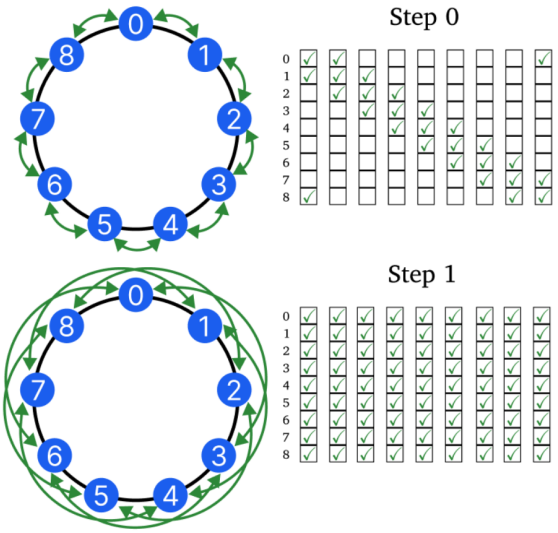


Figure 7.28: Communication pattern of Trivance on a ring of nine nodes; Image Credit: [Juerss et al., 2026]

data from both sides is merged into one value, so less addition is needed overall. The total amount of addition is about one quarter smaller than in the two-copy approach.

Other Network Sizes and Higher Dimensions

The nine-node example worked nicely because nine is a power of three. For other sizes, the algorithm runs the regular tripling steps as long as possible and then adds one smaller final step for the nodes that are still missing. For example, in a ring of seven nodes, a node knows three nodes after the first step and needs four more. It gets two from each side by talking to peers two hops away.

On a torus with D dimensions, each node has $2D$ ports, two in each dimension. A single ring collective would use only two of them at a time. To keep all ports busy, we split the vector into D equal parts and run one collective on each part. Each collective performs its steps along one dimension at a time, moving to the next dimension after it has finished a full pass along the current one. The collectives start in different dimensions, so in every step each dimension is used by exactly one of them. Each collective sends in both directions of its dimension, so all $2D$ ports are in use in every step and no two messages compete for the same link. Each message also travels only within a single small ring, so the distances stay short. In theory, one could use all ports of a node within one single pattern, but this needs very particular network shapes, and real clusters often reserve different dimensions for different kinds of parallelism. Running one ring-style collective per dimension is the more practical choice.

Practice Problem 7.1

Consider training a two-layer MLP with data parallelism across K nodes. The network architecture is as follows: (i) Layer-1: weight matrix $W_1 \in \mathbb{R}^{d\times h}$ and bias $b_1 \in \mathbb{R}^h$; (ii) Layer-2: weight matrix $W_2 \in \mathbb{R}^{h\times o}$ and bias $b_2 \in \mathbb{R}^o$, where d is the input dimension, h is the hidden dimension, and o is the output dimension.

Each node has a full replica of the model and processes a local mini-batch. After each backward pass, every node holds its own gradient tensor, and the nodes must synchronize them via an AllReduce, so that every node ends up with the sum of all K gradient tensors before applying the parameter update. Let G denote the size in bytes of the gradient tensor of a single node. Every node starts with G bytes and ends with G bytes. Throughout this problem, assume every parameter and gradient is stored as a 1 byte float value. The volume of data transferred across links means the number of bytes sent over the links, where each transfer is counted once, at the sender.

(i) Find the number of gradient scalars (biases and weights) in the gradient tensor of a single node. Since each scalar takes 1 byte, this number is G, and we use it in all the upcoming parts.
Solution: The total number of gradient scalars is the sum of all weights and biases across both layers: $G = (dh + h) + (ho + o) = dh + ho + h + o$. Every parameter has exactly one gradient, and since each scalar is 1 byte, G is also the size of the gradient tensor in bytes.

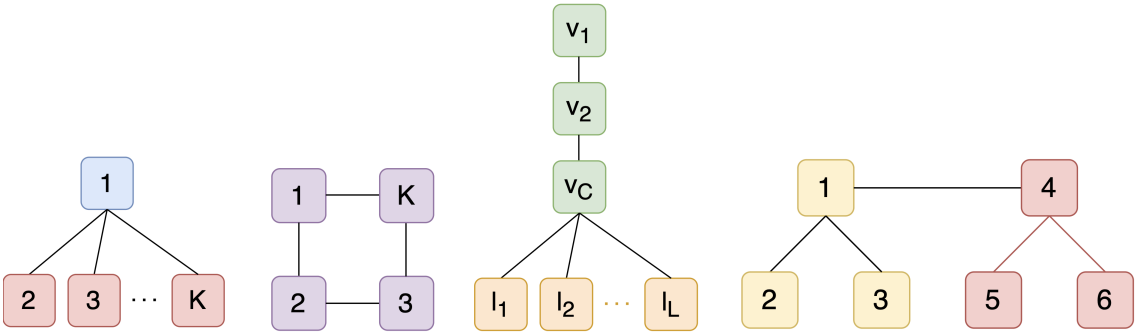
(ii) In the simplest scheme, one of the K nodes acts as a parameter server (PS). Every other node sends its gradient tensor to the PS; the PS accumulates them, then broadcasts the result back to every worker (Figure 7.29a). What is the total volume of data exchanged across all the links (counting both the directions)?
Solution: Each of the $K-1$ workers sends G bytes to the PS, and the PS broadcasts G bytes back to each of the $K-1$ workers. Counting both directions, the total volume is $2(K-1)G$ bytes. Note that all of this traffic passes through the PS's links, which makes the PS a bottleneck.

(iii) Now, arrange the K nodes in a logical ring (Figure 7.29b). There are two phases in AllReduce. In the first phase, each node passes $\frac{1}{K}$-th chunk of gradients to its right neighbour, and accumulates partial sums. In the second phase, each node passes its fully reduced chunk to its right neighbour. How many bytes does each node send and receive in a single step (a phase has multiple steps)?
Solution: In a ring of K nodes, the gradient tensor of G bytes is split into K chunks of G/K bytes each. In every step, each node sends one chunk to its right neighbour and simultaneously receives one chunk from its left neighbour. Therefore, in a single step, the bytes sent plus received by a node are $\frac{G}{K} + \frac{G}{K} = \frac{2G}{K}$.

(iv) For the previous part, what is the total volume of data sent and received by one node over the entire AllReduce operation?
Solution: The first phase (reduce-scatter) takes $K-1$ steps and the second phase (all-gather) also takes $K-1$ steps, so there are $2(K-1)$ data transfer steps in total. Each step moves $\frac{2G}{K}$ bytes (sent plus received) per node, so the total volume of data sent and received by one node is $2(K-1)\times\frac{2G}{K} = \frac{4(K-1)G}{K}$ bytes. Equivalently, each node sends $\frac{2(K-1)G}{K}$ bytes and receives the same amount.

(v) What is total data transferred across all the links over the entire AllReduce operation?
Solution: There are K nodes, and all of them send the same amount of data. Since each transfer is counted once, at the sender, the total volume is $K \times \frac{2(K-1)G}{K} = 2(K-1)G$ bytes. This matches the parameter-server volume, but here the traffic is spread evenly over all the links, so there is no single bottleneck node.

(vi) Now, consider the following topology — there are C chain nodes, $v_1, v_2, \ldots, v_C$ arranged in a linear chain with links in between consecutive nodes. Also, v_C has L leaf children $\ell_1, \ldots, \ell_L$, where $L = K - C$ (Figure 7.29c).



(a) Parameter server (b) Ring (c) Chain (d) Double binary tree

Figure 7.29: Different topologies

All of the nodes hold a full model replica. AllReduce proceeds in two phases: (i) gradients flow up from the leaves through the chain to v_1 (partial sums being accumulated across nodes) and (ii) the final global sum flows back down from v_1 through the chain to the leaves. What is the total volume of data sent across all the links?

Solution: In the first phase, each of the L leaves sends G bytes to v_C, contributing LG bytes, and each of the $C-1$ chain links (from v_C up to v_1) carries one accumulated tensor of G bytes, contributing $(C-1)G$ bytes. In the second phase, each of the $C-1$ chain links carries G bytes back down, contributing $(C-1)G$ bytes, and v_C sends G bytes to each of the L leaves, contributing LG bytes. The total volume is $2LG+2(C-1)G = 2G(L+C-1) = 2G(K-1)$ bytes, using $L+C=K$. This is consistent with the fact that the topology is a tree with $K-1$ links, each carrying G bytes once in each direction.

(vii) A double binary tree uses two complete binary trees T_1 and T_2, together spanning all K nodes (Figure 7.29d). For simplicity, assume each of T_1 and T_2 has (2^H-1) nodes, where H is the number of levels in each tree, so that $K = 2(2^H-1)$. The AllReduce proceeds as follows: (i) gradients move up to the roots in both the trees (partial sums being accumulated across nodes); (ii) the subtree sum gradients are exchanged between roots of T_1 and T_2 and they are accumulated at both the root nodes individually; (iii) the sums flow back across both the trees. What are the total number of steps needed for this entire process? Count every data transfer as a separate step; compute need not be counted.

Solution: Each complete binary tree with 2^H-1 nodes has H levels, so a leaf is $H-1$ edges away from its root. In phase (i), gradients move up one level per step and both trees proceed simultaneously, so this takes $H-1$ steps. In phase (ii), the exchange between the two roots can be counted as a single step because the two directions overlap. In phase (iii), the final sums flow back down both trees in parallel, again taking $H-1$ steps. So, the total number of steps is $(H-1)+1+(H-1) = 2H-1$.

(viii) For the previous part, what is the total volume of data sent across all the links?

Solution: If the root is at level 0, there are 2^ℓ nodes at level ℓ. Moving from level ℓ to level $\ell-1$, each of these nodes sends G bytes to its parent, so $G \times 2^\ell$ bytes cross that level. Summing over all levels, the data transferred in phase (i) for a single tree is $\sum_{\ell=1}^{H-1} G \times 2^\ell = G(2^H-2)$ bytes. Phase (iii) is symmetric

to phase (i), so across both trees, phases (i) and (iii) together contribute $4G(2^H - 2)$ bytes. Phase (ii) contributes $2G$ bytes, since each root sends G bytes to the other. Hence, the total data transferred is $4G(2^H - 2) + 2G = 4G(2^H - 1) - 2G = 2G(2(2^H - 1) - 1) = 2G(K - 1)$ bytes, using $K = 2(2^H - 1)$.

(ix) State one advantage of using double-tree topology over a simple uni-directional ring?
Solution: While the total amount of data is the same ($2G(K - 1)$) for both topologies, the number of steps taken under the double-tree topology is $2H - 1 = O(\log K)$, whereas it is $2(K - 1) = O(K)$ for the ring. Therefore, the time taken to complete the operation is much lower for the double tree, especially for large K or small messages where per-step latency dominates.

Programming Assignment 7.1: Ring AllReduce

In this assignment, you will implement the core of Ring AllReduce. Each simulated GPU runs as its own Python process and communicates with its neighbours in a logical ring using `multiprocessing.Queue` objects. You need to complete the computation inside `gpu_worker` in two phases — ScatterReduce (where each GPU passes partial sums around the ring and accumulates them, so that GPU g ends up with the fully-reduced value for slot g) and AllGather (where each GPU circulates its fully-reduced value so that every GPU ends up holding the global sum in every slot). You must work out the correct send and receive slots at each step using modular arithmetic, and use only the provided `_send` and `_recv` helpers. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/ring_allreduce

Programming Assignment 7.2: Tree AllReduce

In this assignment, you will implement Tree AllReduce. Seven simulated nodes, arranged as a fixed complete binary tree, each run as their own Python process and communicate through `multiprocessing.Queue` objects. You need to complete the communication logic inside `tree_worker` in two phases — Reduce (where values travel bottom-up, with each node summing the contributions of its children before passing the partial sum to its parent, so that the root obtains the global sum) and Broadcast (where the root sends the global sum top-down so that every node ends up holding it). You must handle the different roles of leaves, internal nodes and the root, and use only the provided `_send` and `_recv` helpers. The assignment, along with a detailed `README` is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/tree_allreduce

7.4 Transport and Communication Protocols

In the previous sections, we described the communication patterns of distributed training, the networks that carry them, and the collective operations built on top of them. A collective operation is an abstract pattern. A group of GPUs holds some data, and after the operation, each GPU holds a specified result. In this section, we study how this pattern turns into bytes on the wire. We first look at the RDMA transport that today's NICs run in hardware, and at how a collective library such as NCCL packages data, splits work into channels, and pipelines transfers on top of it. We then see where the hardware transport falls short for ML, and how a transport written in software can leverage the tolerance of ML to errors, many network paths, and better congestion control to improve performance. Finally, we see how these ideas must be adapted when we have clusters of many thousands of GPUs.

It helps to see GPU communication as a stack of layers. The training application calls an operation such as `AllReduce()`. Below it, a collective communication library (NCCL for NVIDIA GPUs, RCCL for AMD GPUs) decides what data to send to whom. It turns a collective into a ring, a tree, or another topology, and then into point-to-point transfers between pairs of GPUs. Below the library, the transport decides how each transfer crosses the network — how a message is split into packets, which path each packet takes, how fast the sender may transmit, and how lost data is detected and resent. At the bottom are the network interface cards (NICs) and the network. For an ordinary internet application, a clean interface such as a TCP socket separates the job of deciding what to send from the job of moving it. In an ML cluster the boundary is blurrier, because part of the transport runs on the GPU, part in the NIC hardware, and part on the CPU. Many ideas in this section come from letting the layers share information.

7.4.1 RDMA Transport and Communication Protocols

NCCL is the most widely used collective library for NVIDIA GPUs, and its design shows many general ideas. RCCL for AMD GPUs follows very similar principles.

How Does a Collective Move Data?

As we saw before, RDMA lets a NIC read from and write to memory without involving the operating system, and GPUDirect lets this happen directly with GPU memory. Let us follow an AllReduce over this machinery. The application calls the collective, and the library launches a reduction kernel on every participating GPU. The library keeps a set of transport buffers in GPU memory, and the kernel copies data between these buffers and the application's tensor buffers. It also does the arithmetic of the collective, such as summing values that arrive from several senders. Since data passes first through a transport buffer and then into the tensor, this design is called a two-stage copy.

On the sending side, the CPU asks the NIC to write data into the receiver's transport buffer, one chunk at a time. On the receiving side, the NIC places arriving data directly into the GPU transport buffer, and the receiving CPU watches a completion flag to learn that a chunk is ready. The GPU

kernel then consumes the chunk. This work is shared across GPU, NIC, and CPU. The GPU moves and reduces data, the NIC moves bytes over the network, and the CPU coordinates this transfer. This division of labor is the key to understanding both NCCL and the software transports of the next subsection.

Transports in NCCL

NCCL supports several transports, that is, mechanisms that carry data between two GPUs, and picks among them based on the hardware. Inside a server, the main choices are the P2P transport (which uses direct GPU-to-GPU access over NVLink or PCIe), the SHM transport (which stages data through shared host memory), and the NVLS transport (which uses the multicast and reduction features of NVLink switches). Between servers, the main choices are the NET transport (which uses InfiniBand, RoCE, or ordinary TCP sockets), and the COLLNET transport (which uses switches that can perform part of the collective in the network). Communication that uses GPUDirect, such as P2P and GPUDirect RDMA, is driven by the streaming multiprocessors (SMs) of the GPU. Socket communication is driven by CPU cores. This is one reason GPUDirect paths are faster — they use the many cores of the GPU and avoid copies through host memory.

Reliable and Unreliable Modes

When using the NET transport over RDMA, the network transport itself runs in the NIC hardware. The transport functionality implemented by NIC is dependent on the type of connection, called the queue pair (QP) type. RDMA offers three main types, shown in Table 7.1.

	Reliable Connection (RC)	**Unreliable Connection (UC)**	**Unreliable Datagram (UD)**
Reliability and retransmission in NIC	yes	no	no
Congestion control in NIC	yes (may be switched off on some NICs)	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

Table 7.1: RDMA connection types from the point of view of a software transport

With a reliable connection (RC), the NIC does the whole job of splitting messages into packets, numbering them, checking acknowledgments, resending lost packets, and controlling the sending speed. An unreliable connection (UC) still splits a large message into packets and puts them back

together at the receiver, but it neither resends lost packets nor controls congestion. An unreliable datagram (UD) is the simplest. Each operation mode sends a single packet, which may go to any destination, and the NIC does no splitting or reassembly. The last row of the table becomes important in the next subsection, when we build a transport in software. Reliability requires a way to recover from loss. Many RDMA NICs use a simple scheme called go-back-N. The receiver accepts packets only in the correct order. For example, if packet 5 is lost, the receiver throws away packets 6, 7, 8, and so on, even though they arrived correctly, and the sender must go back and resend everything starting from packet 5. A better scheme is selective retransmission, where only the lost packet is resent.

Congestion Control

Congestion control is the mechanism that makes senders slow down when network receives data faster than it can send, leading to queuing. Congestion control needs information about the network, and there are four common signals. ECN marks are set by switches when a queue becomes longer than a threshold. RTT measurements show how long a packet and its acknowledgment take, and a growing RTT means growing queues. Packet trimming lets an overloaded switch forward only the header of a packet it cannot buffer, so the receiver learns exactly what was lost. Packet loss is the last resort, a clear but late signal.

In sender-driven control, the sender sets its own rate or window from these signals. The most common algorithm in RDMA data centers is DCQCN (Data Center Quantized Congestion Notification). It works in three steps. First, a switch whose queue is getting long marks the packets that pass through it. Second, the receiver sees the marks and sends a small warning packet back to the sender. Third, the sender cuts its rate sharply, and then raises it again slowly in small steps.

In receiver-driven control, the receiver is in charge. It grants credits to senders, and a sender may transmit only as much as it has credits for. The receiver issues credits at a rate that matches its own link, so traffic arriving at the last hop never exceeds what the link can carry. This keeps queues short without waiting for congestion signals. EQDS is a well-known design of this kind and is used by the Ultra Ethernet standard. The receiver keeps its senders in three lists, with the highest priority for senders that suffered loss, then active senders, then idle ones, and it hands out credits in that order.

How often should congestion control react? Reacting to every packet is the most precise, but it is costly and can overreact to noise. Even hardware transports usually update their rate once per RTT, which is tens of microseconds in a data center.

Flow Control

Congestion control protects the network as a whole. Flow control has a narrower goal — a sender must not send more than the next device, or the receiver, can take in. RDMA networks apply flow control at two levels. At the link level, losses are expensive for a go-back-N transport, so RDMA networks usually try to be lossless with Priority Flow Control (PFC). With PFC, a switch whose

buffer is about to overflow sends a pause message to the device that feeds it, and that device stops sending until the buffer has space again. This prevents drops. At the library level, the two-stage copy gives a natural form of flow control. The receiver tells the sender that a transport buffer is ready by sending a short "clear-to-send" message, and the sender transmits only into announced buffers. In this way the receiver paces the sender.

Protocols: Simple, LL, and LL128

The NIC transport also decides how packets cross the network. On each link, NCCL makes one more choice — the communication protocol. A communication protocol defines how data is packaged on the wire and how the receiver learns that data has arrived. NCCL supports three protocols — the choice is a trade-off between latency and throughput, as shown in Table 7.2.

	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

Table 7.2: Comparison of the NCCL communication protocols

The "Simple" protocol sends data in large chunks. To be sure that the receiver does not read a chunk before it has fully arrived, the sender uses memory fences, which are instructions that enforce the order of memory operations. Fences are expensive, so each hop costs about six microseconds. But since chunks are large, the protocol reaches close to peak bandwidth. It is the best choice for large messages. The "LL" protocol, short for low latency, avoids fences. Each 8-byte unit holds 4 bytes of data and a 4-byte flag. The receiver keeps checking the flag, which is called polling. When the flag has the expected value, it knows the data next to it is valid. No separate synchronization step is needed, so latency is about one microsecond per hop. The price is that half of each unit is the flag, so at most half of the link bandwidth carries data. The "LL128" protocol improves on LL by using the fact that modern GPUs and NVLink move data in 128-byte units. Each unit holds 120 bytes of data and 8 bytes of flag, so about 95 percent of the bandwidth is useful, while latency stays low at about two microseconds per hop.

Flag-based protocols have a limit when data crosses the network. The flag has to be polled, and in practice it can be polled only in CPU memory. Such a protocol therefore cannot use GPUDirect RDMA to place data directly into GPU memory, and data must pass through host memory instead. The extra step makes these protocols slow between servers compared with their speed inside a server. The lesson is that a protocol is a bundle of choices about synchronization, packaging, and data placement, and a choice that is excellent on one kind of link can be poor on another.

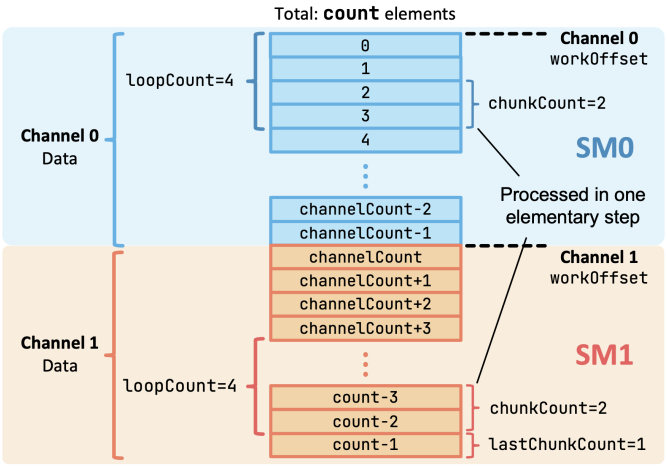


Figure 7.30: NCCL's data partitioning; Image credit [Hu et al., 2026]

Channels

A single stream of data between two GPUs cannot fill a fast link, because one thread block on one SM can move only so much data at a time. NCCL therefore splits a large message into parts and sends them in parallel. Each part is called a channel. Different channels run on different SMs, or on different CPU threads for socket transports. Inside a channel, data is divided further (Figure 7.30). Say the whole message has `count` elements. Each channel is responsible for a contiguous range that starts at the work offset of the channel. The channel processes its range in several loop iterations. In each iteration it handles a group of chunks, and the data processed in one elementary step is one chunk. The last chunk may be smaller, because the range does not always divide evenly.

How many channels should be used? More channels add parallelism, because they use more SMs and more network connections. But too many channels hurt. Each channel needs GPU memory for buffers, takes SMs away from model computation, and cuts data into pieces that are too small to be efficient. NCCL therefore chooses the number of channels dynamically, based on message size, topology, and hardware. In the case study at the end of this section we will see that even this choice can waste memory at very large scale.

Pipelining with Slots

During multihop transfer, NCCL does not wait for a whole message to arrive before it forwards it. Instead, it splits the message into small pieces and passes each piece on as soon as it is received. This idea is called pipelining. To see why it helps, we study a Broadcast along a ring of K GPUs. GPU 0 holds a message of N bytes, and every GPU must receive a copy. We start the ring at GPU 0,

so the data flows along the chain GPU 0 → GPU 1 → ⋯ → GPU $(k-1)$. This chain has $K-1$ links, and each link has bandwidth B.

Suppose each GPU waits until it has received the whole message, and only then sends it to the next GPU. Sending N bytes over one link takes N/B. The message must cross $K-1$ links one after the other, and at any moment only one link is busy. The total time is therefore

$$T_{\text{no pipeline}} = (K-1)\,\frac{N}{B}$$

This time grows with the number of GPUs. It is also wasteful, because while one link works, all the other links sit idle (Figure 7.31a).

Instead, we can split the message into S equal pieces, which we call slots. Each slot has N/S bytes. GPU 0 sends the slots one by one. The key rule is that a GPU forwards a slot to the next GPU immediately after receiving it, without waiting for the other slots. Therefore, while GPU 1 passes slot 1 to GPU 2, GPU 0 can already send slot 2 to GPU 1. This is the same idea as an assembly line in a factory. Each worker does one small job and hands the product on, so all workers are busy at once.

We use time steps to count the total time taken. One step is the time to send one slot over one link, which is

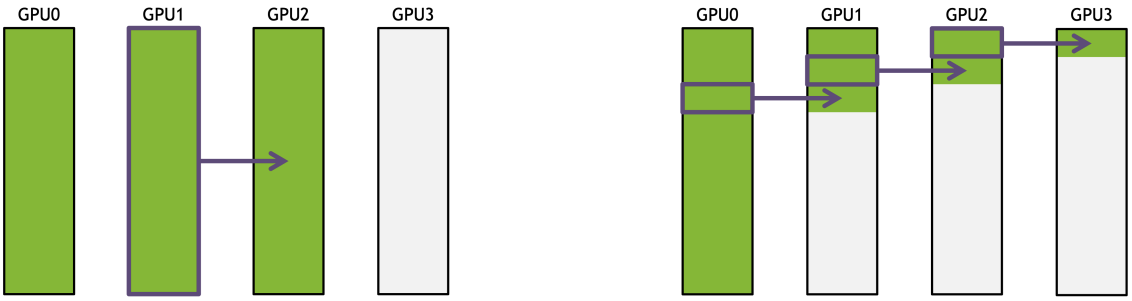
$$t_{\text{step}} = \frac{N}{S\,B}.$$

In every step, each link can carry one slot. Table 7.3 shows an example with $K = 4$ GPUs, so there are 3 links, and $S = 4$ slots. In the table, the first link works alone in step 1, all three links are busy in steps 3 and 4 (the pipeline is full), and the pipeline then drains until the last slot reaches GPU 3 in step 6.

Step	GPU 0 → GPU 1	GPU 1 → GPU 2	GPU 2 → GPU 3
1	slot 1	–	–
2	slot 2	slot 1	–
3	slot 3	slot 2	slot 1
4	slot 4	slot 3	slot 2
5	–	slot 4	slot 3
6	–	–	slot 4

Table 7.3: Which slot crosses which link at each step, for $K = 4$ GPUs and $S = 4$ slots

Generalizing, slot 1 needs $K-1$ steps to reach the last GPU. After that, the remaining $S-1$ slots arrive one per step, because they follow each other one step apart. The total number of steps is therefore $(k-1)+(S-1) = S+k-2$ (Figure 7.31b). Multiplying by the time of one step gives the total



(a) Without slots (b) With slots

Figure 7.31: Pipelining; Image credit [Woolley, 2015]

time with pipelining as

$$T_{\text{pipeline}} = (S + k - 2)\frac{N}{S\,B} = \frac{N}{B}\left(1 + \frac{k-2}{S}\right).$$

In the example ($K = 4$, $S = 4$) this is $1.5\,N/B$ instead of $3N/B$, a twofold speedup. When S is much larger than K, the fraction $(k-2)/S$ approaches zero and the time approaches N/B, the time to cross a single link. With enough slots, the number of GPUs has almost no effect on the broadcast time. Pipelining also lets network transfer overlap with GPU work such as copying and reducing data.

It may seem that we should use as many slots as possible. However, every slot has a fixed cost, for example the cost of synchronization and of starting a transfer. If the slots are very small, this cost becomes larger than the time spent on moving data. Large slots give a long start-up and wind-down phase, as in steps 1–2 and 5–6 of the table, while very small slots waste time on overhead. For this reason, the slot size is a tuned parameter in NCCL.

Kernel Design of Transport

These ideas are implemented in a CUDA kernel with one thread block for each channel. Inside a block, a few warps do initialization, and the other warps manage communication. The threads of a warp work together to move data through the slots in a circular way, so after the last slot is used, the next transfer reuses the first. All threads of a warp perform the same operation at the same time, such as sending, reducing, or copying. The design thus has several levels of parallelism — channels across blocks, slots for pipelining inside a channel, and warps for different tasks inside a block. This keeps communication well overlapped with computation and keeps the links busy. Several parameters, such as the number of channels, the slot size, and the number of warps, are tuned for performance.

7.4.2 Software-Based Transport

The hardware transport of RDMA was designed for general data center traffic such as storage and databases, where there are many small flows that start at different times. As we saw before, ML traffic is different. It is bursty, it has low entropy (only a few large flows, so the network has little variety to spread over many paths), it is synchronized (all GPUs send at the same moment), and its flows often collide on the same link. In this subsection we first study why the hardware transport does badly under such traffic. Then we study ideas that can be implemented in software layers to improve the performance of hardware transport.

Limits of Hardware Transport

RDMA's congestion control protocol DCQCN assumes that traffic consists of many small flows, and that congestion lasts long enough for the warning to travel back and take effect. ML traffic breaks both assumptions. A training job has few but very large flows, and they all start together, so the queue fills within microseconds. By the time the warning reaches the sender, the burst has either already overflowed the buffer or already ended. The sender then cuts its rate for no good reason and takes a long time to recover, which leaves the link half empty. This hurts especially because a collective is only as fast as its slowest transfer, so one flow that is slowed down delays all GPUs. Some operators therefore switch DCQCN off and schedule traffic in the application. This is risky, because a large RDMA network without congestion control can suffer from deadlocks, head-of-line blocking (packets stuck behind other packets that cannot move), and congestion that spreads widely.

Another problem is incast that happens when many senders transmit to one receiver at the same time. The last link before the receiver cannot carry all the traffic, so its queue grows and packets are delayed or dropped. In Mixture-of-Experts inference, for example, some experts are far more popular than others and receive many times the average load, so tokens from many GPUs converge on a few GPUs. Sender-driven congestion control reacts only after the queue has already grown, which is too late for a short burst. Receiver-driven control fits this case much better, since the receiver knows exactly how many senders it has, but commercial RDMA NICs do not offer it.

Further, go-back-N becomes costly at high link speeds. On a 400 Gb/s link, thousands of packets can be in flight at any moment, so a single loss can force the sender to repeat a large amount of data. Go-back-N also has a second problem for ML. If the network spreads the packets of one flow over several paths, they often arrive out of order, and go-back-N treats this harmless reordering as a loss. Hardware that uses go-back-N therefore cannot easily use multipath, which we study later in this subsection. PFC avoids these losses, but it causes new problems. A pause stops all flows on the link, including flows that are not going to the congested destination. The pause can then spread backward from switch to switch, and in the worst case a circle of paused switches waits for each other forever, which is a deadlock. Synchronized ML bursts trigger pauses often, and the problems grow as link speeds increase. If the transport could recover from loss efficiently, the network would depend much less on PFC.

Finally, a large cluster is rarely built at once. It grows over time, and operators mix NIC generations and vendors to control cost and avoid depending on one supplier. All modern NICs support 400 Gb/s RDMA, but their hardware transports differ in details such as reliability and congestion control. When NICs of different kinds talk to each other, these differences can reduce bandwidth by a large factor.

All the problems above could in principle be fixed in the NIC. But hardware changes take years, while ML workloads change every few months. The needs of dense-model training, Mixture-of-Experts, and disaggregated inference were all different, and they appeared in quick succession.

A transport written in software avoids these problems. It can use multiple paths, it can be changed in weeks instead of years, and it behaves the same on every NIC. Before we build one, we look at two properties of ML that such a transport can use, its tolerance to errors and its need for many paths.

Tolerance of ML to Losses

Traditional transports promise that every byte is delivered, in order. This is needed for a file transfer, but is it needed for training? Training with stochastic gradient descent (SGD) is already noisy, because every step uses a random mini-batch, and the algorithm is built to make progress despite noise. So a slightly imperfect network that loses some packets may not harm training, and a network that knows about ML can use this to be faster. Experiments show that when a small fraction of gradient packets is lost, models still converge in the same number of rounds and reach the same accuracy. For example, an LSTM model tolerates a loss rate up to about 0.8%, and a ResNet50 image model up to about 2.5%. Beyond this threshold, convergence slows gradually (Figure 7.32).

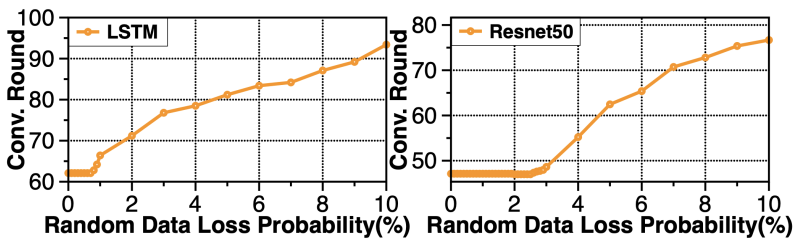
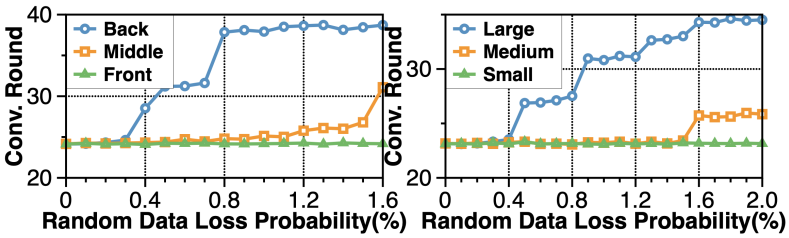


Figure 7.32: Impact of loss on model convergence; Image credit [H. Wang et al., 2024]

The gradient of a layer is a sum over many independent values, and the order in which these values arrive does not change the sum. ML traffic is therefore more tolerant of packet reordering than ordinary traffic as well, as long as the receiver knows where each value belongs. This helps, because reordering is a natural result of sending packets over many paths. Further, not all losses do the same damage. Two observations are useful. The first is about layers. Layers near the input capture simple features and are more tolerant to loss, while layers near the output are more sensitive. The second is about magnitudes. Gradients with large values influence the model more than gradients

with small values, so losing a large gradient hurts more than losing a small one. Both suggest that, when the network must drop something, it should drop the least important packets first (Figure 7.33).



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

Figure 7.33: Gradient losses on different NN layers; Image credit [H. Wang et al., 2024]

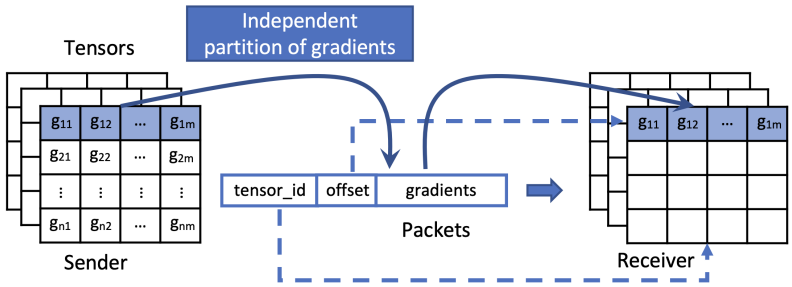


Figure 7.34: Tensor packing and unpacking; Image credit [H. Wang et al., 2024]

A transport that uses these observations has been proposed in prior work [H. Wang et al., 2024] and has three ideas. The first is packing. Gradients are packed into packets together with a tensor identifier, a layer, and an offset (Figure 7.34). Since each packet says where its contents belong, the receiver can unpack it into the right place even if packets arrive out of order or never arrive. The second idea is how losses are treated. If a gradient packet is lost, the receiver fills the missing values with zeros, so they add nothing to the update. If a parameter packet is lost, the receiver keeps the old value of those parameters. In both cases training continues without waiting for a retransmission. The third idea is priority. Each packet gets a priority based on its layer and magnitude, attached as tags. Switches keep several queues, one per priority level, and serve higher-priority queues first. This also speeds up the training pipeline, because the data needed earliest arrives earliest. When congestion occurs, switches use selective dropping. Each queue has a marking threshold, and lower-priority queues have lower thresholds, so low-priority packets are dropped or marked first and high-priority packets are protected (Figure 7.35).

These ideas show how much can be gained by designing the application and the transport together,

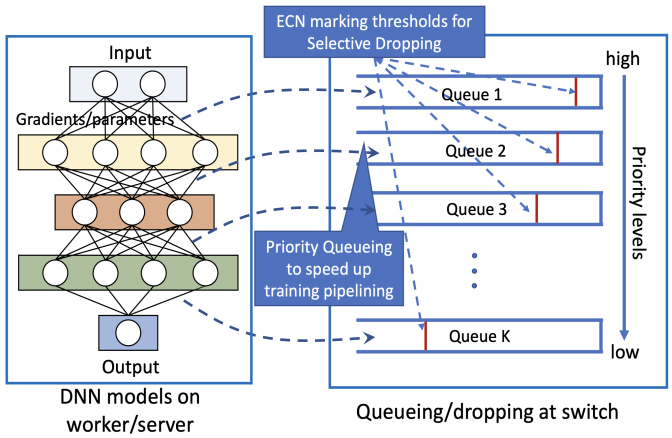


Figure 7.35: Priority queueing and selective dropping; Image credit [H. Wang et al., 2024]

but there are caveats. The loss tolerance depends on the model and the training setup, so there is no universal safe loss rate. The scheme needs support from switches and from the training framework. And it is hard to add to existing NICs, which offer little programmability. This last point is one more reason to make the transport itself easy to change by moving a part of it to software.

Multipath Transport

As we saw before, a connection that uses one path is vulnerable to flow collisions, and one slow flow can delay a whole collective. Some operators answered by redesigning the topology, for example with rail-optimized networks. A different answer [Zhou et al., 2026] works in the transport. Data center networks have many equal-cost paths between two servers, often hundreds, and a multipath transport uses many of them for one connection. Then no single link carries too much of a flow, and a congested path can be avoided.

How can a sender pick a path when switches choose routes by hashing header fields? The sender cannot control the hash, but it can change its inputs. In RDMA, the hash usually includes the source and destination queue pair numbers, so using many queue pairs for one logical connection spreads packets over many paths. For example, 256 queue pairs can give up to 256 paths. Datagram queue pairs can send to any destination, so 16 source queue pairs and 16 destination queue pairs give $16 \times 16 = 256$ paths. For traffic over UDP, the sender can simply vary the UDP port. Sending packets of one connection over many paths on purpose is called packet spraying. With this technique, instead of a flow being stuck on one possibly bad path, each part of the flow takes its own route.

Spraying at random is a good start, but a smarter sender tracks how each path performs. For each chunk, the sender picks two paths at random and sends on the one with the lower recent round-trip time (RTT). This avoids the worst paths without measuring all of them at every step. Keeping state

for every path, such as its latest RTT, needs a lot of memory for a NIC with a small on-chip memory, but is easy in software on a CPU.

With many paths, packets arrive out of order because paths have different delays. A software transport can follow the standard design of TCP. Packets carry sequence numbers, the receiver acknowledges what it has, and a packet is declared lost only after enough later packets have arrived. Since multipath makes reordering common, the threshold of duplicate acknowledgments that triggers a retransmission is set higher than in TCP, to avoid resending packets that are only late. With RDMA, the reordering buffers can live directly in GPU memory, and the NIC writes incoming data to the right place. How the data is then put in order depends on the QP type, as we will see later.

Using hundreds of queue pairs per connection may seem dangerous. NICs have limited on-chip memory for queue pair state, and with too many queue pairs the NIC must swap state in and out of host memory over PCIe. This queue pair swapping is known to hurt small-message workloads. For ML collectives the effect is much milder, for two reasons. First, messages are large and most packets are full sized, so the swapping cost is spread over a lot of data. Second, GPUDirect traffic goes through the PCIe switch and does not compete with the CPU traffic the NIC uses to fetch queue pair state. To limit the count further, all connections between the same pair of NICs share the same set of queue pairs.

Software Congestion Control on Today's NICs

New congestion control ideas are hard to realize in hardware, because the transport logic is baked into the NIC. Further, hardware RDMA NICs consume packet headers themselves, so a software transport on top of them usually cannot see ECN marks or trimming information. It can, however, use NIC timestamps to compute RTT, and it can detect loss. Since latency-based control works well in practice, these limited signals are often enough. With many paths in multipath transport, the sender can keep one state per path in software, such as a window per path, or a single global state for all paths. Experiments show both work well, and the global state is simpler. A software transport also does not need to react to every packet. Since even hardware updates its rate only once per RTT, software can decide at a coarser granularity, such as once per chunk of tens of kilobytes, with little loss in quality.

So, can we realize the gains of software-based congestion control on today's NICs? UCCL-Tran [Zhou et al., 2026] is an example of a software transport that works on top of existing RDMA NICs. Every transport does two kinds of work. The data path moves bytes. It reads the payload from memory, splits it into packets, and writes the arriving packets into the destination memory. The control path makes decisions — how fast to send, which path to take, which packets are lost, and what to resend. In a hardware transport both paths live inside the NIC, so we cannot change the decisions. The key idea of UCCL-Tran is to leave the data path in the NIC, because the NIC is very good at moving bytes, and to move only the control path to the CPU (Figure 7.36). Three requirements follow. First, the NIC should make as few decisions as possible, so that the CPU can make them. Second, the payload must still go directly between the NIC and GPU memory

(GPUDirect), because copying large tensors through the CPU would be too slow. Third, the design should work on NICs from different vendors.

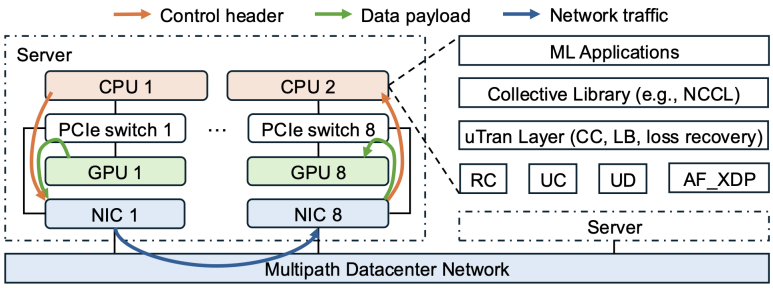


Figure 7.36: A software transport layer sits between the collective library and the NIC primitives; Image credit [Zhou et al., 2026]

The difficulty is that the CPU needs to see the control information of each packet (such as sequence numbers and acknowledgments), but it should not see the payload. The payload is large and lives in GPU memory, while the control information is small and should live in CPU memory. The NIC must therefore deliver the two parts to two different places.

The last row of Table 7.1 ranks the QP types for this purpose. UC is the best starting point. The NIC still splits a large message into packets and puts them back together at the receiver, which saves a lot of CPU work. But it does not run its own reliability or congestion control, so the software is free to do it. If a NIC does not offer UC, the software uses RC with the hardware congestion control switched off. This is less flexible, because the NIC still handles reliability, but it works. If the NIC allows neither, the software falls back on UD, which has no hardware logic at all but also gives no help with splitting and reassembling messages.

With UC (or RC), data is sent with the RDMA write with immediate operation. It does two things at once. It writes a chunk of data into a memory address chosen by the sender, and it delivers a small 32-bit number, called immediate data, to the receiver's CPU (Figure 7.37). The immediate data is delivered together with a completion event, only after the chunk has arrived. So the sender CPU can place its own transport header in these 32 bits, and the receiver CPU receives the header while the payload goes directly to the GPU.

Thirty-two bits are very few, so every bit must be planned. Table 7.4 shows one possible layout. Such a layout limits the number of connections and of messages in flight, but these limits are large enough for collective communication, where the number of simultaneous transfers is small.

With UC and RC, reordering is also simple. Each reordering buffer is a whole chunk, and the sender CPU tells the NIC at which address each chunk must be written, so chunks that take different paths still land in the right place. Some NICs support only datagrams (UD). UD has no write with immediate, so the header cannot travel in the immediate field. A simple idea is to send header

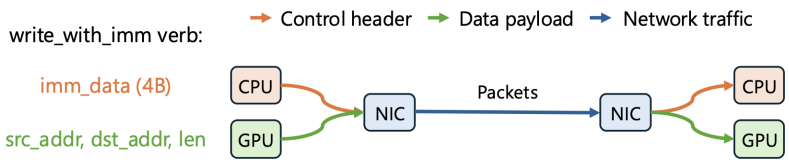


Figure 7.37: Using RDMA write with immediate to separate control header and data payload; Image credit [Zhou et al., 2026]

Field	Bits	Purpose
Connection ID	8	which connection between this pair of NICs
Message ID	7	which message in flight on this connection
Chunk sequence number	8	position of the chunk inside the message
Last-chunk flag	1	marks the end of the message
Reserved	8	room for advanced schemes, such as receiver-driven credits

Table 7.4: Using the 32-bit immediate data as a transport header

and payload together into GPU memory and let the CPU read the header from there. But the CPU reading GPU memory is slow and adds overhead. A better idea uses scatter-gather lists (Figure 7.38). The sender posts a send operation with a list of two memory pieces, a small header in CPU memory and a payload in GPU memory. The NIC reads both and joins them into one packet. At the receiver, the CPU posts a receive operation with a two-piece list as well. The first piece (a fixed number of bytes) is in CPU memory and the rest is in GPU memory. The NIC splits the arriving packet at that boundary. Header and payload travel in one packet, so they are always lost together and arrive in the same order. This is important because the CPU can then trust that a header it sees always describes a payload that arrived.

UD brings a second problem, namely reassembly. Each reordering buffer now holds a single packet. The receiver can post buffers in the correct order, but when packets take different paths they arrive in a different order and land in the wrong places (Figure 7.39). The fix is to let the GPU repair the order. The library already runs a GPU kernel to reduce data from the transport buffer into the tensor, as we saw at the start of this section. The software transport adds a "scattered copy" step to this same kernel. It first copies the packets to their correct positions, using the order given by the CPU, and then does the normal reduction. Because the copy is fused into an existing kernel, no extra kernel launch is needed. The cost is some extra GPU memory traffic, which is limited by the network speed and is small compared with the very high memory bandwidth of a GPU.

A natural doubt is whether a CPU can keep up with control plane processing. A server with eight 400 Gb/s NICs must handle 3.2 Tb/s in each direction, and the next NIC generation will double this.

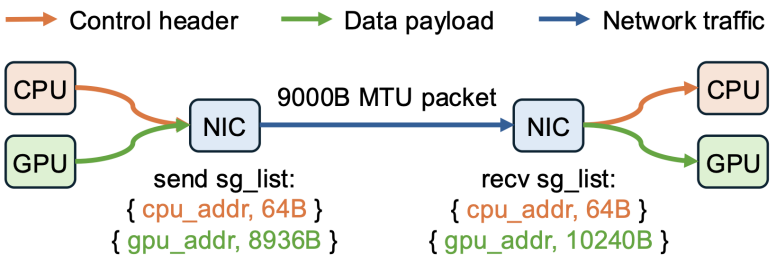


Figure 7.38: Separating control header and data payload with scatter-gather lists on datagram QPs; Image credit [Zhou et al., 2026]

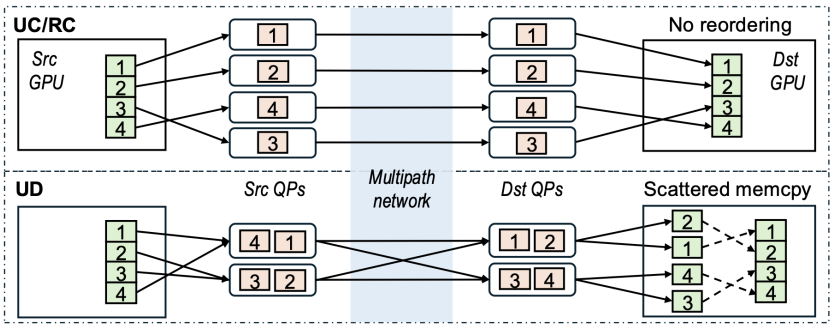


Figure 7.39: Multipathing and reordering buffers in a software transport; Image credit [Zhou et al., 2026]

A software transport that spends thousands of CPU cycles on each packet would need many cores. The design goal of UCCL-Tran is about one CPU core for each 400 Gb/s of one-directional traffic. Several techniques make this possible, and each one reduces the amount of CPU work per byte.

The first technique is to decide how to process traffic per chunk instead of per packet, as discussed for congestion control above. Load balancing and loss recovery can also work on a chunk of several packets that travel together on the same path. With UC or RC, one write operation sends a whole chunk, for example 32 KB, and the NIC splits it into MTU-sized packets. The CPU then handles one event per chunk instead of one per packet.

The second technique is connection splitting (Figure 7.40). One CPU core can handle only a limited amount of traffic. If one connection is always handled by one core, a fast connection is limited by that core, and an unlucky core may be overloaded while others are idle. With connection splitting, the many QPs of a NIC pair are divided among several engine threads. Each thread keeps its own congestion and load-balancing state for its share, which makes it a "sub-connection". The threads of the application then send each new message to the least loaded engine. In this way the work of one connection is spread over several cores, and short imbalances are smoothed out. As a side effect, the sender does not send all messages from one core in a single burst.

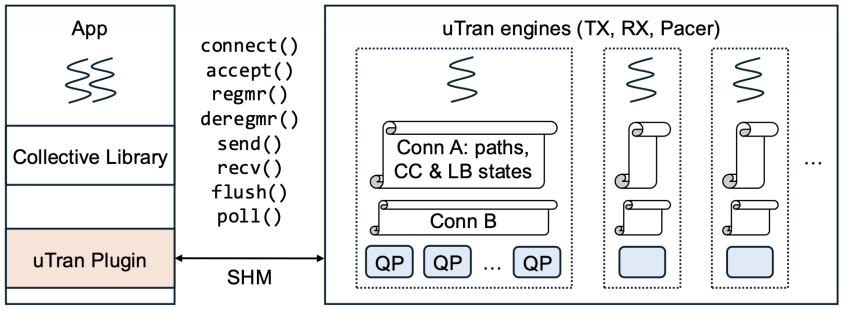


Figure 7.40: Threading model of a software transport, several engine threads share the work of one connection; Image credit [Zhou et al., 2026]

Two further techniques reduce overhead. First, each engine thread runs in a run-to-completion style. It loops over sending, receiving, pacing, timeouts, and retransmission without handing work to other threads or waiting on locks, and a fair scheduler such as round robin shares its time among these functions and the connections. Second, when using UD, every packet needs its own send or receive operation, and each operation normally requires a costly write to the NIC's registers (called an MMIO write). With chained posting, the descriptors of many operations (for example 32) are linked together and the NIC is notified once for the whole chain.

7.4.3 Case Study: Scaling the Communication Stack to Thousands of GPUs

The earlier subsections assumed a cluster that sits in one building, where all GPUs are reached within a few switch hops. Frontier training jobs now use more than one hundred thousand GPUs [Zeng et al., 2026]. At this size, problems appear that are small or invisible in a normal cluster, and some of them lie not in the network but in the communication library itself.

No single building can hold 100,000 GPUs, so the cluster is spread over several buildings, each with several groups of racks ("AI zones"). A multi-stage Clos network connects everything (Figure 7.41). As a result, the latency between two GPUs depends strongly on where they are. Compared with GPUs in the same rack, GPUs in the same zone see about 7× higher latency, GPUs in different zones about 15×, and GPUs in different buildings about 30×.

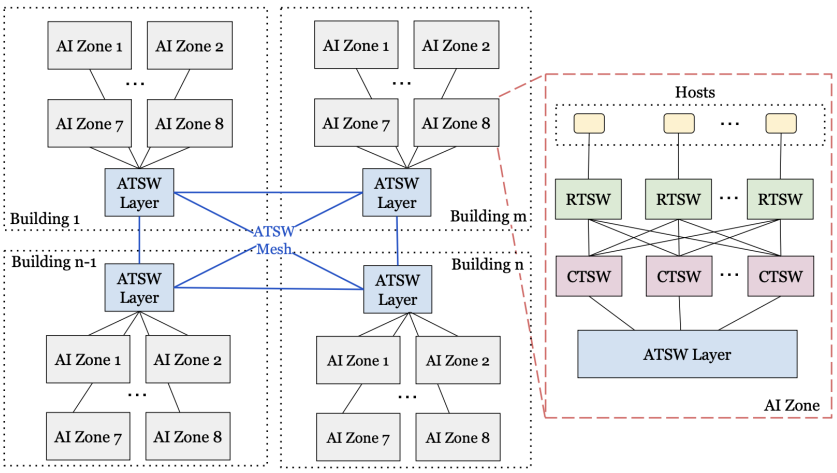


Figure 7.41: Multi-building network architecture; Image credit [Zeng et al., 2026]

Why does this matter for transport? Recall that the bandwidth-delay product (BDP) is the amount of data that must be in flight to keep a link busy:

$$\text{BDP} = \text{bandwidth} \times \text{RTT}.$$

As an illustration, on a 400 Gb/s link, if the RTT is 10 μs, the BDP is 500 KB. If the RTT is 30 times larger, the BDP is about 15 MB. A sender that keeps too little data in flight leaves the long link idle, while a sender that keeps too much fills the switch buffers and creates queues and delay. The correct amount depends on the distance, but the same collective library may talk to a neighbor in the same rack and to a GPU in another building at the same time. A single fixed setting cannot suit both. This is the main transport challenge at this scale.

The transport is not the only part of the stack that struggles. Operations that are cheap in a small cluster, such as starting a job or allocating buffers, also become expensive. Failures are frequent, so

the stack must also be easy to observe and debug. We look at the transport first and then at the collective library.

Matching Data in Flight to the Path

Recall the two-stage copy design from the start of this section, where data passes through small staging buffers and the receiver sends a clear-to-send message before each piece. This naturally splits the data into small pieces and paces the sender, but each clear-to-send message adds a full round trip to the critical path. When the round trip is long, this cost is large. Zero-copy communication using one-sided RDMA removes the staging buffers. The receiver tells the sender once where the destination memory is, and the NIC then writes the data directly. This needs only one control message, which is good for latency.

Zero-copy has a drawback, however. Since there are no small buffers, the whole message is handed to the NIC at once and leaves the sender as one large burst. The receiver has few chances to give feedback, so switch buffers fill up. This is worse for Mixture-of-Experts traffic, which is already bursty. Classic congestion control, such as DCQCN, is hard to tune for a network with such different round-trip times, so we cannot rely on it alone. What we want is the low latency of zero-copy together with the gentle pacing of segmentation.

The solution is to keep zero-copy, but to cut each message into segments inside the library and to limit how many segments are in flight. Three parameters control this — the number of queue pairs used for a destination, the segment size, and the number of unacknowledged messages allowed on each queue pair. Their product is the amount of data in flight, so it should match the bandwidth-delay product of the path. Because the library knows how far away the destination is, it can choose the parameters for each connection. A simple scheme sorts connections into four classes — within the same rack, across racks in the same zone, across zones in the same building, and across buildings. Nearby destinations get a small setting that keeps buffers short, and distant destinations get more queue pairs and more outstanding messages, so that the long link stays full. A smaller segment size also spreads a burst over a longer time.

Keeping Messages in Order

Using several queue pairs for one connection raises a familiar problem. Messages sent on different queue pairs can arrive in a different order than they were sent, but the library must tell the application that a message is complete only when all earlier messages have also arrived. The solution follows the software transport of the previous subsection. Each message gets a sequence number, carried in the immediate data of an RDMA write with immediate together with a flag that marks the last piece of the message. The sender assigns messages to the data queue pairs in round-robin order, and it pauses a queue pair when too many of its messages are still unacknowledged. The receiver keeps the number of the next message it expects. A message that arrives early is stored and waits. When the missing message arrives, the receiver processes it and all consecutive stored messages. A small number of control messages, which carry destination addresses at the start of a collective, travel on

a separate control queue pair.

Starting a Job Quickly

At thousands of GPUs, jobs restart often, because some machine is always failing. The start-up of a communicator is slow for several reasons. Each rank must find its peers, exchange information with all other ranks, and detect the topology. If done in a naive way, the work is serialized, the network is overloaded by many small messages, and some steps need time proportional to the square of the number of ranks. Three ideas remove most of the cost.

First, communicators are created eagerly from one global communicator. Normally, each process group creates its own communicator when it is first used. A large model uses more than ten process groups, so a central key-value store receives a flood of requests, and the same information is exchanged again for each group. Instead, one communicator that contains all ranks is created first, and every smaller group is split from it. The information already exchanged is reused, so creating a subgroup becomes cheap.

Second, the bootstrap phase is made less crowded. In the original design, every rank contacts a server on rank 0 to find its neighbor, and to avoid overloading the server, ranks wait for a time proportional to their rank number before they connect. This wait is the price of a simple design, and it is acceptable even for the last rank, so the staggering is kept but tuned to fit within the restart budget.

Third, work is postponed or reduced. Buffers and connections are not created when the communicator is initialized, but when the first collective runs, and this happens in the background while the application loads data. Together, these changes make initialization about an order of magnitude faster at this scale.

The lesson of this scale is that no single layer can solve its problems alone. The transport needs to know the distance to the destination, which only the library knows. The library must keep its memory small, so that the model can use the GPUs. The tools must be built in from the start, since failures are daily events. As clusters grow beyond one hundred thousand GPUs to regional scale, further changes will be needed, for example start-up that does not require every rank to know every other rank, and diagnosis tools that can follow computation and communication fused into one kernel.

Practice Problem 7.2

Answer the following questions on networking for machine learning workloads.

(i) Which of the following options is/are correct?

(A) Traditional cloud computing traffic is more bursty as compared to ML workloads.

(B) When there are multiple paths of different costs available between a source and destination, Equal Cost Multi Path (ECMP) uses hashing to load balance across all available paths.

(C) ML workloads have lower entropy in the distribution of destination hosts than traditional workloads.

(D) In general, with all else being equal, pipeline parallelism generates larger communication traffic than either tensor or data parallelism.

Solution: (C)

(A) False. ML workloads are the bursty ones, since collective operations (for example, AllToAll) make many flows start and stop together.

(B) False. ECMP only load-balances across paths of equal cost, not across paths of different costs.

(C) True. ML collective operations communicate within a fixed, small set of GPU ranks, so the distribution of destination hosts is highly concentrated.

(D) False. Tensor parallelism needs an AllReduce in every layer across all devices in the group, whereas pipeline parallelism only sends activations between adjacent stages.

(ii) Which of the following options is/are correct?

(A) RDMA is a technology used for intra-node data transfers between GPUs connected to the same server.

(B) In RDMA, the user application and NIC communicate via a single work queue to transfer data and completion notifications.

(C) In a GPUDirect RDMA, the data is directly transferred between GPUs connected to different servers, completely bypassing both the CPUs and NICs.

(D) Dual-ToR and dual-plane connectivity between GPUs lead to better load balancing for low entropy flows, because we no longer need to depend on ECMP hashing.

Solution: (D)

(A) False. RDMA is an inter-node technology; intra-node GPU transfers use links such as NVLink or PCIe.

(B) False. RDMA uses separate work queues (send/receive) and a completion queue, not a single queue.

(C) False. The NIC is still actively involved, because it performs the DMA; only the CPU and host-memory staging are bypassed.

(D) True. Dual-ToR and dual-plane designs provide physically distinct network paths, so low-entropy flows need not rely on ECMP hashing.

(iii) Which of the following options is/are correct?

(A) By the definition of rail-optimized topologies, we connect GPUs of the same rank across hosts through ToR switches.

(B) Both NVLink and PCIe can be used for intra-node GPU data transfers.

(C) NCCL uses pipelining across slots to parallelize the communication work across thread blocks running on different SMs in a GPU.

(D) An AllReduce is equivalent to ReduceScatter followed by AllGather.

Solution: (A), (B), (D)

(A) True. In a rail-optimized topology, GPUs with the same rank on different hosts connect to the same ToR switch (a "rail").

(B) True. GPUs within a node can communicate over NVLink or over PCIe.

(C) False. Parallelization across SMs happens by distributing channels to different thread blocks; pipelining across slots overlaps steps within a channel.

(D) True. ReduceScatter leaves each rank with one reduced shard, and AllGather then distributes all shards to every rank, which gives the AllReduce result.

(iv) Which of the following options is/are correct?

(A) A transport protocol must be less tolerant to reordering of packets for ML workloads.

(B) There is no other way to use RDMA technology to connect multiple hosts except via Ethernet switches.

(C) A transport protocol can be more tolerant to the loss of packets containing gradients, because we can use zeroes instead of old values.

(D) A transport protocol can be more tolerant to the loss of packets containing parameter updates, because we can use zero values in lieu of the lost parameter values.

Solution: (C)

(A) False. ML workloads are more tolerant to packet reordering, since collectives operate on chunks whose placement is known in advance.

(B) False. InfiniBand is also widely used to connect hosts with RDMA.

(C) True. A lost gradient can be replaced with zeros, which just drops that contribution, and training (SGD) is robust to this small noise.

(D) False. Zeroing parameters would corrupt the model; the old (stale) parameter values are used instead.

7.5 Network Job Scheduling

So far in this chapter, we have looked at the network from the point of view of a single training job. We asked what the traffic of one job looks like, how the topology should be wired to carry it, and how collective operations should be implemented on top of that topology. In practice, however, a GPU cluster is rarely used by only one job. Large clusters are shared by tens or hundreds of training jobs that start and finish at different times, and these jobs compete for the same network links. This raises a new question — given that many jobs must share one network, can we decide where each job runs, and when its communication happens, so that the jobs get in each other's way as little as possible? This is the problem of network-aware job scheduling. In this section, we study it through CASSINI [Rajasekaran et al., 2024], a scheduler that was designed to plug into existing cluster schedulers and make them aware of the network.

7.5.1 Why Job Placement Matters for the Network?

A cluster scheduler decides which GPUs each job receives. Existing schedulers make this decision using goals like fairness between users, overall throughput, or short completion times. Many of them also try to reduce network overhead, but they do so in a simple way — they place the workers of one job as close together as possible, for example under the same rack, so that the job's traffic crosses as few links as possible. This is a sensible rule, but it does not solve the whole problem. Jobs arrive and leave all the time, so the free GPUs in a cluster become scattered over many racks, a situation known as fragmentation. As a result, a job often cannot be packed neatly into one rack, and its traffic must cross links that are also used by other jobs. No scheduler can promise a perfectly packed placement at all times, unless it keeps migrating jobs to defragment the cluster, which is expensive in its own right.

Once two jobs share a link, the question then becomes how they share it. In an RDMA cluster, the congestion control algorithm decides this. As we saw before, DCQCN is widely used, which tries to give each competing flow a fair share of the link. Fairness sounds like a good property, but as we show next, it can be exactly the wrong behavior for jobs whose traffic comes in synchronized bursts.

Training Traffic Is Periodic

Recall from the beginning of this chapter that training traffic is bursty. There is a second property of this traffic that is even more useful for scheduling — it is periodic. A training job repeats the same iteration thousands of times, and each iteration produces almost the same pattern of network demand as the one before it. Within one iteration, the pattern consists of one or more phases in which the network is heavily used, which the authors of the paper call "Up" phases, and phases in which the network is almost idle, are called "Down" phases. The exact shape of these phases depends on the parallelization strategy. Figure 7.42 shows four examples of traffic patterns measured on a testbed with A100 GPU servers and 50 Gbps network cards. In data parallelism, each iteration starts with a forward pass that uses almost no network bandwidth, followed by a single long Up phase

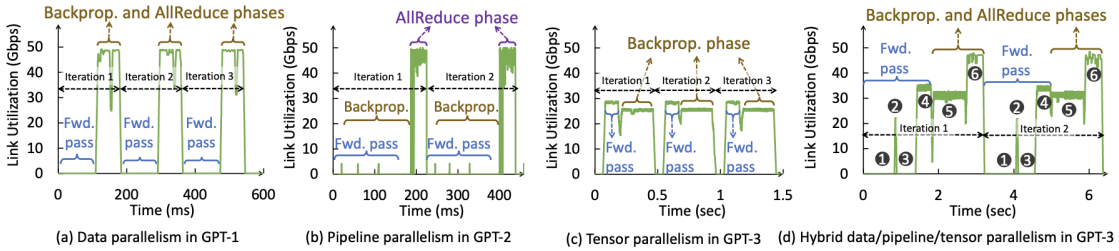


Figure 7.42: Traffic pattern of different parallelization strategies; Image credit: [Rajasekaran et al., 2024]

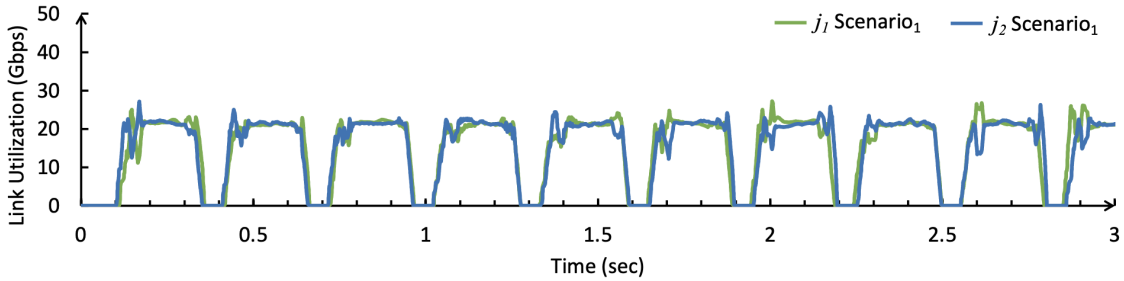
during which backpropagation and the AllReduce of gradients take place. In pipeline parallelism, the forward pass contains a few small peaks, one for each micro-batch of activations sent between stages, and these are followed by a heavy AllReduce phase between the embedding layers. In tensor parallelism, both the forward and the backward pass send a steady stream of traffic, with only a short quiet period when new data is loaded. Finally, in a hybrid setup that combines data, pipeline, and tensor parallelism, one iteration contains six different Up and Down phases, each with its own duration and bandwidth demand. The takeaway is that every job has a repeating pattern of "Up" and "Down" phases that can be measured in advance, and this is the property that CASSINI exploits.

What Happens When Two Jobs Collide?

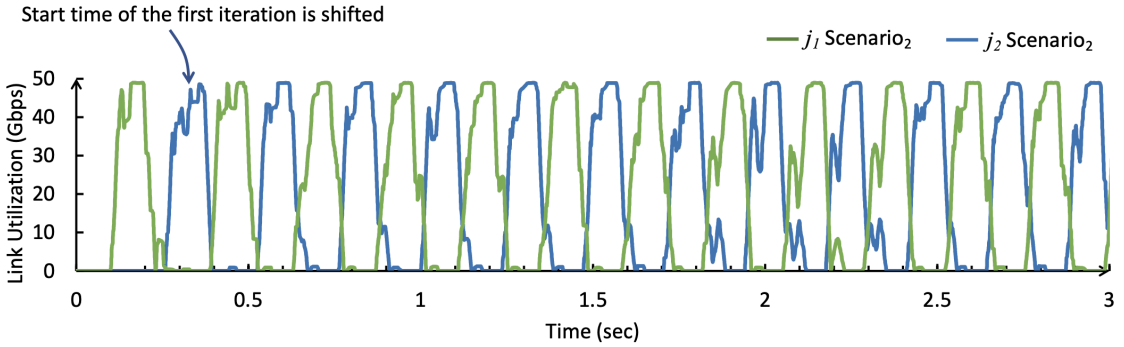
Now imagine that two such jobs share one network link, and that they happen to start at the same time. Both jobs reach their Up phase together, so both want the whole link at the same moment. The congestion control algorithm does what it is designed to do and splits the link evenly, so each job receives only half of the bandwidth. Both jobs then finish communicating late, and both go back to computing late. Because the two jobs have the same rhythm, the same collision repeats in every iteration. During the Down phases, on the other hand, the link sits idle, since neither job has anything to send. The link is therefore overloaded at some moments and wasted at others, even though its average load would be easy to handle. This is the same behaviour we saw earlier, now amplified by the presence of a second job.

7.5.2 Interleaving Communication Phases

The observation above suggests a simple remedy. If the "Up" phase of one job could be made to fit into the duration of the "Down" phase of the other job, then each job would see an empty link whenever it needs to communicate, and neither would slow the other down. We call this interleaving. To interleave two jobs, we do not need to change the switches or the congestion control protocol. We only need to delay the start of one job's iterations by a small amount of time, which we call a time-shift, so that the two periodic patterns line up in a favorable way.



(a) Jobs start at the same time



(b) Second job shifted

Figure 7.43: Impact of interleaving the Up and Down phases of two jobs that share a link; Image credit: [Rajasekaran et al., 2024]

Figure 7.43 shows this effect in a small experiment. When the two jobs start at the same time, each of them receives about half of the link bandwidth, because their Up phases overlap in every iteration. When the start of the second job is delayed by a few milliseconds, the Up phase of each job moves into the Down phase of the other. Now each job can use close to the full bandwidth whenever it communicates. Nothing was changed in the network or in the jobs themselves; only the timing was adjusted.

Note that perfect interleaving is not always possible. Whether it can be achieved depends on the jobs that share the link. We say that a group of jobs is compatible if some time-shift allows them to share a link without overloading it. Since compatibility is not an all-or-nothing property, CASSINI measures it with a number called the compatibility score. A higher score means that the jobs can be interleaved more completely, and the scheduler can use this score to compare different ways of placing jobs on the cluster.

7.5.3 When Can Jobs Be Interleaved?

To use interleaving in a scheduler, we need a simple way to describe the repeating traffic of a job, and a number that says how well a group of jobs can be interleaved. CASSINI describes each job as a circle and searches over rotations of these circles. The network demand of a job, measured while it runs alone, repeats once per iteration. CASSINI draws a circle whose perimeter is the iteration time and wraps the time axis around it, so the Up and Down phases of all iterations land on the same arcs. The circle shows where the job needs the network and how much bandwidth it needs there (Figure 7.44).

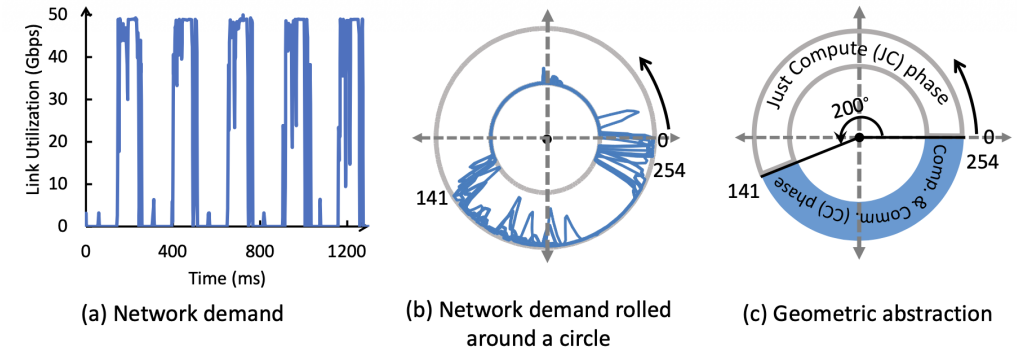


Figure 7.44: CASSINI's geometric abstraction; Image credit: [Rajasekaran et al., 2024]

For two jobs sharing a link, we put their circles on top of each other and add their demands at every angle. If the sum is above the link capacity at some angle, the jobs collide there. Rotating one circle moves its Up phase to another angle, which is the same as delaying that job by a time-shift. If some rotation keeps the sum below the capacity everywhere, the jobs are compatible. This turns a question about timing into a question about angles, which are easy to search. Jobs with different iteration times share one bigger circle whose perimeter is the least common multiple of their iteration times. For example, jobs with 40 ms and 60 ms iterations share a 120 ms circle, where the first job appears three times and the second twice.

Jobs are rarely fully compatible, so CASSINI measures how well they fit. For each angle, it takes the demand that the link cannot carry and averages this excess over all angles, relative to the link capacity. The compatibility score is one minus this average. A score of one means the jobs are fully compatible, and a low score means a poor combination. CASSINI searches for the rotations that give the highest score, using coarse angle steps of 5 degrees, which is fast and accurate enough, and converts the best rotation of each job into a time-shift.

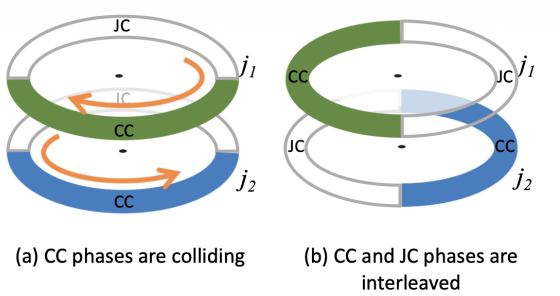


Figure 7.45: Rotating the circles interleaves the network demand of two jobs; Image credit: [Rajasekaran et al., 2024]

One Time-Shift per Job Across Many Links

The score is computed for one link, but a job usually crosses several links. If job j_2 shares one link with j_1 and another with j_3, solving each link alone gives j_2 two different time-shifts. A job cannot run with two delays, because its workers depend on each other in every iteration. Picking one value at random breaks the interleaving on the other link, and solving one global problem for all jobs does not scale. To solve this problem, CASSINI builds a graph with two kinds of vertices, jobs and links. A job is connected to a link if it uses it, and the edge stores the time-shift found for that job on that link. Jobs running alone and links with a single job need no coordination and are left out. CASSINI then walks through this graph, starting from any job with a time-shift of zero, since only the delays between jobs matter. When it moves from one job to another job over a shared link, it adds the relative shift that the optimizer chose for that link to the shift of the first job. Each job is visited once, so each job gets exactly one time-shift, and for any two jobs sharing a link, the difference of their shifts is the one that was found best for that link. This is guaranteed only if the graph has no loops, because with a loop two paths could give one job two different values. CASSINI therefore discards placements whose graph have a loop, which costs little since many candidate placements are loop-free.

7.6 Summary

In this chapter, we began by understanding why the networks built for traditional cloud computing are not good enough for machine learning training. Every parallelism strategy from the previous chapter eventually depends on GPUs exchanging large amounts of data, so the network can easily become the bottleneck. We saw that ML training traffic differs from cloud traffic in four important ways. First, it is bursty — GPUs stay silent while they compute and then all try to send data at full speed at the same time. Second, it has low entropy as there are only a few very large flows. Third, the traffic is driven by the parallelism strategy, with most of the bytes generated by tensor parallelism among small groups of nearby GPUs. Fourth, training is very sensitive to failure, since a single broken GPU, NIC, or link can stall the whole job and force a rollback to the last checkpoint.

Next, we reviewed how traditional data center networks are built. These use Clos (fat-tree) topologies, with servers connected to Top-of-Rack switches, which connect to aggregation and core layers. We then studied Remote Direct Memory Access (RDMA), the transport on which almost all ML networks are built. By letting the NIC read and write remote memory directly, bypassing the CPU and the kernel, RDMA gives very low latency and very high bandwidth. We also saw how GPUDirect RDMA lets the NIC access GPU memory directly, removing the extra copy through host memory. We then zoomed into a single server and looked at how GPUs are connected inside and across nodes. GPUs in one node are linked by fast interconnects like NVLink and form a high-bandwidth domain. We studied how NCCL picks a path for each transfer. We also studied Alibaba's HPN design, which was built to scale to tens of thousands of GPUs while removing single points of failure. It uses a non-stacked dual-ToR design, in which each NIC connects to two independent ToR switches that do not need a synchronization link.

We then moved from the network to the software that uses it, namely collective communication libraries (CCLs) such as NCCL. We reviewed the standard collective primitives, which are Broadcast, Scatter, Gather, AllGather, Reduce, AllReduce, ReduceScatter, and AllToAll, and saw how the NCCL API exposes them. Behind this simple interface, a CCL must decide which GPUs talk to which, how much data to send in each step, and how to overlap communication with computation. Most libraries answer the first question by arranging GPUs in a ring or a tree. In the ring algorithm, AllReduce is done as a ReduceScatter followed by an AllGather. In the tree algorithm, AllReduce is done as a Reduce followed by a Broadcast. NCCL chooses between rings, trees, and their combinations based on message size and the number of ranks. We also saw that on direct-connect networks such as tori, where every chip has several ports, algorithms like Trivance use both directions of every link and triple the communication distance at each step.

Next, we studied the transport layer, which decides how each transfer actually crosses the network. We first saw how a collective library such as NCCL turns a collective into bytes on the wire. Data moves through transport buffers with a two-stage copy, and NCCL chooses among transports and among the Simple, LL, and LL128 protocols, which trade latency against bandwidth. A large message is split across channels that run on different SMs, and each channel is pipelined in small slots, so

that a GPU forwards a slot as soon as it receives it. We then saw how a transport can exploit the tolerance of ML. Training tolerates small losses and reordering, and some gradients matter more than others, so lost gradients can be replaced by zeros and packets can be given priorities. We also studied multipath transport, which sprays the packets of one connection over many paths using many queue pairs, and congestion control that is either sender-driven or receiver-driven. Such ideas can be built today without new hardware. A software transport such as UCCL-Tran keeps the data path in the NIC and moves only the control path to the CPU, using RDMA write with immediate on UC or RC queue pairs, or scatter-gather lists on UD, to separate headers from payloads. Finally, we looked at clusters of more than a hundred thousand GPUs. There, latency depends on distance, so the data in flight must match the bandwidth-delay product of each path, and the library groups connections by distance to set this. Keeping messages in order across many queue pairs and starting jobs quickly through eager communicator creation also become important at this scale.

Finally, we considered a cluster shared by many jobs and studied network-aware job scheduling through CASSINI. Schedulers place the workers of a job close together, but fragmentation still forces different jobs to share links, and fair sharing by congestion control is a poor choice for jobs whose traffic comes in periodic "Up" and "Down" phases, since their bursts collide in every iteration. CASSINI instead shifts the start of one job by a small time-shift, so that its Up phase falls into the Down phase of the other job. Because a job may cross several shared links but can have only one time-shift, CASSINI uses a graph of jobs and links to assign a single consistent shift to each job.

In this chapter, we have seen that the network is as important as the GPUs for training large models, and that it must be designed, programmed, and scheduled with the special nature of ML traffic in mind. In the next chapter, we turn to LLM inference, and study how trained models are served efficiently to users.

References

Greenberg, Albert, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta (2009). "VL2: a scalable and flexible data center network". In: *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. URL: https://doi.org/10.1145/1592568.1592576.

Hu, Zhiyi, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler (2026). *Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms*. URL: https://arxiv.org/abs/2507.04786.

Juerss, Anton, Vamsi Addanki, and Stefan Schmid (2026). "Trivance: Latency-Optimal AllReduce by Shortcutting Multiport Networks". In: *Proceedings of the ACM SIGCOMM 2026 Conference*. URL: https://doi.org/10.1145/3789240.3829181.

NVIDIA Corporation (2023). *NVIDIA GPUDirect: Enhancing Data Movement and Access for GPUs*. NVIDIA Developer. URL: https://developer.nvidia.com/gpudirect.

Qian, Kun, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai (2024). "Alibaba HPN: A Data Center Network for Large Language Model Training". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. URL: https://doi.org/10.1145/3651890.3672265.

Rajasekaran, Sudarsanan, Manya Ghobadi, and Aditya Akella (2024). "CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. URL: https://www.usenix.org/conference/nsdi24/presentation/rajasekaran.

Wang, Hao, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen (2024). "Towards Domain-Specific Network Transport for Distributed DNN Training". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. URL: https://www.usenix.org/conference/nsdi24/presentation/wang-hao.

Wang, Weiyang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani (2024). "Rail-only: A Low-Cost High-Performance Network for Training LLMs with Trillion Parameters". In: *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*.

Woolley, Cliff (2015). *NCCL: Accelerated Multi-GPU Collective Communications*. Conference presentation. URL: https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf.

Yujing (2024). *An In-Depth Understanding of RDMA Interaction Mechanism between Software and Hardware*. Alibaba Cloud Community Blog. URL: https://www.alibabacloud.com/blog/601598.

Zeng, Hongyi, Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adithya Gangidi, Prashanth Kannan, Bingzhe Liu, Saif Hasan, Dong He, Deep Shah, Ashmitha Jeevaraj Shetty, Gregory R Steinbrecher, Srikanth Sundaresan, Yulun Wang, Yexin Wu, Mingran Yang, Kenny Yu, Minlan

Yu, Cen Zhao, Shengbao Zheng, Wesley Bland, Denis Boyda, Suman Gumudavelli, Subodh Iyengar, Daniel Johnson, Cristian Lumezanu, Kai Luo, Rui Miao, Zhe Qu, Venkatraghavan Ramesh, Jingliang Ren, Maxim Samoylov, Jan Seidel, Qiye Tan, Feng Tian, Xinfeng Xie, Jingyi Yang, Yimeng Zhao, Shuqiang Zhang, and Tiane Art Zhu (2026). "Connecting 100K+ GPUs: Building the Communication Stack for Large-Scale LLM Training". In: *Proceedings of the ACM SIGCOMM 2026 Conference*. URL: https://doi.org/10.1145/3789240.3829152.

Zhou, Yang, Zhongjie Chen, Ziming Mao, ChonLam Lao, Shuo Yang, Pravein Govindan Kannan, Xizhi Zhang, Jiaqi Gao, Yilong Zhao, Yongji Wu, Kaichao You, Fengyuan Ren, Zhiying Xu, Costin Raiciu, and Ion Stoica (2026). "UCCL-Tran: An Extensible Software Transport Layer for GPU Networking". In: *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. URL: https://www.usenix.org/conference/osdi26/presentation/zhou-yang.