PART VI · INFRASTRUCTURE AT SCALE · CHAPTER 14

Networking & Collective Communication

NVLink, InfiniBand, RoCE, and NCCL — The Arteries of Distributed AI

📖 **~55 pages** 🎯 **Level:** All (Foundational → Principal) 🔗 **Prereqs:** Ch 3 (GPU arch), Ch 10 (training), Ch 13 (MoE)

⏱ **Est. read:** 90 min

> *"A 10,000-GPU cluster is not 10,000 individual computers. It is a single parallel computer whose performance is determined not just by the GPUs, but by the bandwidth and latency of the wires between them."*
>
> — Distributed Systems Architecture Principle

When 1,024 GPUs train a 70B model together, every single training step requires terabytes of gradient data to move between GPUs at microsecond latency. The network is not secondary infrastructure — it is a **first-class performance component**. A well-configured InfiniBand cluster achieves 85% scaling efficiency at 1,024 GPUs; a poorly configured one might achieve 40%. That 45-point gap translates directly to $3M per year in wasted compute on a 1,000-GPU cluster at $3/GPU-hr.

This chapter builds the complete picture from silicon interconnect to collective algorithm. By the end, you will be able to design the networking layer for a 10,000-GPU training cluster, tune NCCL for maximum throughput on both InfiniBand and RoCE fabrics, diagnose scaling efficiency regressions, and justify every topology and configuration decision quantitatively.

LEARNING OBJECTIVES BY LEVEL

- → **Foundational:** Understand why networking is a bottleneck in distributed training, what NVLink vs InfiniBand vs RoCE are for, and how AllReduce moves gradients between GPUs.
- → **Intermediate:** Calculate AllReduce time for any message size and topology. Configure NCCL environment variables for production clusters. Diagnose congestion and topology mismatches with profiling tools.
- → **Principal:** Design the full networking stack for multi-thousand GPU training clusters. Justify IB vs RoCE tradeoffs with TCO math. Map 4D parallelism to physical topology. Analyze and recover from scaling efficiency regressions in production.

14.1 The Complete Communication Stack

Before optimizing anything, you need a mental model of the communication hierarchy. Every byte of gradient data, every activation tensor, every KV cache transfer in a distributed AI system passes through one of five communication levels — each with wildly different bandwidth, latency, and software

API. The core insight that shapes every parallelism decision is this: **optimize at the lowest possible level.** NVLink is 18× faster than InfiniBand for the same bandwidth. This is not a minor optimization opportunity — it is the difference between Tensor Parallelism being viable or catastrophic.

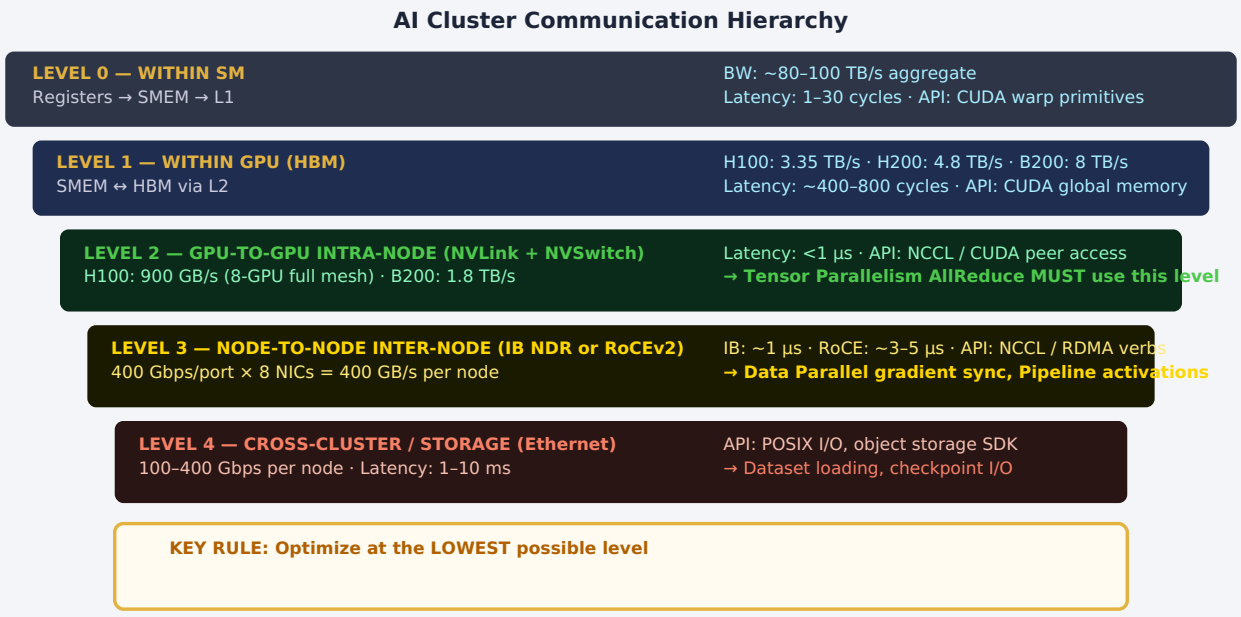


Fig 14.1 — The 5-level communication hierarchy in AI clusters. The 2-decade bandwidth gap between levels forces parallelism strategies to match communication to the fastest available interconnect.

WHY THIS HIERARCHY EXISTS

The bandwidth gap between levels is not arbitrary — it reflects physics and cost. NVLink at 900 GB/s requires wide, short copper traces on the same substrate. InfiniBand at 50 GB/s requires high-quality SerDes over DAC cables or fiber across rooms. Storage networks at 40 GB/s share infrastructure with general traffic. Each boundary is a physical constraint, and every performance optimization strategy in distributed training is ultimately about respecting these boundaries.

14.2 NVLink and NVSwitch — Intra-Node Fabric

NVLink is NVIDIA's proprietary high-speed GPU-to-GPU interconnect. It is the single most important hardware feature enabling Tensor Parallelism in production. Without it, TP would require every AllReduce to cross InfiniBand — making the TP latency budget 18× worse and rendering TP at any scale infeasible for most models.

14.2.1 NVLink Architecture SHIPPED H100 SHIPPED B200

```
── NVLink Physical Specifications ──────────────────────────────

NVLink 4.0 (H100 / H200):
  Links per GPU:          18 NVLink links
  Bandwidth per link:     50 GB/s bidirectional
  Total BW per GPU:       18 × 50 = 900 GB/s bidirectional
  Protocol:               NVIDIA proprietary (NOT PCIe)
  Connection topology:    GPU → NVSwitch ASIC (not direct GPU-GPU)
  Used for:               Tensor Parallelism, Tensor Parallel AllReduce

NVLink 5.0 (B200 / GB200):
  Bandwidth per link:     100 GB/s bidirectional
  Total BW per GPU:       18 × 100 = 1,800 GB/s = 1.8 TB/s
  Improvement:            2× over NVLink 4.0
  NVLink-C2C (GB200):     CPU-GPU unified memory at 900 GB/s

── NVLink vs PCIe Comparison ───────────────────────────────────

  PCIe Gen5 x16:          ~128 GB/s (one direction only)
  NVLink 4.0:             900 GB/s bidirectional (450 GB/s each way)
  NVLink speedup:         ~7× over PCIe for GPU-GPU transfers

  This 7× difference is why NVLink-C2C (Grace Blackwell) enables viable
  unified CPU-GPU memory: 900 GB/s CPU-GPU is only 7× slower than
  intra-GPU HBM access, making CPU DRAM a usable overflow tier.
```

14.2.2 NVSwitch — The Full-Bisection Fabric

NVSwitch is a dedicated ASIC that implements an all-to-all NVLink fabric inside a DGX/HGX node. The critical word is *full bisection bandwidth*: any GPU can communicate with any other GPU simultaneously at the maximum NVLink bandwidth. This is what makes AllReduce inside a node so fast — there is no contention.

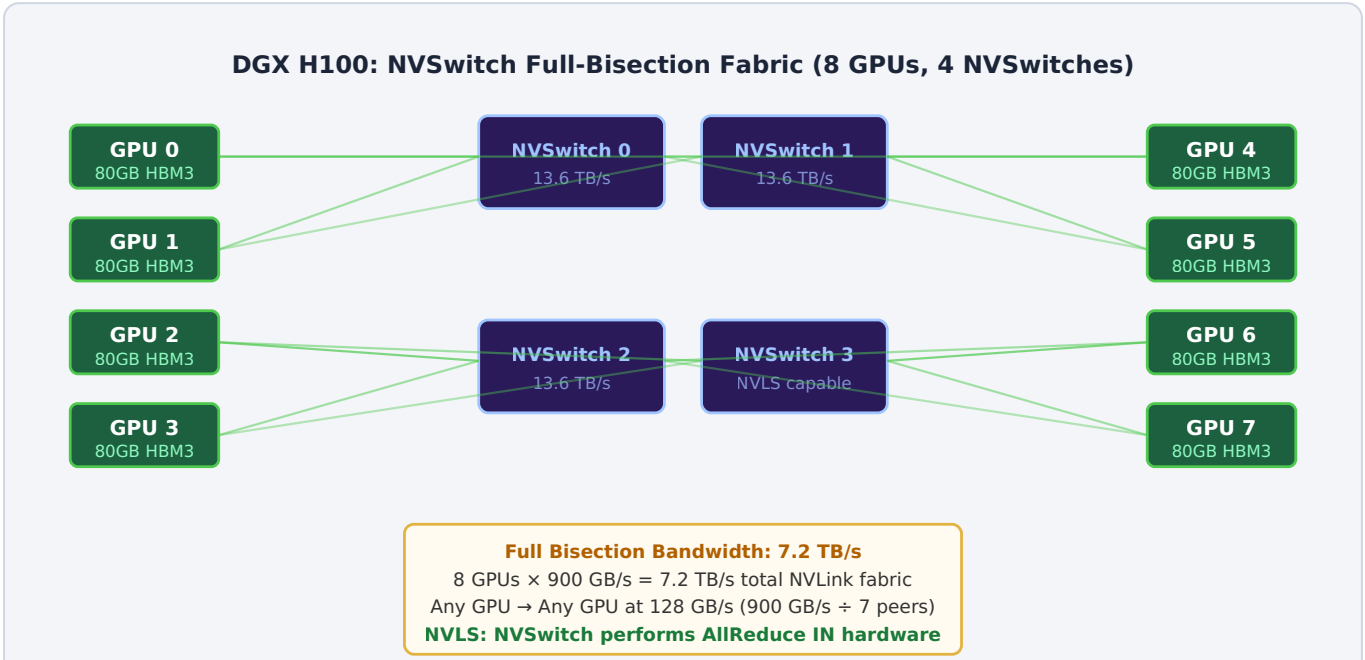


Fig 14.2 — DGX H100 NVSwitch fabric. Four NVSwitch 3.0 ASICs create full bisection bandwidth: any-to-any GPU communication at full 900 GB/s simultaneously. NVLS offloads AllReduce computation to NVSwitch hardware.

14.2.3 NVLS — NVLink SHARP (In-Network AllReduce) SHIPPED H100

NVLS offloads AllReduce computation onto NVSwitch hardware, freeing GPU SM cycles. Without NVLS, the GPU's streaming multiprocessors must execute the reduction loop on every gradient element. With NVLS, the NVSwitch sums all values in dedicated combinatorial logic — the GPU SM touches zero cycles of the AllReduce.

```
── NVLS AllReduce vs Standard Software AllReduce ──────────────────────

WITHOUT NVLS (software ring on GPU SMs):
  Each GPU executes: result = my_grad + recv_grad  (N-1 times in ring)
  SM cycles consumed: 5–15% of total step time for AllReduce
  Latency (8 GPU intra-node): ~8–12 ms for 1B parameter gradient

WITH NVLS (NVSwitch hardware reduction):
  GPUs send shards to NVSwitch simultaneously
  NVSwitch dedicated logic sums all shards → broadcasts result
  GPU SM cycles consumed: ZERO (hardware does the math)
  Latency improvement: ~40% lower vs software ring
  Effective BW improvement: ~2× (no reduce overhead)

ENABLING NVLS:
  export NCCL_NVLS_ENABLE=1          # Enable (default on H100+)
  export NCCL_ALGO=NVLS              # Force NVLS (or leave auto)

  # Verify NVLS is active:
  NCCL_DEBUG=INFO python train.py 2>&1 | grep "NVLS"
  # Expect: "NCCL INFO Channel 00 : ... NVLS"

  NCCL 2.27 (2025) enhancement: Symmetric memory
  GPUs register identical virtual addresses → AllReduce uses
  single-copy zero-copy semantics → 7–9× latency reduction
  for small collectives (<256KB) — critical for TP per-layer AllReduce
```

14.3 InfiniBand — The Inter-Node Standard

When gradients must leave a node, they enter the inter-node fabric. InfiniBand (IB) is the dominant technology for AI training clusters in 2025–2026, used in NVIDIA DGX SuperPOD, most cloud provider HPC instances, and the majority of frontier model training runs. Its key properties — low latency (~1 µs), high bandwidth (400 Gbps per port), and native RDMA (Remote Direct Memory Access) — make it purpose-built for this workload.

14.3.1 InfiniBand Stack Architecture SHIPPED NDR 400G

```
── InfiniBand End-to-End Stack ──────────────────────────────────────

PyTorch / NCCL / application
        ↓
NCCL library  (collective operations: AllReduce, AllGather, ReduceScatter)
        ↓
IB Verbs API  (libibverbs: queue pairs, completion queues, memory regions)
        ↓
RDMA kernel driver  (mlx5_core, ib_uverbs — Mellanox/NVIDIA kernel modules)
        ↓
Host Channel Adapter (HCA) — the GPU node NIC
  Mellanox ConnectX-7 (NDR 400G): most common H100 cluster NIC
  NVIDIA BlueField-3 DPU: NIC + ARM CPU + security in one package
        ↓
InfiniBand Switch Fabric
  Quantum2 (NDR 400G):   64 ports, 40 Tbps aggregate, SHARP-capable
  Quantum-X800 (XDR 800G): 2× bandwidth, 2025+ [ANNOUNCED]
        ↓
Remote HCA → Remote NCCL → Remote GPU HBM (via GPUDirect RDMA)

── NDR 400G Specifications (current standard, 2025–2026) ────────────

  Port rate:              400 Gbps per port = 50 GB/s per direction
  Typical config:         8 × ConnectX-7 NICs per DGX node
  Per-node BW:            8 × 50 GB/s = 400 GB/s bidirectional
  Switch latency:         ~100 ns per hop
  End-to-end latency:     ~1 µs (2–3 hops in fat-tree)
  RDMA:                   YES — NIC writes directly to remote GPU HBM
  SHARP:                  YES — in-switch AllReduce computation

── IB Addressing ────────────────────────────────────────────────────

  LID (Local Identifier):   16-bit address within subnet
  GID (Global Identifier):  128-bit address (subnet manager assigns)
  QP  (Queue Pair):         communication endpoint (TX QP + RX QP)
  Each NCCL channel = one IB Queue Pair
```

14.3.2 IB SHARP — In-Network AllReduce SHIPPED Quantum2

IB SHARP (Scalable Hierarchical Aggregation and Reduction Protocol) performs AllReduce inside InfiniBand switches — the inter-node analog of NVLS. Without SHARP, gradient data traverses the network twice (scatter phase + gather phase). With SHARP, switches aggregate in dedicated compute units, cutting network traffic by 50% and latency by 30–40%.

```
── IB SHARP Mechanism ──────────────────────────────────────────

WITHOUT SHARP (traditional ring AllReduce):
  Phase 1 (ReduceScatter): GPU sends gradient chunks around the ring  →  crosses EVERY link
  Phase 2 (AllGather):     Aggregated result distributed back         →  crosses EVERY link again
  Total network traffic:   2 × gradient_size × (N-1)/N ≈ 2G per step

WITH SHARP (in-switch aggregation):
  GPUs send to nearest aggregation switch
  Switch compute units sum partial results
  Switch broadcasts final result downward
  Network traffic reduction: ~50%
  Latency reduction: ~30–40% (fewer hops)
  GPU SM cycles freed: SMs compute while SHARP runs

SHARP CONFIGURATION:
  export NCCL_SHARP_ENABLE=1
  export SHARP_COLL_LOCK_ON_COMM_INIT=1    # Pre-allocate resources at init
  export NCCL_SHARP_GROUP_SIZE_THRESH=16   # Use SHARP only for groups ≥ 16 ranks
  export NCCL_ALGO=SHARP                   # Or leave auto — NCCL selects for large msgs

SHARP LIMITATIONS:
  Only InfiniBand — not available on RoCEv2 Ethernet
  Requires SHARP daemon (sharpd) on every node
  Best for large-message AllReduce (>1MB)
  Small messages (<64KB) → use Tree algorithm instead
```

14.4 RoCEv2 — Ethernet-Based RDMA

Meta, Microsoft, Google, and other hyperscalers have built some of the world's largest AI training clusters on RoCEv2 (RDMA over Converged Ethernet) rather than InfiniBand. The technical reasons — and the specific engineering challenges Meta had to solve to make RoCE work at 24,576-GPU scale — are a masterclass in infrastructure tradeoffs.

14.4.1 IB vs RoCE Decision Framework

Dimension	InfiniBand NDR 400G	RoCEv2 (400G Ethernet)	Winner
End-to-end latency	**~1 µs**	**~3–5 µs**	IB
Port bandwidth	400 Gbps (50 GB/s)	400 Gbps (50 GB/s)	Tie
Switch unit cost	**HIGH (Quantum2 ~$100K+)**	**LOW (Ethernet ~$20K)**	RoCE
NIC cost	**HIGH**	**MEDIUM**	RoCE
Vendor ecosystem	**NVIDIA/Mellanox dominated**	**Multi-vendor (Broadcom, Intel, Marvell)**	RoCE
Congestion control	**Built-in hardware CC**	**Requires explicit PFC + DCQCN config**	IB
SHARP (in-network compute)	**YES (Quantum2+)**	**NO**	IB
Multi-tenant cloud	**Limited isolation**	**Excellent (standard L2/L3 VLAN)**	RoCE
Operational tooling	**IB-specific (ibstat, opensm)**	**Standard (netstat, tcpdump, etc.)**	RoCE
TCO at 1,000+ nodes	Lower if SHARP savings dominate	Lower if switch cost dominates	Context-dependent

META'S CHOICE AT 24,576-GPU SCALE (LLAMA 3/4)

Meta selected RoCEv2 for their Grand Teton / RSC clusters. Their reasoning: (1) Switch cost savings at 24K+ GPUs offset the 4× latency penalty, (2) Standard Ethernet integrates with existing datacenter operations tooling, (3) Supply chain diversification across multiple NIC/switch vendors, (4) Network fabric shared with inference and general datacenter traffic. The challenge: Meta had to *invent* a new congestion control protocol (CTS) to make RoCE reliable at this scale — a 2-year engineering investment.

14.4.2 Making Ethernet Lossless: PFC and DCQCN

RDMA requires lossless transport — a single dropped packet causes retransmission that can stall an entire AllReduce ring, multiplying latency by 10×. Standard Ethernet drops packets whenever switch buffers overflow. Two complementary mechanisms prevent this:

```
── MECHANISM 1: Priority Flow Control (PFC) ──────────────────────────

When switch output buffer reaches threshold:
  Switch sends PAUSE frame to upstream sender
  Sender halts transmission for PAUSE duration
  Result: No packet drops — lossless behavior

Problem: Head-of-Line Blocking (HOL)
  PAUSE blocks ALL flows in the same priority class
  Unrelated traffic (storage, management) pauses alongside RDMA
  Cascading PFC storms can dark out entire racks

Mitigation:
  Isolate RDMA on a dedicated priority queue (priority 3)
  mlnx_qos -i eth0 --pfc 0,0,0,1,0,0,0,0   # PFC on priority 3 only

── MECHANISM 2: DCQCN (Data Center Quantized Congestion Notification) ──

DCQCN prevents PFC storms by rate-limiting BEFORE buffers fill:

  Step 1: Switch detects congestion → marks packets with ECN
  Step 2: Receiver sends CNP (Congestion Notification Packet) to sender
  Step 3: Sender reduces transmission rate (CNPQ algorithm)
  Step 4: Rate recovers gradually when congestion clears
  Result: Buffer utilization stays low → PFC rarely triggers

Configuration (Mellanox ConnectX NIC):
  mlnx_qos -i eth0 --dscp2prio "set,106,3"    # DSCP 106 → priority 3
  cma_roce_tos=106   # in /etc/rdma/rdma.conf
```

14.4.3 Meta's CTS Protocol — Receiver-Driven Admission SHIPPED Meta RSC

Meta discovered that DCQCN reacts *after* congestion develops — at 400G speeds, switch buffers can fill in microseconds before DCQCN responds. Their solution, published at SIGCOMM 2024, is a receiver-driven admission protocol called CTS (Clear to Send) built into their NCCLX fork:

```
── Meta CTS Protocol (NCCLX fork) ──────────────────────────────

PROBLEM with standard DCQCN at 400G:
  ECN marking happens AFTER switch buffers start filling
  At 400G: buffers fill in ~1 µs before DCQCN reacts
  Brief but frequent drops → NCCL retransmissions → 3–5× slower

CTS SOLUTION — receiver-driven admission control:

  Phase 1 (SENDER requests):
    "I have N bytes to send to GPU_X"
    Sender WAITS — does NOT start transmitting immediately

  Phase 2 (RECEIVER grants):
    GPU_X checks available receive buffer space
    If space: sends 5-byte CTS packet back to sender

  Phase 3 (SENDER transmits):
    Only after CTS received does RDMA write begin
    Switch buffers NEVER overflow → PFC never triggers

  Result: Switch buffer utilization 10× lower than standard RoCE
  Zero packet drops under normal operation
  ~90% scaling efficiency at 24,576 GPUs (vs ~60% without CTS)

Implementation: NCCLX = Meta's NCCL fork with CTS
Used for: All Llama 3 and Llama 4 training runs
Published: SIGCOMM 2024 — "Reconfiguring Meta's production AI cluster"
```

14.5 Network Topology — Fat-Tree and Rail-Optimized Fabrics

Bandwidth numbers tell only half the story. The *topology* — how switches and cables are physically connected — determines whether that bandwidth is actually available to training workloads. The wrong topology can cut effective AllReduce throughput by 30–50% even on a 400G fabric.

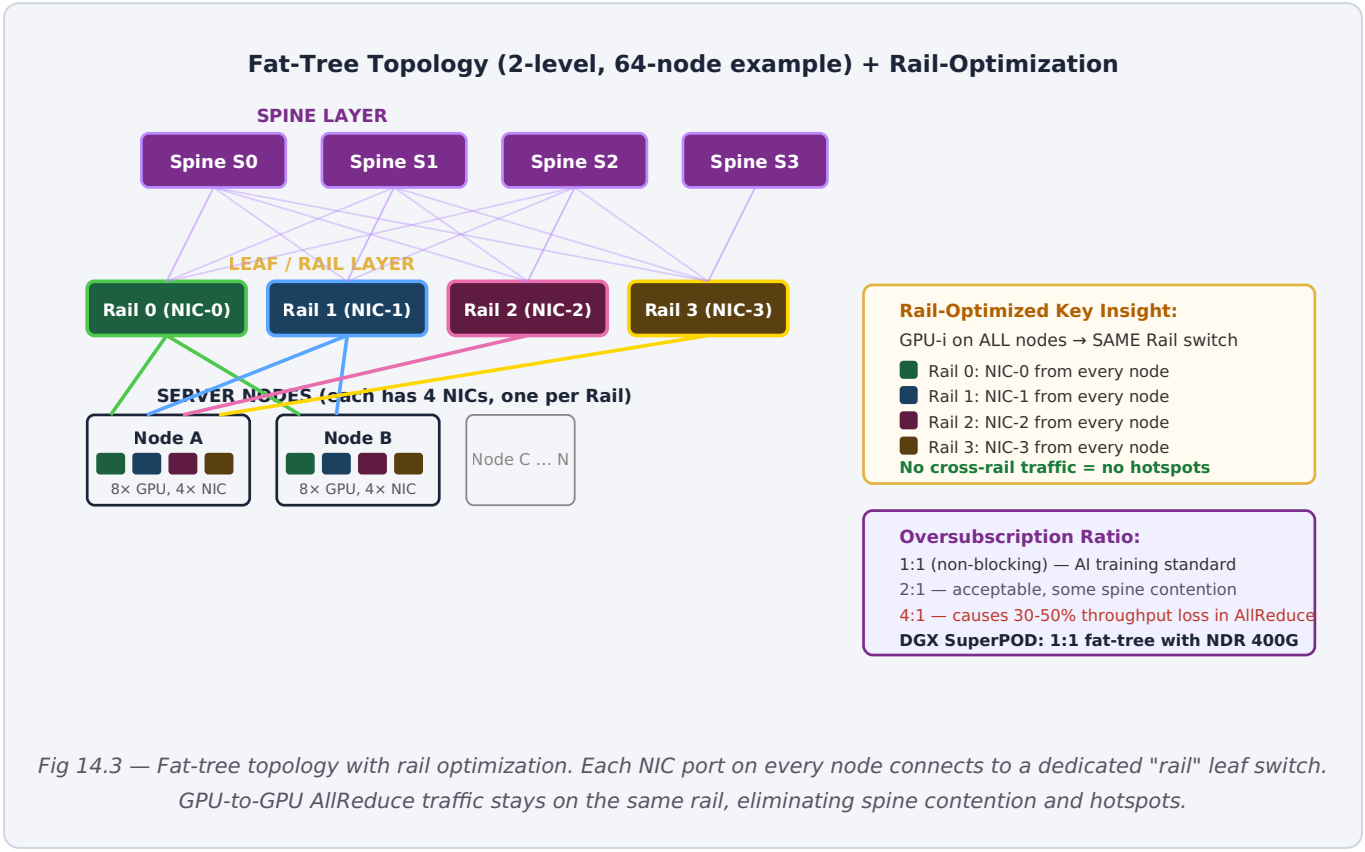


Fig 14.3 — Fat-tree topology with rail optimization. Each NIC port on every node connects to a dedicated "rail" leaf switch. GPU-to-GPU AllReduce traffic stays on the same rail, eliminating spine contention and hotspots.

```
── Rail-Optimized Topology: Why It Matters ──────────────────────────

Without rail optimization (random NIC-to-switch assignment):
  GPU-0 on Node A sends via Rail0 → Spine → Rail2 → GPU-0 on Node B
  GPU-1's traffic also crosses same spine links → hotspot at spine
  Queue buildup → ECN → retransmissions → 30–50% throughput loss

With rail optimization (GPU-i → Rail-i always):
  GPU-0 (all nodes) → Rail0 ONLY → no cross-rail traffic
  Each rail carries exactly 1/N_rails of total AllReduce traffic
  Spine switches: balanced load, no hotspots
  Effective AllReduce bandwidth: full 400 GB/s utilized

NCCL configuration for rail-optimized topology:
  export NCCL_IB_HCA=mlx5_0,mlx5_1,mlx5_2,mlx5_3  # GPU 0→mlx5_0, GPU 1→mlx5_1, etc.
  # CRITICAL: ordering must match GPU-to-NIC PCIe topology
  # Verify: NCCL_TOPO_DUMP_FILE=/tmp/topo.xml python -c "import torch.distributed"
```

14.6 NCCL — Complete Configuration Guide

NCCL (NVIDIA Collective Communication Library) is the software layer that abstracts all hardware differences — NVLink, InfiniBand, RoCE — into a unified collective API. Understanding NCCL is non-negotiable for distributed training performance. An unconfigured NCCL on a well-built InfiniBand cluster can achieve only 40–50% of available bandwidth. A properly configured NCCL consistently achieves 85–95%.

14.6.1 Collective Primitives Reference

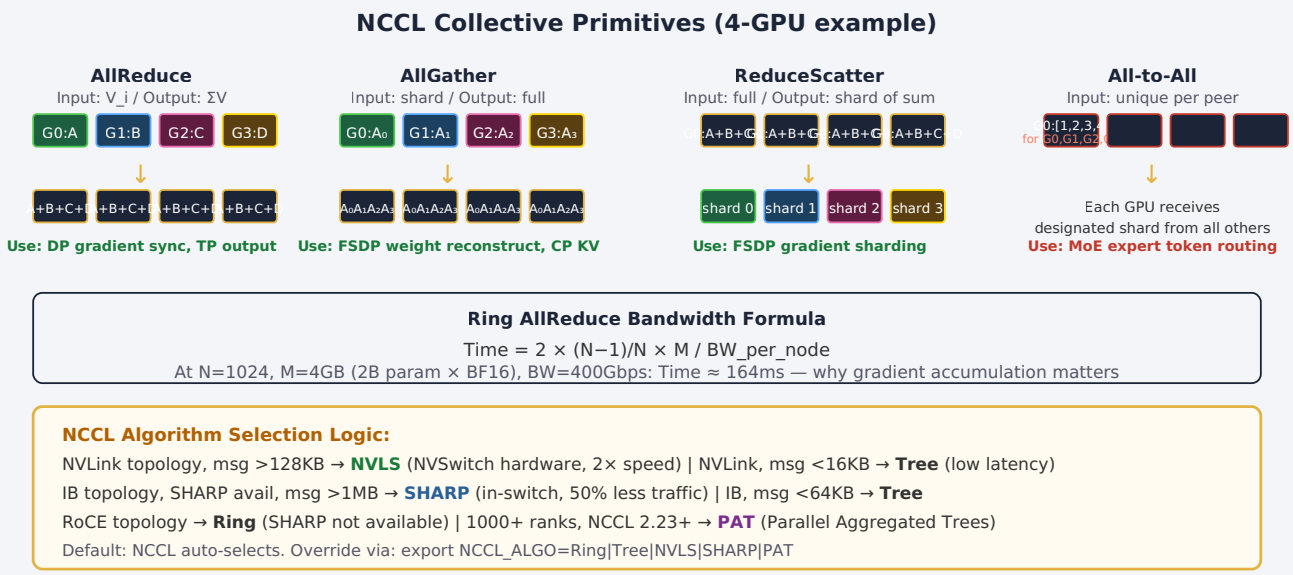


Fig 14.4 — NCCL collective primitives with their use cases and algorithm selection logic. AllReduce bandwidth = (N−1)/N × BW — at 1024 GPUs this is ≈99% of peak, so the bottleneck is bandwidth not the formula.

14.6.2 Complete NCCL Production Environment Variables

```
── NCCL Production Configuration Reference ──────────────────────────

# ── Debug and Diagnostics ──────────────────────────────────────────
export NCCL_DEBUG=WARN                  # WARN | INFO | TRACE
export NCCL_DEBUG_FILE=/tmp/nccl_%h_%p.log  # Per-host per-process log
# NCCL_DEBUG=INFO reveals: algorithm selection, topology detection, ring order

# ── Transport Selection ────────────────────────────────────────────
export NCCL_IB_DISABLE=0                # 0=enable IB/RoCE, 1=force TCP socket
export NCCL_IB_HCA=mlx5_0,mlx5_2,mlx5_4,mlx5_6  # Which HCAs (match to GPUs!)
export NCCL_SOCKET_IFNAME=eth0          # Fallback interface if IB unavailable

# ── InfiniBand / RoCE Performance ──────────────────────────────────
export NCCL_IB_CUDA_SUPPORT=1           # Enable GPUDirect RDMA — CRITICAL
export NCCL_NET_GDR_LEVEL=2             # 0=off, 1=same PCIe root, 2=system-wide
export NCCL_IB_GID_INDEX=3              # RoCE GID (check: ibv_devinfo -d mlx5_0)
export NCCL_IB_TC=106                   # DSCP for ECN marking on RoCE
export NCCL_IB_QPS_PER_CONNECTION=4    # QPs per NIC (1=default, 4=recommended 400G)

# ── Algorithm and Protocol ─────────────────────────────────────────
export NCCL_ALGO=Ring                   # Ring|Tree|CollNet|NVLS|PAT|auto
export NCCL_PROTO=Simple                # Simple|LL|LL128 (Simple=IB, LL128=NVLink small)

# ── Channels and Buffer ────────────────────────────────────────────
export NCCL_MIN_NCHANNELS=4
export NCCL_MAX_NCHANNELS=8             # More channels = more BW used
export NCCL_BUFFSIZE=4194304            # 4MB buffer/channel (larger=better for big msgs)

# ── SHARP (IB in-network) + NVLS (NVSwitch) ────────────────────────
export NCCL_SHARP_ENABLE=1              # IB SHARP for inter-node
export SHARP_COLL_LOCK_ON_COMM_INIT=1
export NCCL_NVLS_ENABLE=1              # NVLink SHARP for intra-node

# ── Timeout and Stability ──────────────────────────────────────────
export NCCL_TIMEOUT=300                 # Watchdog timeout (seconds)
# Increase to 1800 for large-cluster checkpoint operations

NCCL 2.27 (2025) — Key new defaults:
  Symmetric memory: auto-enabled — 7-9× latency reduction for small collectives
  Direct NIC (Grace Blackwell): proxy threads eliminated → 10% step time reduction
```

14.6.3 GPUDirect RDMA — The Zero-Copy Path

GPUDirect RDMA allows the InfiniBand NIC to read from and write to GPU HBM directly, bypassing the CPU and host DRAM entirely. Without it, every NCCL send operation requires a GPU→CPU→NIC data hop, consuming 16 CPU cores and adding 2–3 µs latency. With it, data moves GPU HBM → NIC DMA queue → network in a single transaction.

```
── Data Path Comparison: With vs Without GPUDirect RDMA ──────────────────

WITHOUT GPUDirect (legacy):
  GPU HBM → [PCIe] → CPU copy buffer → [PCIe] → NIC DMA → network
  Extra latency:  +2–3 µs per collective message
  CPU overhead:   16 cores per DGX node (1 proxy thread per NIC × 8 + buffers)
  BW bottleneck:  Limited by PCIe Gen5 = 128 GB/s (NIC bandwidth underutilized)

WITH GPUDirect RDMA:
  GPU HBM → [PCIe P2P BAR mapping] → NIC DMA → network
  Latency:        ~0.5 µs (direct memory-mapped DMA)
  CPU overhead:   2–4 cores (metadata only, no data movement)
  Freed CPU cores: 12 per DGX node
  Performance:    10–20% improvement on communication-bound steps

VERIFICATION:
  nvidia-smi topo -p2p r          # Should show OK/PIX/NV#, not HOST
  NCCL_DEBUG=INFO python train.py 2>&1 | grep "GDR"
  # Expected: "NCCL INFO GDR enabled"

  # Benchmark NIC-to-GPU bandwidth:
  ib_write_bw -d mlx5_0 -x 3 --report_gbits      # server
  ib_write_bw -d mlx5_0 -x 3 <server> --report_gbits  # client
  # Target: ~380–395 Gbps (95%+ of 400G line rate)
```

14.7 Collective Communication Optimization

14.7.1 Compute-Communication Overlap

The highest-leverage networking optimization is not choosing the right algorithm — it is ensuring that communication never *blocks* computation. A training step where AllReduce runs sequentially after the backward pass wastes 100% of the communication time as dead GPU cycles. The goal is to make AllReduce invisible by overlapping it with the next computation.

```
── Overlap Strategy: Sequential vs Pipelined ──────────────────────────

WITHOUT OVERLAP (naive DDP):
  ┌──────────────────────────────────────────────────────────┐
  │  Forward (compute)  │  Backward (compute)  │  AllReduce  │
  └──────────────────────────────────────────────────────────┘
  Total: T_fwd + T_bwd + T_allreduce
  AllReduce fraction: typically 20–40% of step time = GPU dead time

WITH OVERLAP (DDP gradient bucketing):
  ┌──────────────────────────────────────────────────────────┐
  │  Forward Pass  │  Backward Pass (layer N)  │             │
  │                │  + AllReduce (bucket 0)    │             │
  │                │       + AllReduce (bucket 1)│             │
  └──────────────────────────────────────────────────────────┘
  Total: T_fwd + max(T_bwd, T_allreduce)
  Speedup: up to 1.3× for communication-heavy models

DDP CONFIGURATION for maximum overlap:
  model = torch.nn.parallel.DistributedDataParallel(
      model,
      bucket_cap_mb=25,           # Gradient bucket size MB (tune per model)
      gradient_as_bucket_view=True,  # Zero-copy gradient bucketing
      static_graph=True,          # Optimize for fixed computation graph
  )

FSDP CONFIGURATION for forward+backward overlap:
  model = FSDP(
      model,
      forward_prefetch=True,      # AllGather next layer while computing current
      backward_prefetch=BackwardPrefetch.BACKWARD_PRE,  # Prefetch during backward
      limit_all_gathers=True,     # Prevent OOM from too many inflight AllGathers
  )
```

14.7.2 Gradient Accumulation — Amortizing Communication

```
── Gradient Accumulation: Reducing AllReduce Frequency ──────────────────

WITHOUT accumulation (accum_steps=1, frequent communication):
  1000 training steps → 1000 AllReduce operations
  AllReduce fraction of time: ~25–40% (too high)

WITH accumulation (accum_steps=8, batched communication):
  125 "super-steps" → 125 AllReduce operations (8× fewer)
  AllReduce fraction of time: ~4–6%  (excellent)

Global batch size unchanged:
  micro_batch_per_gpu=4, accum_steps=8, world_size=512
  Global batch = 4 × 8 × 512 = 16,384 tokens per update

PyTorch FSDP implementation:
  for step, batch in enumerate(loader):
      ctx = model.no_sync() if (step % accum_steps != 0) else nullcontext()
      with ctx:
          loss = model(batch) / accum_steps  # Normalize gradient
          loss.backward()              # Grad accumulated, no AllReduce

      if (step + 1) % accum_steps == 0:
          optimizer.step()             # AllReduce fires HERE (once per 8 steps)
          optimizer.zero_grad()
```

14.8 Parallelism-to-Network Mapping

This is the section where networking and ML training architecture converge. Every parallelism strategy generates a specific communication pattern. The physical fabric it uses — NVLink or InfiniBand — determines whether that communication is an imperceptible overhead or a catastrophic bottleneck.

4D Parallelism → Network Fabric Mapping

Parallelism	Collective Op	Network Required	Frequency	Why
TP (Tensor Parallel) degree 1–8	AllReduce	**NVLink ONLY** MUST stay intra-node	2× per transformer layer	Over IB: 160 ops/fwd = 80ms latency (+2000%)
PP (Pipeline Parallel) stage-per-node	P2P Send/Recv activations	InfiniBand / RoCE	Once per micro-batch per stage	Low frequency, ~1ms tolerance
DP (Data Parallel) FSDP / ZeRO	AllReduce or AllGather+ReduceScatter	InfiniBand / RoCE	Once per step (or gradient bucket)	Large messages, BW-dominated
EP (Expert Parallel) MoE routing	All-to-All	NVLink if EP≤8 IB required if EP>8	Per MoE layer (every 2–4 layers)	Hardest to optimize at scale
CP (Context Parallel) long sequence	AllGather (K,V)	InfiniBand / RoCE	Once per attention layer	Sequence shards across nodes

4D Layout Example: TP=8, PP=4, CP=2, DP=16 = 1,024 GPUs
Innermost→fastest: TP=8 (8 GPUs, NVLink) → CP=2 (2 nodes, IB low-latency) → PP=4 (4 node-groups, IB) → DP=16 (16 replicas, IB BW)
Rule: Pack TP+CP+PP groups onto fewest IB switches possible. SLURM: --switches=N constraint enforces this.

Fig 14.5 — Mapping each parallelism dimension to its required network fabric. This mapping is the most important design decision in cluster topology planning. A single violation (e.g., TP=16 across IB) can reduce training throughput by 10× for that component.

```
── Quantitative Justification: Why TP Must Stay on NVLink ──────────────

TP=8 AllReduce overhead per transformer layer:
  Message size per AllReduce: hidden_dim × seq_len × 2 bytes = 4096 × 2048 × 2 = 16 MB
  Number of AllReduces per 80-layer forward pass: 80 layers × 2 ops = 160 ops

  Over InfiniBand (IB) at 50 GB/s per NIC × 8 = 400 GB/s:
    Time per AllReduce = 2 × 7/8 × 16MB / 400GB/s = 0.07ms
    Total TP overhead = 160 × 0.07ms = 11.2ms per forward pass
    Wait — but NCCL ring uses multiple hops, and latency dominates small messages
    Realistic: 160 × 0.5ms = 80ms per forward pass (unusable)

  Over NVLink at 900 GB/s intra-node:
    Time per AllReduce ≈ 0.05ms (NVLS hardware, symmetric memory)
    Total TP overhead = 160 × 0.05ms = 8ms per forward pass (acceptable)

  IB vs NVLink for TP: 10× difference in overhead — not a minor optimization
```

14.9 Scaling Efficiency — Measuring Network Impact on MFU

Scaling efficiency is the ground truth metric for how well your cluster networking supports your training workload. It answers: "If I double the GPUs, how close to double do I get in throughput?" Every point of lost efficiency is wasted infrastructure cost. At 1,000 GPUs × $3/hr, going from 79% to 85% scaling efficiency saves ~$1.5M per year.

```
── Scaling Efficiency Definition and Measurement ──────────────────────

Scaling Efficiency (SE) = (Multi-node tokens/s) / (Single-node tokens/s × N_nodes)

Ideal SE = 100% — N GPUs deliver exactly N× single-GPU throughput
Typical SE = 70–90% for well-configured clusters at 1K+ GPUs

── LLaMA-3 70B Training Benchmark (empirical, 128-node × 8 GPU) ─────────

  8 GPUs (1 node baseline):              4,800 tokens/s
  1,024 GPUs (128 nodes, 4D parallel):   488,000 tokens/s
  Scaling efficiency: 488,000 / (4,800 × 128) = 79.4%

  The 20.6% efficiency loss breaks down:
    AllReduce overhead (DP dimension):   ~8%   (gradient sync across 128 nodes)
    Pipeline bubbles (PP=8, 16 microbatch): ~6% (bubble = (PP-1)/(PP-1+M) = 7/23 = 30% → ~6% step time)
    Data loading stalls:                  ~3%   (storage I/O not fast enough)
    Other overhead:                       ~3%   (SLURM scheduling, kernel launch)

── How to Measure in Production ──────────────────────────────────────────

  # Step 1: Single-node baseline
  torchrun --nproc_per_node=8 train.py --steps 100 --profile
  # Record: tokens_per_second, avg_step_time_ms

  # Step 2: Multi-node run
  torchrun --nproc_per_node=8 --nnodes=128 train.py --steps 100 --profile
  # Compare: (multi-node tok/s) / (single-node tok/s × 128)

  # Step 3: Attribute the gap (via Nsight Systems or NCCL timing)
  nsys profile --trace=cuda,nvtx,nccl --output=profile python train.py
  # Look for: NCCL AllReduce rows serialized (bad) vs overlapped (good)

── Pipeline Bubble Analysis ──────────────────────────────────────────────

  Bubble fraction = (PP - 1) / (PP - 1 + num_micro_batches)

  PP=8, micro_batches=8:  bubble = 7/15 = 46.7% — terrible
  PP=8, micro_batches=16: bubble = 7/23 = 30.4% — still bad
  PP=8, micro_batches=32: bubble = 7/39 = 17.9% — acceptable
  PP=4, micro_batches=16: bubble = 3/19 = 15.8% — good

  Solution: Interleaved 1F1B scheduling with virtual pipeline stages
  Each physical PP stage contains V virtual stages → reduces bubble to 1/(V×PP)
```

Profiling & Diagnostic Cookbook

🔬 **PRODUCTION PROFILING GUIDE — NETWORKING**

Symptom: Training step time higher than expected, scaling efficiency below 75%

Step 1 — Measure AllReduce fraction of step time

```
export NCCL_DEBUG=INFO
nsys profile --trace=cuda,nvtx,nccl -o profile_run python train.py --steps 20
# In nsys-ui: look at the "NCCL" row
# Metric: NCCL_total_time / total_step_time
# <10%: networking not the bottleneck
# 10–20%: acceptable, minor tuning possible
# 20–30%: investigate topology and overlap
# >30%: networking IS the primary bottleneck
```

Step 2 — Test raw AllReduce bandwidth (baseline the fabric)

```
# NCCL bandwidth test (from nccl-tests package)
mpirun -np 1024 --hostfile hosts all_reduce_perf   -b 1M -e 4G -f 2 -g 1 -c 1
# Output: busBW (GB/s) vs message size
# Compare against theoretical peak: (N-1)/N × NIC_BW × N_NICs
# If achieved < 80% of theoretical: topology or config problem
```

Step 3 — Check IB fabric health

```
ibstat mlx5_0 | grep -E "State|Physical|Rate"
# Expect: Active / Polling / 400 Gb/sec

perfquery -x 3              # Check port error counters
# PortXmitDiscards > 0 → congestion, bad cable, or config
# PortRcvErrors > 0    → signal integrity issue (check cable)

ib_write_bw -d mlx5_0 --report_gbits [server]
# Target: ~380–395 Gbps (95%+ of 400G)
# If < 350 Gbps: GPUDirect disabled, PFC misconfigured, or cable issue
```

Step 4 — Decision tree

```
IF AllReduce > 30% of step time:
  → Check if NCCL operations are overlapped with compute (nsys)
  → If NOT overlapped: add DDP(static_graph=True, gradient_as_bucket_view=True)
  → Increase gradient_accumulation_steps to reduce AllReduce frequency
  → Check NCCL_IB_CUDA_SUPPORT=1 (GPUDirect RDMA enabled)

IF raw bandwidth test < 80% of peak:
  → Check NCCL_IB_HCA ordering (must match GPU-to-NIC PCIe topology)
  → Verify rail-optimized topology: GPU-i traffic → NIC-i → Rail-i
  → Check oversubscription ratio (spine bandwidth adequate?)
  → Enable NCCL_IB_QPS_PER_CONNECTION=4

IF ibstat shows errors or link speed != 400G:
  → Replace cable (most common cause)
  → Check SFP/QSFP transceiver compatibility
  → Verify PFC and DCQCN configuration on all switches

IF scaling efficiency degrades beyond 1024 GPUs:
  → Consider IB SHARP (export NCCL_SHARP_ENABLE=1)
  → Evaluate gradient accumulation to reduce sync frequency
  → Profile per-GPU step times — identify straggler nodes (thermal throttle?)
```

Worked Example

✏ WORKED EXAMPLE — ALLREDUCE BUDGET FOR 70B MODEL TRAINING AT 1,024 GPUS

Setup

You are planning a LLaMA-3 70B training run on 1,024 H100 GPUs (128 nodes × 8 GPUs). Parallelism: TP=8 (intra-node NVLink), PP=4, DP=32. Cluster uses IB NDR 400G with 8 × ConnectX-7 NICs per node. The training objective: minimize step time while keeping AllReduce overhead below 10% of step time. Calculate the AllReduce budget, check feasibility, and decide whether gradient accumulation is required.

Step 1: DP group size and message size

```
DP groups:    1,024 GPUs / (TP=8 × PP=4) = 32 DP replicas
Model size:   70B parameters × 2 bytes/param (BF16) = 140 GB
Gradient size per step: 140 GB (full model gradient in DP AllReduce)
# But FSDP ReduceScatter sends only 1/DP fraction per node, then AllGather
Message per GPU per ReduceScatter: 140 GB / 32 = 4.375 GB
Total AllReduce volume: 2 × (31/32) × 140 GB = 271 GB of network traffic
```

Step 2: AllReduce time calculation

```
Available IB bandwidth (one DP group = all 128 nodes):
  Per-node NIC BW: 8 NICs × 50 GB/s = 400 GB/s
  Ring algorithm: bandwidth = (N-1)/N × per_node_BW = (31/32) × 400 GB/s = 387 GB/s

AllReduce time = message_size / bandwidth = 271 GB / 387 GB/s = 0.70 seconds

With IB SHARP (50% traffic reduction):
  Effective time = 0.70 × 0.5 = 0.35 seconds
```

Step 3: Check against compute time

```
LLaMA-3 70B forward+backward compute time per step (H100, BF16):
  ~2.5–3.5 seconds at global batch = 4M tokens

AllReduce fraction WITHOUT overlap or accumulation:
  0.70 / 3.0 = 23%  TOO HIGH — exceeds 10% budget

With overlap (DDP/FSDP gradient bucketing, 60% overlap achievable):
  Effective overhead = 0.70 × (1 - 0.60) = 0.28s → 9.3%  within budget

With SHARP + overlap:
  Effective overhead = 0.35 × (1 - 0.60) = 0.14s → 4.7%  excellent
```

Step 4: Gradient accumulation decision

```
Target: AllReduce fraction < 10% without relying on overlap
AllReduce time per sync: 0.70s (without SHARP), 0.35s (with SHARP)

Compute time per accumulated super-step with accum=4:
  super_step_time = 4 × 3.0s = 12.0s
  AllReduce fraction = 0.70 / 12.0 = 5.8% — easily within budget

Recommendation:
  Minimum: NCCL_SHARP_ENABLE=1 + DDP gradient bucketing
  Conservative: gradient_accumulation_steps=4 + SHARP
  Both: AllReduce cost becomes negligible (<5% of step)
  Global batch with accum=4: 4M × 4 = 16M tokens per update
  (verify convergence at this batch size before committing)
```

Result

A 70B model at 1,024 H100s on IB NDR 400G is feasible with AllReduce overhead within budget — but requires at minimum: (1) IB SHARP enabled, (2) FSDP with backward_prefetch overlap. Without both, AllReduce exceeds 20% of step time and scaling efficiency drops below

75%. The gradient accumulation option adds a 6× reduction in sync frequency, providing a margin of safety for clusters with less optimal topology.

Key Takeaways

CORE CONCEPTS

- AI clusters have a 5-level communication hierarchy. Each level differs by 2–3 orders of magnitude in bandwidth and latency. **Optimize at the lowest possible level** — NVLink is 18× faster than IB for the same bandwidth dollar.
- NVLink (900 GB/s, H100) enables Tensor Parallelism. InfiniBand NDR (400 Gbps/port × 8) enables Data and Pipeline Parallelism. RoCEv2 provides IB-equivalent bandwidth at lower switch cost but 4× higher latency — requiring explicit lossless configuration via PFC and DCQCN.
- NCCL selects algorithm based on topology + message size: NVLS (in-NVSwitch) for intra-node large messages, IB SHARP for inter-node large messages, Tree for small messages (<64KB), Ring as default. NCCL 2.27 symmetric memory provides 7–9× latency reduction for small collectives.
- Ring AllReduce bandwidth = (N−1)/N × BW — at N=1024 GPUs this is effectively 100% of NIC bandwidth. The bottleneck is absolute bandwidth, not the algorithm. A 2B parameter gradient (4GB) takes ~164ms on a 400G IB cluster without SHARP or accumulation.

COST & PRODUCTIVITY IMPACT

- Raising scaling efficiency from 79% to 85% on a 1,024-GPU cluster at $3/GPU-hr saves ~$1.5M per year — achievable with SHARP + FSDP gradient overlap + proper rail-optimized topology.
- Gradient accumulation at 8× reduces AllReduce operations by 8× — typically moving AllReduce from 25–40% of step time to 4–6%. At 1,000 GPUs this alone can recover 20%+ MFU, equivalent to 200 free GPUs of compute.
- Enabling GPUDirect RDMA (NCCL_IB_CUDA_SUPPORT=1) frees 12 CPU cores per DGX node from NCCL proxy threads — equivalent to recovering $15K/year in CPU headroom per node on dedicated HPC instances.

PRINCIPAL-LEVEL INSIGHT

- **The topology IS the architecture:** A TP=8 training job that accidentally crosses IB instead of NVLink is not "10% slower" — it is 10× slower on that dimension and often untrainable. The physical mapping of TP groups to nodes is the most consequential hardware decision in cluster design.
- **RoCE vs IB is a TCO question, not a performance question:** At 400G per port, both deliver the same bandwidth. Meta proved RoCE can achieve 90% scaling efficiency at 24K GPUs — but required inventing CTS congestion control. The hidden cost of RoCE is engineering debt, not hardware.
- **Scaling efficiency has a floor set by pipeline bubbles, not networking:** At PP=8 with 16 micro-batches, pipeline bubbles alone consume 30% of step time. Doubling IB bandwidth won't fix this. The correct solution is increasing micro-batches (to 32+) or using interleaved 1F1B scheduling — a model architecture choice, not a network choice.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. A new engineer asks: "Why can't we just use regular Ethernet for GPU-to-GPU communication? It's cheaper." Explain the three specific failure modes that would occur — covering both latency and loss — and describe what would happen to a 128-GPU AllReduce if even 0.1% of packets were dropped.
2. Explain the difference between NVLink, InfiniBand, and RoCEv2 in one sentence each. For each, name one parallelism strategy that depends on it and explain why it requires that specific technology rather than one of the others.
3. What is NCCL's role in distributed training? Why does a ML engineer need to configure it explicitly rather than simply using PyTorch's `dist.all_reduce()` call and letting it choose defaults?

INTERMEDIATE Apply and calculate — production context

1. A 256-GPU training cluster (32 nodes × 8 GPUs, IB NDR 400G, 8 NICs per node) trains a 13B parameter model in BF16. DP=32 (one DP group = all nodes), TP=8 (intra-node), PP=1. Calculate: (a) the AllReduce message size, (b) the ring AllReduce time in seconds, (c) the AllReduce fraction of step time if total step time is 1.2 seconds, and (d) whether this is acceptable and what you would do if it is not.
2. You enable `NCCL_DEBUG=INFO` and see the log line: `"Channel 00 : Coll/chnls 4/4 ... Ring"` instead of the `NVLS` you expected on H100. List three possible reasons NCCL is falling back to Ring instead of NVLS, and give the diagnostic command to verify each.
3. A training job achieves 65% scaling efficiency at 512 GPUs. Nsight Systems shows NCCL AllReduce consuming 35% of step time, with no overlap with backward computation. Write the specific PyTorch code changes (DDP or FSDP configuration) that would enable overlap, and explain why each parameter matters.

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. You are designing the networking architecture for a 10,000-GPU training cluster with a $500M capex budget. Leadership is asking whether to use InfiniBand NDR 400G (higher NIC/switch cost, SHARP available) or RoCEv2 400G (lower switch cost, no SHARP, requires CTS-style congestion control). Walk through the decision framework quantitatively: calculate the switch cost difference at 10K GPUs, estimate the SHARP bandwidth savings for a 175B model AllReduce, factor in Meta's CTS engineering investment, and make a final recommendation with explicit assumptions.
2. A 4D parallel training job (TP=8, PP=8, CP=4, DP=8 = 2,048 GPUs) is achieving only 62% scaling efficiency. You have Nsight profiling data showing: AllReduce is 12% of step time (well-overlapped), but each training step shows 28 sequential NCCL barriers per step that are not overlapped with compute. Identify the specific parallelism dimension causing these sequential barriers, explain why it cannot be overlapped (unlike DP AllReduce), and propose two architectural changes —

one in training configuration, one in physical cluster placement — that would recover efficiency without changing the total GPU count.

3. The engineering team proposes increasing Tensor Parallelism from TP=8 (intra-node) to TP=16 (crossing two nodes over IB) to support a new 400B model that doesn't fit on 8 GPUs. Quantify the communication overhead at TP=16 over IB NDR 400G for a model with hidden_dim=16384 and 120 layers (assume each TP AllReduce message = hidden_dim × seq_len × 2 bytes, seq_len=4096). Compare this to the compute time for one transformer forward pass on H100. Then propose an alternative architecture that achieves the memory capacity goal without crossing IB with TP traffic.

✦ END OF CHAPTER 14 ✦

NEXT: CHAPTER 15 — STORAGE & I/O: FEEDING 10,000 GPUS WITHOUT STARVING THEM