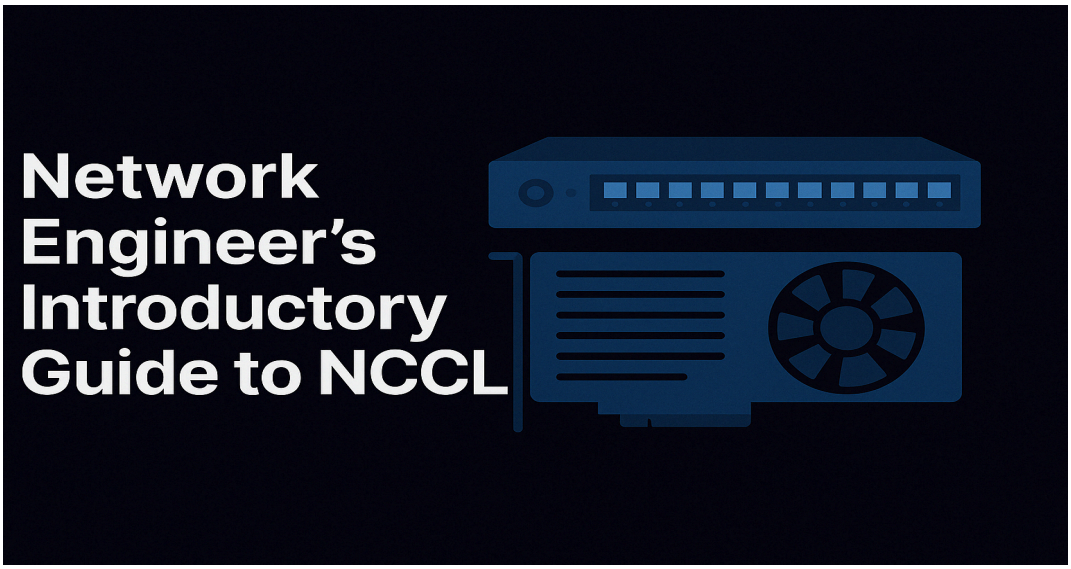
Bits & Bytes in Plain English

Follow

Network Engineers' Introductory Guide to NCCL

adel lazzag

Apr 29, 2025 · 12 min read



Show more

Introduction

In the rapidly evolving field of large language models (LLMs) and deep learning, training these complex models often requires distributed computing. This involves splitting the workload across multiple GPUs or even multiple nodes to achieve faster training times. However, running these parallel workloads necessitates a high level of synchronization and efficient communication between the GPUs.

NCCL, or the NVIDIA Collective Communications Library, is a powerful tool that enables deep learning frameworks like PyTorch and TensorFlow to coordinate communications between NVIDIA GPUs in a distributed manner, ensuring speed and efficiency. This library plays a crucial role in optimizing the collective communication operations essential for parallel and distributed computing.

Wait a second, I'm a network engineer. Why should I care?

As network engineers, we speak a certain language, one made up of bytes, bandwidth, switches, and links. The compute infrastructure and ML/AI application teams have their own technical jargon as well. But to design and operate the networks that power modern AI workloads, we need to understand what's actually running over them.

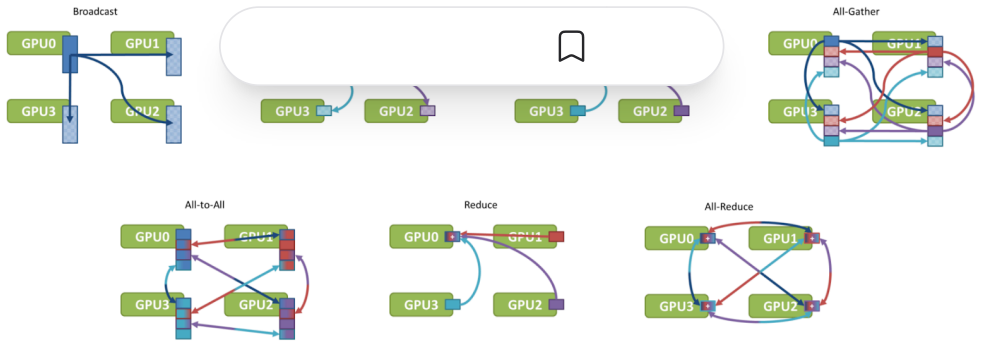
A GPU cluster works as a single system — which is why it's often called a supercomputer. Bridging the gap between teams means going beyond our own domain and learning enough about theirs to make smarter decisions and build better infrastructure.

What are Collective Communication Operations?

Imagine you're faced with a massive problem. If you were to work on it alone, the time required to solve it is directly proportional to the amount of effort you can personally invest. Now, imagine having a team of 10 people. You could break the problem into 10 smaller pieces, assign one piece to each person, and then combine the results. Instead of taking the full time to do everything yourself, the team solves the problem in parallel, completing it in roughly one-tenth the time. This is the core idea behind collective communication in distributed systems and parallel computing: splitting work across multiple processing units, executing in parallel, and efficiently reassembling the results.

Collective communication operations are fundamental communication patterns used in parallel and distributed computing scenarios. Depending on the deep learning algorithm configured in PyTorch or TensorFlow, these frameworks choose the most suitable GPU communication pattern for the specific use case. Some key collective operations include:

- **AllReduce** (Reduce + Broadcast): Combines data from all participating GPUs into a single value and copies it back to each GPU. Useful for tasks like calculating the total loss during training or summarizing statistics across all data.
- **Reduce**: Combines data from all participating GPUs into a single value and sends it to one GPU.
- **Broadcast**: Sends data from one GPU to all other participating GPUs. Ideal for sharing initial model parameters or sharing important updates with all GPUs.
- **Gather**: Gathers data from all GPUs and distributes it to one GPU, resulting in that GPU having a copy of all the data from all other GPUs.
- **AllGather**: Gathers data from all GPUs and distributes it to all GPUs, resulting in each GPU having a copy of all the data from all other GPUs.
- **ReduceScatter**: Splits the data into chunks, reduces each chunk across GPUs, and scatters each chunk back so that each GPU ends up with a distinct portion of the reduced result.
- **Scatter**: Sends data from one GPU and distributes it across multiple GPUs.



NCCL (pronounced nickel), also known as the Nvidia Collective Communications Library, is a library created to optimize and coordinate the collective communication operations explained above between Nvidia GPUs.

How Each GPU Receives “Ranks”

Each GPU participating in the logical job receives a unique identifier called a rank. This happens when MPI is initialized. Here is a high-level overview of the order of operations:

1. Resource Allocation:
    - Orchestrators like SLURM or Kubernetes allocate the requested resources. For example, if the user requests 8 GPUs, the scheduler might allocate 2 nodes with 4 GPUs each.
2. Process Initialization:
    - The orchestrator initiates processes on each allocated node.
    - These processes initialize and set up the infrastructure for communication between them. Each process receives a unique identifier called a rank.
    - The specific mechanism for assigning ranks depends on the framework or system being used, such as MPI, TensorFlow, PyTorch, or custom scripts. Each process knows its rank and can establish connections with other processes based on predefined communication patterns (e.g., all-to-all, ring, tree).
3. NCCL Initialization:
    - NCCL initializes within each MPI process.
    - NCCL discovers the hardware topology and creates optimized communication paths based on the detected hardware. For example, if

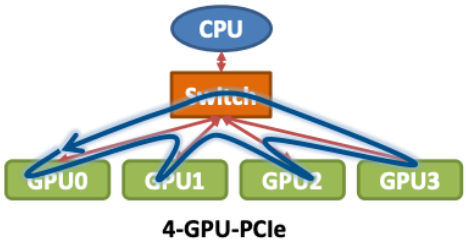
NCCL detects that 2 GPUs are interconnected using an NVSwitch, it will prefer using exchange.

Underlying collective communication topologies

During the collectives discussed above, NCCL orchestrates the communication paths between GPUs necessary for exchanging the required data. We can see the communication patterns in two ways:

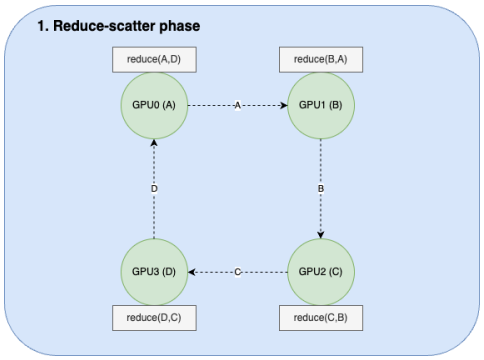
Ring-based topology

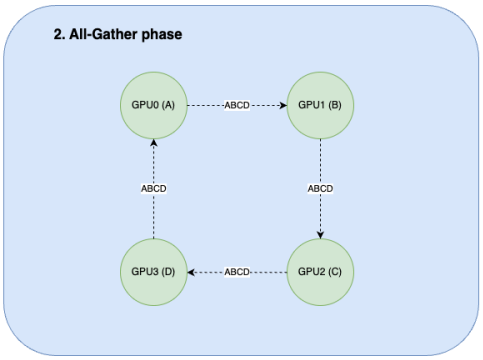
In ring-based topology, each GPU communicates with a GPU that NCCL determined to be its neighbor. It determines that by using different parameters such as bandwidth, interconnect type, inter-intra node GPU, etc. As seen below, this ring topology can be applied to multiple topologies.



4-GPU-PCIe

Practical example of ring-based topology:



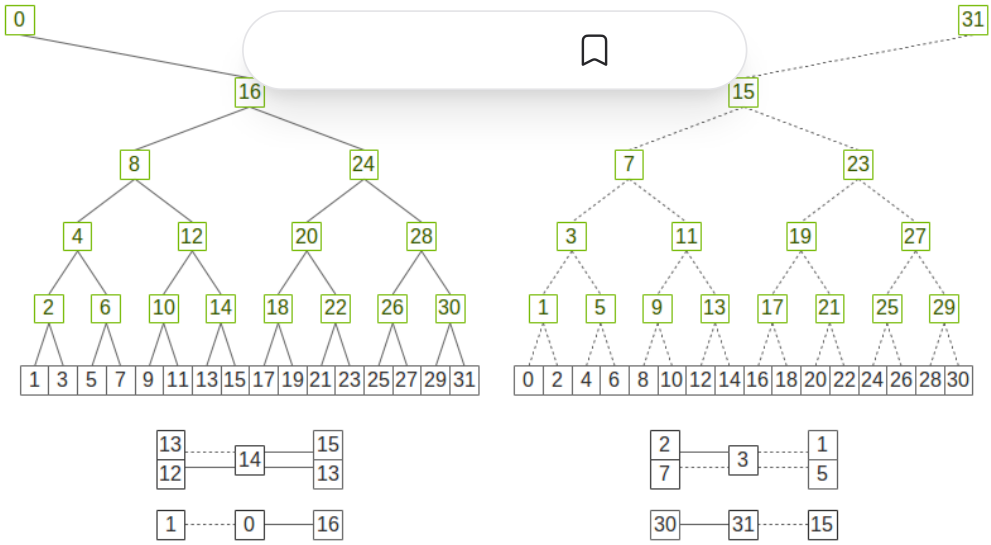


Each arrow actually represents N–1 sub-steps (one chunk per hop), but I've collapsed them into a single pass here for simplicity. More info on the exact steps in this NCCL presentation slide deck.

Tree-based topol

In tree-based topology, GPUs are organized hierarchically. This is useful for operations like reductions and broadcasts where data can be aggregated or distributed in a hierarchical manner. The tree structure is built such that communication flows efficiently from leaf nodes to the root (for reductions) or from the root to the leaf nodes (for broadcasts).

- Reduction Operation
  - Tree construction:
    - A single tree structure is constructed with one GPU designated as the root.
    - Other GPUs are organized hierarchically below the root in a way that minimizes communication latency and maximizes bandwidth.
  - Data Aggregation:
    - Leaf nodes send their data to their parent nodes.
    - Intermediate nodes receive data from their children, perform partial reductions, and send the results up to their parent nodes.
    - The root node receives the final partial results and performs the final reduction.
- Broadcast Operation
  - Tree Construction:
    - A single tree structure is constructed with one GPU designated as the root.
    - Other GPUs are organized hierarchically below the root.
  - Data Distribution:
    - The root node sends the data to its child nodes.
    - Intermediate nodes receive data from their parent nodes and forward it to their child nodes.
    - Leaf nodes receive the final broadcasted data.



Point-to-point communications

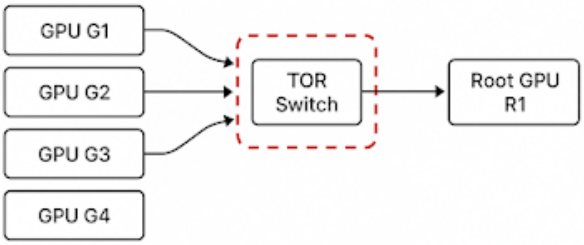
Point-to-point communication in NCCL involves direct data exchange between pairs of GPUs. This type of communication is different from collective operations, as it only involves two GPUs at a time, rather than coordinating across multiple GPUs.

Common network issues during collective communications

Incast Congestion and Buffer Drops

> *Too many senders, one receiver.*

In distributed GPU training, certain collective operations like gather and reduce can cause many GPUs to send data simultaneously to a single destination. This creates a many-to-one communication pattern, known as incast. This usually happens in tree-based topologies in large-scale networks as data is moving up the tree and is aggregated by intermediate nodes.

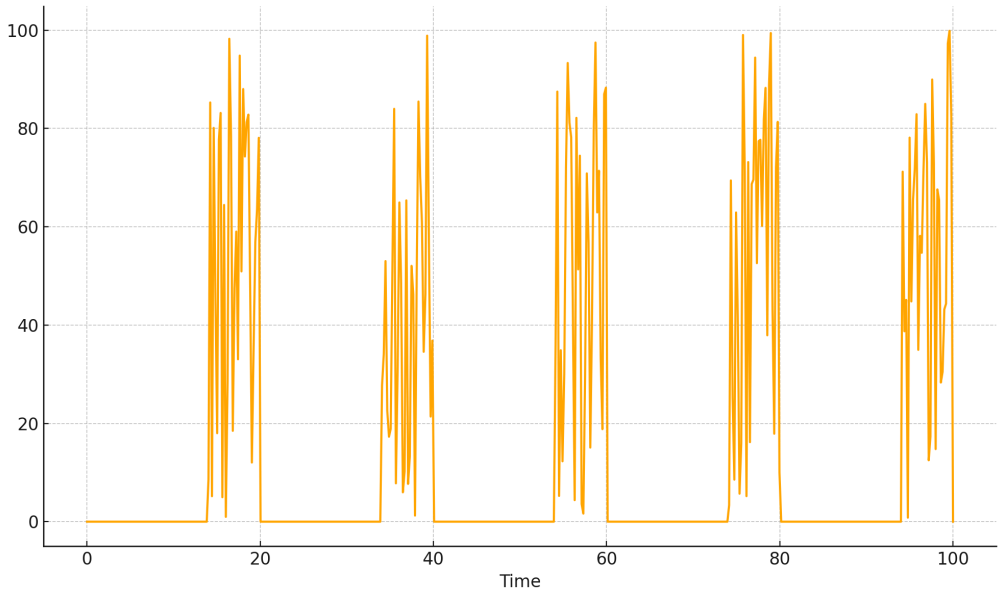


For example, assume GPUs on leaves L1, L2, L3, and L4 all sending data to a root GPU (R1) via the spine. Each sends at 100 Gbps but the receiver's link or switch port only supports 100 Gbps, the total ingress exceeds capacity. In traditional Ethernet networks, this oversubscription results in buffer overflows and packet drops — a critical issue in training workloads where retransmissions are costly or unsupported.

These drops are especially problematic in direct gather or reduce collectives, where synchronized communication is required and recovery from packet loss may not be automatic.

Bursty traffic

During a model training workload, GPUs alternate between computing and communication phases. This means they switch from performing calculations to exchanging results across the cluster. If you were to analyze a GPU switch port using a network monitoring tool, the graph you would see might look like this:



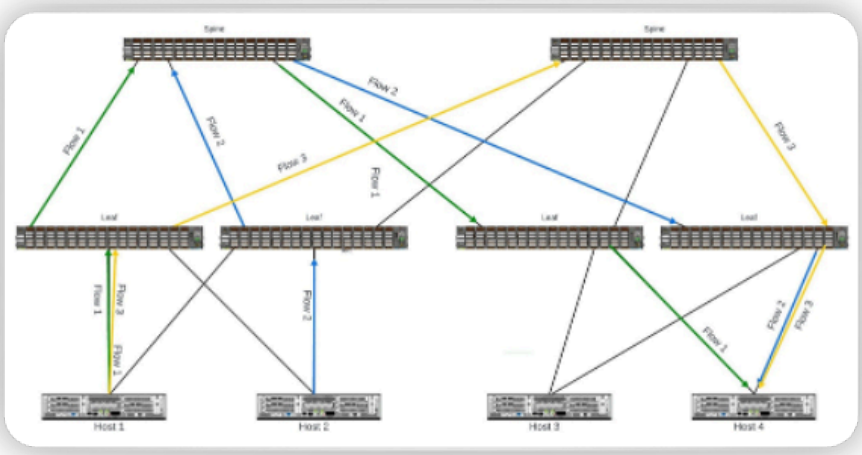
As you can see, the traffic is very bursty and is often close to line rate. Note that the above is just for conceptual understanding; real-world results may vary. For example, modern NCCL tries to overlap compute and comm, so you may see some “baseline” chatter during the heavy compute sections rather than perfect silence.

This communication pattern is challenging in traditional Ethernet networks using TCP as the transport layer because of how TCP handles congestion. TCP congestion control relies on packet loss. It "probes" the network for congestion signs, and if none are found, it keeps increasing the traffic it sends (known as

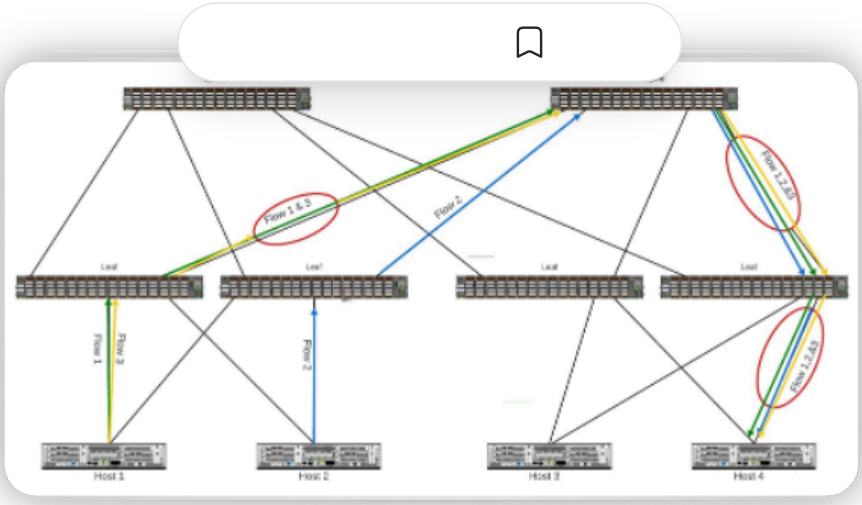
the congestion windo[illegible]d (through duplicate acknowledgments, r[illegible] the traffic it sends. The issue is that when traffic is bursty, as shown above, TCP may overreact to short bursts and reduce the congestion window, limiting network throughput. Another problem with TCP is that it needs a signal from the network to detect congestion, and AIML networks must be lossless. By the time TCP congestion control activates, most of the burst has already been dropped. There are efforts to improve TCP to prevent this behavior, such as DCTCP (RFC 8257), but it's still not specifically designed for AIML networks.

Elephant Flows

In general-purpose computing, you have a large number of small flows that are usually short-lived. This makes load-balancing those flows using ECMP hash-based forwarding very efficient as it will map each individual flow to a different path, avoiding too many flows saturating a certain path and creating a hotspot as we see in the figure below.



AIML compute, on the other hand, has a few number of very large flows which are long-lived. Because NCCL typically uses the same destination port (4791), there's a risk that ECMP hashing might route multiple large flows through the same network path, creating hotspots. Because of that, all flows in a job will be mapped to the same ECMP path, which has the potential to saturate links and create hotspots as we see in the figure below.



NCCL Tests

Before transitioning a GPU compute cluster to production, infrastructure teams typically run NCCL tests, which is a library designed to validate and benchmark the performance of collective communication across the entire cluster. The NCCL Test executes specific collectives based on the user's choice, for example, all-reduce, and runs them once for each data size. An example of the NCCL Test output is below:

```
ladmin@gh200-1:~/nccl-tests$ mpirun -np 2 -host localhost,172.16.10.12 ./build/all_reduce_perf -b 8 -e 2048M -f 2 -g 1
# nThread 1 nGpus 1 minBytes 8 maxBytes 2147483648 step: 2(factor) warmup iters: 5 iters: 20 agg iters: 1 validation: 1 graph: 0
#
# Using devices
#  Rank  0 Group  0 Pid 204234 on    gh200-1 device  0 [0x01] NVIDIA GH200 480GB
#  Rank  1 Group  0 Pid 148252 on    gh200-2 device  0 [0x01] NVIDIA GH200 480GB
#
#                                                              out-of-place                       in-place
#       size         count      type   redop    root     time   algbw   busbw #wrong     time   algbw   busbw #wrong
#        (B)    (elements)                               (us)  (GB/s)  (GB/s)            (us)  (GB/s)  (GB/s)
           8             2     float     sum      -1    22.86    0.00    0.00      0    20.03    0.00    0.00      0
          16             4     float     sum      -1    20.11    0.00    0.00      0    20.47    0.00    0.00      0
          32             8     float     sum      -1    20.50    0.00    0.00      0    21.37    0.00    0.00      0
          64            16     float     sum      -1    21.02    0.00    0.00      0    20.82    0.00    0.00      0
         128            32     float     sum      -1    21.30    0.01    0.01      0    21.06    0.01    0.01      0
         256            64     float     sum      -1    21.74    0.01    0.01      0    21.68    0.01    0.01      0
         512           128     float     sum      -1    21.71    0.02    0.02      0    20.78    0.02    0.02      0
        1024           256     float     sum      -1    21.74    0.05    0.05      0    22.06    0.05    0.05      0
        2048           512     float     sum      -1    22.36    0.09    0.09      0    22.60    0.09    0.09      0
        4096          1024     float     sum      -1    23.10    0.18    0.18      0    23.07    0.18    0.18      0
        8192          2048     float     sum      -1    23.73    0.35    0.35      0    23.41    0.35    0.35      0
       16384          4096     float     sum      -1    25.82    0.63    0.63      0    26.95    0.61    0.61      0
       32768          8192     float     sum      -1    30.82    1.06    1.06      0    30.89    1.06    1.06      0
       65536         16384     float     sum      -1    36.34    1.80    1.80      0    36.06    1.82    1.82      0
      131072         32768     float     sum      -1    47.29    2.77    2.77      0    48.62    2.70    2.70      0
      262144         65536     float     sum      -1    64.60    4.06    4.06      0    60.23    4.35    4.35      0
      524288        131072     float     sum      -1    102.9    5.09    5.09      0    101.4    5.17    5.17      0
     1048576        262144     float     sum      -1    306.6    3.42    3.42      0    499.7    2.10    2.10      0
     2097152        524288     float     sum      -1    705.3    2.97    2.97      0    748.6    2.80    2.80      0
     4194304       1048576     float     sum      -1    137.9   30.43   30.43      0    137.4   30.53   30.53      0
     8388608       2097152     float     sum      -1    230.9   36.33   36.33      0    230.8   36.35   36.35      0
    16777216       4194304     float     sum      -1    425.4   39.44   39.44      0    423.2   39.64   39.64      0
    33554432       8388608     float     sum      -1    819.4   40.95   40.95      0    819.0   40.97   40.97      0
    67108864      16777216     float     sum      -1   1567.9   42.80   42.80      0   1565.7   42.86   42.86      0
   134217728      33554432     float     sum      -1   3006.2   44.65   44.65      0   3009.8   44.59   44.59      0
   268435456      67108864     float     sum      -1   5864.0   45.78   45.78      0   5855.7   45.84   45.84      0
   536870912     134217728     float     sum      -1    11483   46.75   46.75      0    11501   46.68   46.68      0
  1073741824     268435456     float     sum      -1    22657   47.39   47.39      0    22609   47.49   47.49      0
  2147483648     536870912     float     sum      -1    44845   47.89   47.89      0    44832   47.90   47.90      0
# Out of bounds values : 0 OK
# Avg bus bandwidth    : 15.3294
#
```

Let’s take some time to decipher what we see in the output above:

- Size: The size of the message used to evaluate performance. The larger the size, the more data is being passed around.

- Count: The numb[illegible]xample, if we have 2 counts for 8 Byt[illegible]
- Type: Data type of each element.
- Redop: The reduction operation being used. This is when we are doing reduce or all-reduce tests.
- Root: Rank of the root node. This is for collectives where the source is one GPU, such as Reduce or Broadcast. In collectives like All-Reduce, which are symmetric, meaning that data is being passed around through all GPUs, we will say -1 as we see above.

The next 6 columns are divided into 2 sections, Out-of-place and In-Place:

- **Out-of-place** requires two device buffers per GPU, a send-buffer and recv-buffer. It will copy the contents of each GPU's send-buffer to the neighbor's receive buffer, always keeping a copy of the original send buffer, thus keeping the original data intact. This requires two buffers worth of space and has some overhead for the extra write to the receive buffer.
- **In-place** requires only one device buffer and overwrites the buffer at each data exchange with the neighbor. NCCL-Tests runs both modes because they represent two common real-world usage patterns—and their performance can differ. We want the full picture of the performance of all scenarios here.

Here are the 3 out-of-place and in-place columns we see above:

- Time: Time it took to complete the collective operation.
- Algbw: Now pay attention here, this is where it's really important for network engineers. Algbw is a simple calculation of time divided by data size, so if you look at the bottom row data size, which is message size (GB) / time (s) = 2.147483648 GB ÷ 0.044848 s ≈ 47.89 GB/s.
- Busbw: Algbw is a simplistic calculation that doesn't take into account the different collective communication patterns. For example, in an all-reduce, the traffic goes once through the ring to accumulate every GPU's data, and then the data goes through the ring a second time to distribute the result, so the bandwidth effectively goes through the ring 2 times. In large-scale environments with a large number of GPUs, it's not uncommon to see that the busbw is double the algbw. The above example is only between 2 GPUs, so it does not fully demonstrate this concept.

FAQ

Are same rank GP[illegible]other or is it any-to-any communi[illegible]

In a multi-node multi-GPU setup, the communication between nodes will often involve GPUs of the same ranks, but it is not exclusively limited to this pattern. The communication strategy depends on the specific collective operation being performed and the optimization chosen by NCCL based on the hardware topology. Here are some examples.

All-Reduce:

- Intra-Node Reduction: Each node first performs the reduction operation among its own GPUs.
- Inter-Node Reduction: GPUs with the same rank on different nodes then communicate with each other to further reduce the data.
- Intra-Node Broadcast: The reduced results are broadcast back to all GPUs within each node.

All-Gather:

- Intra-Node Gathering: Each node gathers data from its own GPUs.
- Inter-Node Gathering: GPUs with the same rank across nodes gather data from each other.
- Intra-Node Distribution: The gathered data is distributed back to all GPUs within each node.

How do we know which optimization is chosen by NCCL and how can we influence it?

NCCL automatically selects the optimal communication strategy based on the hardware topology and the collective operation being performed. To determine the optimizations chosen by NCCL, you can enable debugging and logging via the NCCL_DEBUG environment variable and use profiling tools like NVIDIA Nsight Systems and Nsight Compute. To influence NCCL's optimizations, you can set various environment variables such as NCCL_ALGO, NCCL_PROTO, NCCL_NTHREADS, and more. These settings allow you to customize and optimize NCCL's behavior for your specific hardware and workload.

References

NCCL Reference docs

https://github.com/NVIDIA/nccl-tests

https://network.nvidia.com/sites/default/files/related-docs/solutions/hpc/paperieee_copyright.pdf

https://outshift.cisco.com/blog/training-llms-efficient-gpu-traffic-routing

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

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html

https://www.ietf.org/archive/id/draft-yao-tsvwg-cco-problem-statement-and-usecases-00.html

adel lazzag

Ultra Ethernet 1.0 Specification: What Is Libfabric ?

Introduction Hello everyone! It’s been a while. I am immersing myself in the UE spec once again and ...

What You Need to Know About Artificial Intelligence

What is AI ? AI stands for artificial intelligence, it is rather a broad field of study than one spe...

adel lazzag

Discover the Power of Learning by Doing

Reid Hoffman once said "I believe starting a company is like jumping off a cliff and assembl