Multi GPU
Part 2

Prateek Shukla

Slurm

SLURM is an open-source workload manager and job scheduler designed for Linux-based high-performance computing clusters.

It allocates the GPUs, CPUs, and memory. It knows where your job is running but doesn't necessarily know how your application talks to itself across those nodes.

In the old days, mpirun would use SSH to connect to every node, check hostnames, and exchange keys. On large clusters (e.g., 64+ nodes), this "handshake" could take minutes

With `srun --mpi=pmix`, Slurm launches the processes on all nodes simultaneously.

PMIx (Process Management Interface for Exascale)

When Slurm launches your 1,000 processes. Now rank 5 (Process #5) knows it is alive, but it doesn't know the IP address of Rank 0, or how to talk to Rank 999. They are isolated islands.

Before heavy training begins, the processes need to exchange technical details (IP addresses, GPU handles). PMIx provides a temporary database.

`PMIx_Put`: A process pushes its metadata (e.g., host IP, port, CUDA IPC handle) to the local PMIx server.

`PMIx_Commit`: Pushes local data to the global namespace.

`PMIx_Get`: Other processes query this data to discover their peers.

Why SLURM and PMIx are important together

Slurm alone handles resource allocation (nodes, GPUs, CPUs) and task placement, but it historically relied on older PMI versions that don’t scale well beyond ~10k GPUs.

PMIx is the modern, exascale-ready process management interface (replacing PMI-1/PMI-2).

Slurm + PMIx integration allows:

Direct launch of processes without mpirun in many cases.

Efficient job info exchange (rank, node list, endpoints) at scale.

Tight integration with NCCL via direct NCCL-PMIx plugins.

SLURM scripts

It is a simple text file (usually ending in .sh or .sbatch) that tells Slurm two things:

What resources you need (Time, GPUs, CPUs, Memory).

What you want to do once you get those resources (Run Python, compile code, etc.).

Once we write the script we need to launch using `sbatch script.sh`

Once the script is launched, Slurm logs into the compute node, sets up the environment, and runs the commands listed in the script.

Terminology

Job: The Job is the highest-level unit of work in Slurm. It represents the resource allocation. When you run sbatch or salloc, you are creating a Job.

Task: A Task is a single process of the running application. It is the number of processes which run per node/gpu.

Rank: rank is simply the ID of a task, unless specified it refers to the global rank which is the rank in the whole job.

Local rank: This is the unique ID of a task within a specific node. Ranges from 0 - Number of tasks run on a single node

Namespace: for every rank inside a job this is the jobID. Unique for every rank in the job

Device isolation

If you set slurm to use a single GPU per task then slurm doesn't just politely ask your program to use one GPU; it forces it at the operating system level using two mechanisms:

CUDA_VISIBLE_DEVICES: Slurm sets this environment variable inside the process. This prevents CUDA to see any other devices than the ones listed here. Works for a single node

Slurm uses the Linux Control Groups (cgroups) feature, specifically the device allowlist where it creates a "sandbox" for Task 0.

This means that if you debug and every GPU have different task then when you look at logs every single error would point to GPU 0 even if its not actually GPU 0

A sample script

```
#!/bin/bash

# ------------- PART 1: DIRECTIVES -------------
#SBATCH --job-name=h100_training_run      # A clear name
#SBATCH --partition=gpu_h100              # (Check your cluster's partition name!)
#SBATCH --account=your_lab_group          # Account for billing/priority

# Resource Request
#SBATCH --nodes=1                         # We want 1 DGX node
#SBATCH --ntasks-per-node=8               # 8 Tasks (Processes) per node
#SBATCH --gpus-per-task=1                 # 1 GPU per Task
#SBATCH --cpus-per-task=12                # 12 CPU cores per Task (Keep the GPU fed!)
#SBATCH --time=02:00:00                   # Run for max 2 hours

# Logging (The "Pro" way)
#SBATCH --output=logs/%x_%j.out           # Save to logs folder: jobname_jobID.out
#SBATCH --error=logs/%x_%j.err            # Save errors separately

# ------------- PART 2: ENVIRONMENT -------------
echo "Job started at: $(date)"
echo "Running on node: $(hostname)"

# 1. Clean the environment
module purge

# 2. Load necessary modules (Customize this for your cluster!)
# module load python/3.10
# module load cuda/12.2

# 3. Activate your virtual environment
source ~/my_envs/pytorch_env/bin/activate

# 4. Debugging Information (CRITICAL for H100s)
echo "---------------------------------"
echo "Slurm Job ID: $SLURM_JOB_ID"
echo "Allocated Nodes: $SLURM_JOB_NODELIST"
echo "CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES"
echo "---------------------------------"

# ------------- PART 3: EXECUTION -------------

# Use 'srun' to launch the parallel processes.-
# It uses the --ntasks info from the header automatically.

srun python train_model.py \
    --batch_size 64 \
    --epochs 10 \
    --lr 0.001

echo "Job finished at: $(date)"
```

How SLURM scripts launch PMIx for kernels

When you run `srun --mpi=pmix ./my_cuda_kernel_app`, Slurm doesn't just fork processes blindly. It uses PMIx to orchestrate the startup.

After this we have to initialize PMIx using these steps:

1. Step 1: Connecting to slurm daemon
2. Step 2: Initialize PMIx, get namespace and global rank
3. Step 3: Get local rank,
4. Step 4: Use the Local Rank to select the specific GPU/GPUs
5. Step 5: finish the process

Initializing PMIx and connecting process to slurm

```c
#include <pmix.h>
#include <stdio.h>

int main(int argc, char **argv) {
    pmix_proc_t myproc;
    pmix_status_t rc;

    // Initialize the PMIx client.
    // &myproc is an OUTPUT parameter; PMIx will fill it with your identity.
    rc = PMIx_Init(proc: &myproc, info: NULL, ninfo: 0);

    if (rc != PMIX_SUCCESS) {
        fprintf(stream: stderr, format: "PMIx_Init failed: %d\n", rc);
        return -1;
    }

    // ... Your application logic here ...

    // Finalize before exiting to clean up the connection
    PMIx_Finalize(info: NULL, ninfo: 0);

    return 0;
}
```

Getting identity (global rank and local rank)

```c
void discover_gpus(pmix_proc_t *me) {
  // 1) Local rank from PMIx (rank within this node)
  int local_rank = -1;
  pmix_value_t *val = NULL;
  if (PMIx_Get(proc: me, key: PMIX_LOCAL_RANK, info: NULL, ninfo: 0, val: &val) == PMIX_SUCCESS && val) {
    local_rank = (val->type == PMIX_UINT16) ? (int)val->data.uint16 : (int)val->data.integer;
    PMIX_VALUE_RELEASE(v: val);
  }

  // 2) Visible GPUs after Slurm/CUDA filtering
  int nvis = 0;
  cudaGetDeviceCount(&nvis);
  const char *cvd = getenv(name: "CUDA_VISIBLE_DEVICES");

  // 3) Optional: map visible indices back to physical IDs for logging
  int dev = (nvis > 0) ? (local_rank % nvis) : 0;
  char pci[32] = {[0]=0};
  cudaDeviceGetPCIBusId(pci, (int)sizeof(pci), dev);

  fprintf(stream: stderr, format: "local_rank=%d, nvis=%d, CVD=%s, chosen_dev=%d (%s)\n",
          local_rank, nvis, cvd ? cvd : "", dev, pci);
}
```

Slurm typically restricts GPU visibility per task by setting `CUDA_VISIBLE_DEVICES`, so your process sees a filtered, re-indexed set of GPUs starting at device 0.

PMIx provides the node-local rank (`PMIX_LOCAL_RANK`), which is the natural index to use when distributing tasks across the GPUs assigned to that node.

Use cudaGetDeviceCount to learn how many devices are visible after filtering, and optionally parse `CUDA_VISIBLE_DEVICES` if you need original physical indices.

For deeper debugging, query PCI bus IDs (`cudaDeviceGetPCIBusId`) so you can correlate “visible device 0” with the actual hardware GPU Slurm allocated.

Binding to local GPU

```c
void bind_gpu(const pmix_proc_t *me) {
  // local_rank ∈ [0..tasks_on_node-1]
  pmix_value_t *val = NULL;
  pmix_proc_t self = *me;
  int local_rank = 0, nvis = 0;

  if (PMIx_Get(proc: &self, key: PMIX_LOCAL_RANK, info: NULL, ninfo: 0, val: &val) == PMIX_SUCCESS && val) {
    local_rank = (val->type == PMIX_UINT16) ? (int)val->data.uint16 : 0;
    PMIX_VALUE_RELEASE(v: val);
  }

  cudaGetDeviceCount(&nvis);
  int dev = (nvis > 0) ? (local_rank % nvis) : 0;

  cudaError_t e = cudaSetDevice(dev);
  if (e != cudaSuccess) {
    fprintf(stream: stderr, format: "cudaSetDevice(%d) failed: %s\n", dev, cudaGetErrorString(e));
  }
}
```

After this launch the kernels and once its done, use `PMIx_Finalize(NULL, 0);` in the host function to tell SLURM we are done

So what is NCCL

NVIDIA Collective Communications Library. It's a GPU-focused library that makes it fast (and relatively painless) to do collective communication between GPUs

In distributed training, GPUs constantly need to exchange tensors (gradients/parameters). NCCL provides highly optimized primitives for that, such as Allreduce, Allgather, ReduceScatter etc.

It integrates at the CUDA stream + pointer level: you hand NCCL device pointers and a `cudaStream_t`, and NCCL schedules its own GPU work (kernels + memcopies + network ops) on that stream, so it composes with your kernels via normal CUDA stream ordering.

When you do an allReduce operation the CPU just launches a NCCL Kernel onto the GPU stream.

Setting up communication with NCCL

Before any communication happens, NCCL needs a way for all processes to find a common meeting point. This meeting point is represented by a piece of data called the NCCL Unique ID.

Only one process can generate this ID, Every single process that wants to be part of the group must possess this exact same ID.

Rank 0 asks NCCL to create a new Unique ID, takes it and puts it into the PMIx Key-Value Store (KVS) so that other processes can read.

Rank 0 tells PMIx to commit the data, ensuring it is actually pushed out to the server and visible to the network. A fence ensures everyone can read the data and then Followers check the Key Value Store and call NCCL initialization function

`ncclGetUniqueId`

When we begin the GPUs don’t know about any other GPU in their surroundings and how to reach them using nvlink/infiniband.

To communicate with other GPUs NCCL needs to build a ‘communicator’

NCCL calls `ncclGetUniqueId` to create `ncclUniqueId` struct and logs in the ip address of the first GPU. Now every GPU comes in and logs its own IP in this struct to communicate

We will see how to use PMIx to do this

usage

```
ncclUniqueId id;
ncclComm_t comm;
char key[] = "NCCL_UNIQUE_ID";

// --- A. Rank 0 Creates the ID ---
if (global_rank == 0) {
    NCCLCHECK(ncclGetUniqueId(&id));

    // Wrap the binary ID struct into a PMIx Value to publish it
    pmix_value_t value;
    value.type = PMIX_BYTE_OBJECT;
    value.data.bo.bytes = (char*)&id;
    value.data.bo.size = sizeof(id);

    // Put it in the Key-Value Store
    // PMIX_GLOBAL indicates data is available to all nodes
    PMIx_Put(PMIX_GLOBAL, key, &value);

    // Push to server
    PMIx_Commit();
    printf("Rank 0: Published NCCL ID.\n");
}
```

NCCL is very fast, but it doesn't know where the other GPUs are when it starts. Different processes are running on different GPUs. They have separate memory. Process B cannot see variables inside Process A.

We need to ensure that data committed by participants is “collected” and made available according to scope rules (e.g., PMIX_GLOBAL). We create a fence where nobody returns until everyone has arrived.

Then every single process needs to find the key value store, for that we must specify who owns it. To do that we use the nspace of the processes and set rank 0.

We use PMIx_Get to get the key which we posted through PMIx_Put, this key serves no security or verification purpose; it is purely for addressing the data.

Once that’s done we copy the NCCL unique ID bytes out of PMIx’s temporary buffer into our own local ncclUniqueId id variable.

```c
// --- B. The Barrier (Wait for Rank 0) ---
// Everyone waits here until the Commit is finished.
pmix_info_t info; // sometimes needed for fence directives, can be NULL often
PMIx_Fence(NULL, 0, NULL, 0);

// --- C. Others Retrieve the ID ---
if (global_rank != 0) {
    pmix_value_t *val;
    // Get data posted by Rank 0
    // Note: You might need to specify the proc ID of Rank 0 specifically
    // depending on your PMIx version strictness.
    // For standard cases, querying the namespace often finds it.
    // Construct the proc ID for Rank 0
    pmix_proc_t rank_zero_proc;
    strncpy(rank_zero_proc.nspace, myproc.nspace, PMIX_MAX_NSLEN);
    rank_zero_proc.rank = 0;

    if (PMIx_Get(&rank_zero_proc, key, NULL, 0, &val) == PMIX_SUCCESS) {
        // Copy the bytes back into our local ID struct
        memcpy(&id, val->data.bo.bytes, sizeof(id));
        PMIX_VALUE_RELEASE(val);
    } else {
        fprintf(stderr, "Rank %d failed to get NCCL ID\n", global_rank);
        exit(1);
    }
}
```

`ncclCommonInitRank`

This is the most expensive and complex function in the setup phase. It is where the ID maps to Hardware Connections, we log IDs of nrank ranks to the GPU0

Every rank looks at the `ncclUniqueId` struct. It extracts the IP:Port of Rank 0.

Every rank creates a standard TCP socket and connects to Rank 0.

Rank 0 waits until it receives exactly nranks connections.

It's a barrier. If Rank 799 is slow, everyone waits here.

Once the TCP connections are alive, they don't send data yet. They send metadata about their rank, connection type, device etc.

```
ncclResult_t ncclCommInitRank(
    ncclComm_t* comm,   // [Output] The resulting communicator object
    int nranks,         // Total number of GPUs in this group
    ncclUniqueId id,    // The shared meeting ID (must be same for all)
    int rank            // My ID (Who am I? 0, 1, 2, ... nranks-1)
);
```

```
    }

    // --- D. Initialize NCCL ---
    // This is the heavy lifting step.
    // It connects NVLinks within the node and IB across nodes.
    printf("Rank %d: Initializing NCCL...\n", global_rank);
    NCCLCHECK(ncclCommInitRank(&comm, 1024, id, global_rank));
    // Note: '1024' is the total number of ranks (nranks).
    // You should get this dynamically from PMIx (PMIX_JOB_SIZE).

    printf("Rank %d: NCCL Ready!\n", global_rank);
```

The next steps

Rank 0 acts as the architect. It analyzes the global graph to find the path with the highest bandwidth and lowest latency for operations like AllReduce.

It prioritizes NVSwitch for intra-node communication. It selects specific InfiniBand NICs for inter-node jumps. It decides whether to build Rings (latency-optimized) or Trees (bandwidth-optimized) for the communication pattern.

Rank 0 sends the specific "routing table" (who to send to, who to receive from) back to every rank. Inside a single node, Ranks map each other’s memory. Configure the NVSwitch to allow direct GPU-to-GPU memory access

Ranks identify their paired peers on other nodes and RDMA Handshake to ensure they can write directly into each other's memory buffers without CPU involvement.

`ncclCommDestroy`

Its the way to dismantling hardware paths we created earlier

It marks the communicator object as invalid for future kernel launches. If the GPU is currently executing an NCCL kernel (like an AllReduce inside a CUDA stream), `ncclCommDestroy` will not yank the memory out from under it immediately. It relies on internal reference counting

The 4MB-8MB scratchpad buffers in HBM that were allocated during Init are freed.

The staging areas in CPU RAM used for PCIe transfers are released

NvLink Mappings and Infiniband Queue Pairs are destroyed

NCCL Collective Primitives

On an H100 cluster we don't think in terms of send and receive. We think in terms of patterns of data manipulation across the NVLink fabric. Here are big 6 primitives

Broadcast — one GPU copies its data to all GPUs.

Reduce — all GPUs combine data, result lands on one GPU.

AllReduce — all GPUs get the reduced sum of everyone’s data.

AllGather — GPUs start with fragments, end with the full combined buffer.

ReduceScatter — reduce first, then split output so each GPU gets a different chunk.

All-to-All — every GPU sends a piece to every other GPU, and receives pieces back

4 Important types of parallelism in AI

There are 4 types of parallelism which one deals with when working with AI applications in these multi GPU systems

1. Data Parallelism
2. Tensor Parallelism
3. Pipeline Parallelism
4. Expert Parallelism

Let’s discuss each one of them in detail

Data parallelism

Data parallelism is the most common distributed training strategy used in deep learning. It allows you to train a model faster by distributing the input data across multiple GPUs, while replicating the model itself on every device

A copy of the entire model is placed on every GPU. The global batch of training data is split into smaller "mini-batches". Each GPU receives its unique slice of data. Every GPU performs forward and backward passes on its own piece of data. Before the optimizer updates the model weights, the system must ensure that every model copy stays identical. The gradients from all GPUs are aggregated (usually averaged).

For Data Parallelism, strictly speaking, you primarily rely on AllReduce, but Broadcast is essential for initialization

NCCL operations involved

AllReduce is the single most important operation in Data Parallel training. It happens at the end of every backward pass. GPU 1 has gradient g1, GPU 2 has gradient g2, etc. We need every GPU to have the average gradient $(1/N)\Sigma gi$.

AllReduce sums up vectors from all GPUs and distributes the result back to all GPUs.

Broadcast is typically used at the very beginning of training or when resuming from a checkpoint. During weight initialization, Rank 0 initializes the parameters and Broadcasts them to all other ranks. This guarantees that at step 0, all replicas are mathematically identical

Tensor Parallelism

If you try to load a model that is too big for a single GPU, the program will simply crash before computation even begins. Tensor Parallelism is used when the weight matrices are so big that you can’t do matrix multiplies in a single GPU

If you are training the model (not just running it), the memory requirements triple or quadruple. You don't just store the weights; you need to store gradients, optimizer states, activations.

What we do here is that we divide our matrix into pieces and distribute it to different GPUs for computation. Once the computation is done then we put all the results together.

Sequence parallelism

In standard Tensor Parallelism, we split the heavy Matrix Multiplications (Linear Layers) across GPUs. However, we do not split the operations that occur between the Linear Layers, specifically: LayerNorm, Dropout, GeLU/SiLU activations

In standard TP, the output of the AllReduce (after a Linear Layer) is a replicated tensor. This means if you have 8 GPUs, every single GPU stores an identical copy of the full activation matrix (size: [Sequence Length, Hidden Dimension]) just to perform LayerNorm or Dropout.

LayerNorm acts on the Hidden Dimension of a single token. It is independent across the Sequence Dimension. Thus we don't need to replicate the full sequence on every GPU. We can partition the sequence across GPUs for these operations.

Breaking the allreduce on TP

Standard TP uses AllReduce to synchronize the output of a matrix multiplication. Mathematically, AllReduce is actually a composition of two primitive operations:

```
AllReduce = ReduceScatter + AllGather
```

When using SP, instead of doing the AllReduce atomically (NCCL fusion), SP injects the LayerNorm/Dropout operations inside the communication loop

We perform the ReduceScatter, stop there, do our LayerNorm on the sharded data, and only AllGather later when we absolutely need the full data for the next Matrix Multiplication.

NCCL operations involved in TP

Broadcast: Used in Column-wise sharding to copy complete input matrices to each worker.

All-Gather: Used in Column-wise sharding to combine the results after multiplication, Used in Sequence Parallelism during the forward pass to combine sharded sequence chunks.

All-Reduce: Used in Row-wise sharding to sum the results from different workers for the final result. In a standard Transformer block, Tensor Parallelism implies two all-reduce per transformer block (one for the Attention block and one for the Feedforward block).

Reduce-Scatter: Used in Sequence Parallelism to reduce gradients while scattering along the sequence dimension. In TP, Reducescatter is used for gradients during the backward pass of a column linear operation.

Pipeline parallelism

If Tensor Parallelism is slicing a single layer horizontally, Pipeline Parallelism is slicing the model vertically (splitting the stack of layers).

By splitting the model layers across different GPUs, each GPU only needs to store the parameters and optimizer states for its specific chunk of layers.

If you simply passed one large batch through the pipeline, most GPUs would sit idle waiting for their turn. Pipeline Parallelism breaks a large batch of data into tiny micro-batches. As soon as GPU 1 finishes the first micro-batch, it passes it to GPU 2 and immediately starts working on the second micro-batch

Because PP only requires communication between "neighbors" in the pipeline, it only uses point to point send receive operations.

Expert parallelism

Unlike standard "dense" models where every parameter is used for every input, MoE models activate only a small subset of parameters (experts) for each input.

This is good for cost. You can increase the parameter count by 100x (adding more experts), but if you only select the top-2 experts per token, the compute cost remains relatively low.

The model learns that certain "experts" are good at coding, while others are good at creative writing. Expert parallelism ensures that when a coding token comes in, it is routed specifically to the device holding the "coding expert."

How expert parallelism works

A small network decides if the token needs expert A or expert B etc.

Experts are distributed across GPUs. GPU 1 holds Expert A & B. GPU 2 holds Expert C & D. GPU 3 holds Expert E & F and so on

If a token on GPU 1 needs Expert D (which is on GPU 2), it must be sent over the network to GPU 2. Eventually every GPU needs to send data to every other GPU. This is called “all to all” communication.

Specialized libraries like DeepEP expertise on such situations by bundling data to send efficiently across the inter node connections first, then distributing it quickly locally with faster nvlink connections.

General format of NCCL operations

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

`sendbuff, recvbuff, count, datatype` tell the GPU how many elements to move and how to interpret them.

`op` tells what math operation to perform

`comm` is the communication handle storing all the metadata for the node and cluster

`stream` helps specify the asynchronous behavior

Broadcast

The Broadcast operation in NCCL is a collective communication primitive where a single "Root" GPU sends a tensor of data to every other GPU in the communicator.

Generally it was implemented using a ring/tree operation but with H100 DGX, the Root GPU sends the data packet once to the NVSwitch. The NVSwitch itself physically replicates the packet to all 7 other ports (in a single node) simultaneously.

Because the switch handles the replication, the time to broadcast to 8 GPUs is roughly the same as sending to just 1 GPU.

ncclBroadcast

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

`root`: this is the rank of GPU holding the data we want to broadcast

`sendbuff`: this is the location of the buffer of data we want to broadcast

`recvbuff`: The destination pointer on all GPUs (including the root)(if they are same then in place reduction is enabled).

`Datatype`: if you pass ncclFloat8e4m3 or ncclFloat8e5m2, NCCL switches to using Hopper-specific intrinsic instructions

`comm`: The communicator object

`stream`: The CUDA stream

The math of the operation

Let P={0,1,...,N−1} be the set of N processes (GPUs) in the communicator. Let V(i) represent the data vector held by process i.

Initial State:

$$V^{(i)} = \begin{cases} D & \text{if } i = r \\ \emptyset \text{ or undefined} & \text{if } i \neq r \end{cases}$$

Operation:

$$\text{Broadcast}(r, V)$$

Final State:

$$\forall i \in \mathcal{P}, \quad V^{(i)} = D$$

Reduction

Reduction takes an array of data from every participating GPU, combines them using a mathematical operator (like sum or max), and stores the single consolidated result on a specific GPU (or all of them).

In previous systems, GPUs had to pass data between each other (like a bucket brigade) to sum it up. In the DGX H100, the NVSwitch chips themselves perform the math, offloading the work from the GPUs.

the general reduction (ncclReduce) typically uses NVLSTREE (introduced in NCCL 2.18+) to handle the specific flow of reducing data to a single root, especially when scaling across multiple nodes

ncclReduce

```
ncclResult_t ncclReduce(const void* sendbuff, void* recvbuff, size_t count,
                        ncclDataType_t datatype, ncclRedOp_t op, int root,
                        ncclComm_t comm, cudaStream_t stream);
```

You use `ncclReduce` when the consumer of the data is centralized (usually Rank 0) and the other ranks do not need the result to proceed immediately.

It is generally used in the last layer during inference when only Rank 0 needs the full logits to perform argmax or top-k sampling.

The count is the number of elements each GPU contributes. The root receives exactly count elements.

The General Formula

$$R[i] = V_0[i] \oplus V_1[i] \oplus \cdots \oplus V_{P-1}[i]$$

Where $\oplus$ is one of the supported reduction operators:

- **Sum (`ncclSum`):** $R[i] = \sum_{p=0}^{P-1} v_{p,i}$
- **Product (`ncclProd`):** $R[i] = \prod_{p=0}^{P-1} v_{p,i}$
- **Minimum (`ncclMin`):** $R[i] = \min(v_{0,i}, \ldots, v_{P-1,i})$
- **Maximum (`ncclMax`):** $R[i] = \max(v_{0,i}, \ldots, v_{P-1,i})$
- **Average (`ncclAvg`):** $R[i] = \frac{1}{P} \sum_{p=0}^{P-1} v_{p,i}$

Let P be the number of processors (GPUs), indexed 0 to P–1. Each processor p holds an input vector Vp of size N:

`Vp=[vp,0,vp,1,…,vp,N-1]`

AllReduce

AllReduce is one of the most important operations in training of AI models

Combines data from N GPUs -> get a global result -> distributes to N GPUs

All 8 GPUs read their local gradients from their HBM3 memory

Instead of sending data to a peer GPU, all 8 GPUs simultaneously push their data onto the NVLink lanes, targeting the NVSwitch chips

The data goes to switch where the reduction happens “in flight” the switch immediately multicasts this result back to all 8 GPUs simultaneously.

The result lands directly in the destination buffer in HBM3 on all GPUs.

ncclAllReduce

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

This is one of the most popular and important operation for deep learning use cases

AllReduce is the right primitive when every participant both contributes data and needs the final reduced result, such as gradient or metric aggregation across workers

ncclRedOp_t

This is the mathematical operation for reduction(`ncclSum, ncclProd, ncclMin, ncclMax, ncclAvg`)

`ncclSum` is the only operator that is fully optimized for every hardware path on H100. If you use this NCCL will offload all the operation to nvswitch

`ncclMin` and ncclMax are also offloaded from hardware(fp8)

`ncclProd` will offload all the operations to tensor cores because nvswitch can’t do floating point multiplications

`ncclAvg` is done in two parts, the sum part done in nvswitch and the division is done in GPUs

Math of AllReduce

Let there be P processes. Each process k holds a vector Vk of size N. Let Vk[i] denote the i-th element of the vector on process k.

The goal of AllReduce is for every process k to end up with a result vector R, where the i th element is the sum (or other associative operator $\oplus$) of that element across all processes

$$R[i] = V_0[i] \oplus V_1[i] \oplus \cdots \oplus V_{P-1}[i]$$

this is used to sum gradients: Vk is the gradient vector calculated by GPU k, and R is the total gradient used to update the model.

ReduceScatter

ReduceScatter is an operation where every GPU starts with a full buffer of gradients, and the goal is to sum these buffers across all GPUs, but then scatter the results so that each GPU ends up holding only a distinct "slice" of the final summed vector

Input: Each GPU has a vector [A,B,C,D].

Math: Element-wise sum across all GPUs

Output:

- GPU 0 holds Sum(Part A)
- GPU 1 holds Sum(Part B)
- GPU 2 holds Sum(Part C)
- GPU 3 holds Sum(Part D)

ncclReduceScatter

```
ncclReduceScatter(const void* sendbuff, void* recvbuff,
                  size_t recvcount, ncclDataType_t datatype,
                  ncclRedOp_t op, ncclComm_t comm, cudaStream_t stream);
```

`size_t recvcount:` This is the element count of the output buffer, **not** the total input buffer. You get this by total elements in gradients buffer / number of GPUs

```
Total Elements Reduced = recvcount * nranks
```

If you have a total vector size of 100 and 3 GPUs: You cannot divide 100 by 3 evenly (33.33...). NCCL does not support "jagged" arrays where GPU 0 gets 34, and GPU 1 gets 33. Thus you must pad your data to the next multiple of the GPU count

The math of ReduceScatter

Let P be the number of GPUs (ranks). Let each GPU i hold a vector Vi of length N. We divide the vector V into P blocks, denoted as B0,B1,...,BP−1.

GPU 0 holds $B^0_0 B^0_1 B^0_2 \ldots\ldots\ldots B^0_{n-1} B^0_n$

GPU 1 holds $B^1_0 B^1_1 B^1_2 \ldots\ldots\ldots B^1_{n-1} B^1_n$ and so on...

$$\text{Result on GPU}_k = \sum_{i=0}^{P-1} \text{Block}_k \text{ from GPU}_i$$

AllGather

AllGather is an operation where every GPU starts up with different pieces of the gradients and after the transformation every GPU gets the complete copy of the whole data

All 8 GPUs push their gradients into the NVSwitch fabric simultaneously

The NVSwitch combines the data and route the result to all the GPUs

Total bus utilization is maximized. You get closer to the theoretical 900 GB/s aggregate throughput because you aren't waiting for a "ring" to cycle

```
ncclResult_t ncclAllGather(const void* sendbuff, void* recvbuff,
                           size_t sendcount, ncclDataType_t datatype,
                           ncclComm_t comm, cudaStream_t stream);
```

`sendcount` is the number of elements this rank contributes (the size of its local slice). Every rank ends up with all ranks' slices in recvbuff.

Mathematically

Let P be the number of processes (nodes/GPUs), indexed 0 through $P-1$.

The Input: Each process i holds a vector (or tensor) v_i of size n.

$$v_i = [x_{i,1}, x_{i,2}, \dots, x_{i,n}]$$

The Output: At the end of the operation, **every** process i holds the exact same large vector V of size $P \times n$:

$$V = [v_0, v_1, v_2, \dots, v_{P-1}]$$

Mathemetically, this acts as a concatenation of the local buffers of all participants, replicated across all ranks.

All to All

Every GPU holds data for every other GPU in the node and it transfers that data to its respective GPU

The way it works is that each GPU receives its piece of data from every GPU and it sends data which belongs to every GPU

Unlike AllReduce, which often uses complex Ring or Tree algorithms and hardware offloading (SHARP), AllToAll on this system is physically simpler but bandwidth-intensive.

```
ncclResult_t ncclAllToAll(
    const void* sendbuff,
    void* recvbuff,
    size_t count,
    ncclDataType_t datatype,
    ncclComm_t comm,
    cudaStream_t stream
);
```

Assume nranks = N GPUs in a communicator.

You have a send buffer logically split into N blocks: block j is the data you want to send to rank j.

After all-to-all, each rank has a receive buffer split into N blocks: block i contains what rank i sent to you.

So it’s like doing N×(N−1) point-to-point transfers (plus self) but coordinated and optimized as one collective.

The MOE problem

Using NCCL's standard AllToAll for MoE models is generally avoided because it was designed for static, bulk synchronous communication, whereas MoE routing is inherently dynamic, sparse, and latency-sensitive.

Major problems with NCCL All to all include NCCL being CPU-driven (Host API), NCCL using static data sizes, NCCL requiring data in contagious blocks and just the blocking nature of the instruction.

Specialized libraries like DeepEP (and the underlying NVSHMEM primitives) are preferred because they allow for device-initiated, fine-grained data movement that can handle the irregular traffic patterns of expert routing without stalling the GPU.

ncclGroupStart ncclGroupEnd

In a distributed system a lot of times you have a single CPU thread managing multiple GPUs. when CPU launches a NCCL instruction the CPU will be blocked by NCCL if its a blocking call. Most NCCL collective calls (like ncclAllReduce) are asynchronous from the CPU, The more blocking parts are often communicator initialization (ncclCommInitRank),

If one host thread is issuing NCCL calls for multiple GPUs, launching them one-by-one can create partial states where some participants have started an operation while others haven’t been issued yet which can lead to hangs

ncclGroupStart()/ncclGroupEnd() let you batch a set of NCCL calls: NCCL collects the calls during the group and then submits them together at ncclGroupEnd(). This avoids problems caused by partially launched NCCL work.

```
// 1. Start the group
ncclGroupStart();

// 2. Queue multiple collective operations (or ops for different GPUs)
// These return immediately; they are NOT executed yet.
ncclBroadcast(buff1, ..., comm, stream);
ncclAllReduce(buff2, ..., comm, stream);
ncclReduceScatter(buff3, ..., comm, stream);

// 3. End the group
// NCCL now optimizes the DAG of operations and launches them to the GPU.
ncclGroupEnd();

// 4. Synchronization (Optional but standard)
// Only sync *after* GroupEnd.
cudaStreamSynchronize(stream);
```

ncclSend

```
ncclResult_t ncclSend(
    const void* sendbuff, // Source buffer (on device)
    size_t count,         // Number of elements to send
    ncclDataType_t type,  // Data type (e.g., ncclFloat, ncclInt)
    int peer,             // Rank of the destination GPU
    ncclComm_t comm,      // The communicator object
    cudaStream_t stream   // The CUDA stream to enqueue the op
);
```

It is a non-blocking operation used to send data from one specific GPU (the sender) to another specific GPU (the receiver). It is almost always paired with a corresponding ncclRecv on the target device.

peer: The rank (ID) of the destination GPU.

NCCL finds the fastest physical path available between the two GPUs, can be using NVLink, PCIe or IB RDMA

ncclRecv

```
ncclResult_t ncclRecv(
    void* recvbuff,       // Destination buffer (on device)
    size_t count,         // Number of elements to receive
    ncclDataType_t type,  // Data type (must match sender)
    int peer,             // Rank of the source GPU
    ncclComm_t comm,      // The communicator object
    cudaStream_t stream   // The CUDA stream to enqueue the op
);
```

Tells a specific GPU to allocate space in its memory and wait for incoming data from a designated "peer." It is asynchronous, meaning the CPU resumes control immediately after the command is enqueued on the CUDA stream

peer: The rank (ID) of the GPU that is sending the data.

A ncclRecv must have a matching ncclSend on the source GPU. If you are doing multiple P2P transfers, they must be enqueued in a compatible order across the different GPUs to avoid circular dependencies

Prevent all GPUs from launching the same instruction

```
int my_rank;
ncclCommUserRank(comm, &my_rank);

ncclGroupStart();

if (my_rank == 0) {
    ncclSend(send_buff, count, type, 1, comm, stream);
}

else if (my_rank == 1) {
    ncclRecv(recv_buff, count, type, 0, comm, stream);
}

ncclGroupEnd();
```

Probably end here