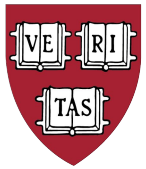
Kempner INSTITUTE



HARVARD UNIVERSITY

Introduction to Distributed Computing

October 10, 2024

Objectives

By the end of this workshop, you will be able to:

- Understand distributed computing, different forms of communication between computers, and how to determine when distributed computing could be helpful for your research
- Use array jobs for hyperparameter sweeps, an example of embarrassingly parallel processes
- Use Distributed Data Parallel (DDP) when training multi-layer perceptrons in PyTorch

Setting up conda environment

1.1. Create conda environment

Creating the conda envireonment named `dist_computing` (one can use their own customized name).

```
conda create -n dist_computing python=3.10
```

1.2. Installing PyTorch

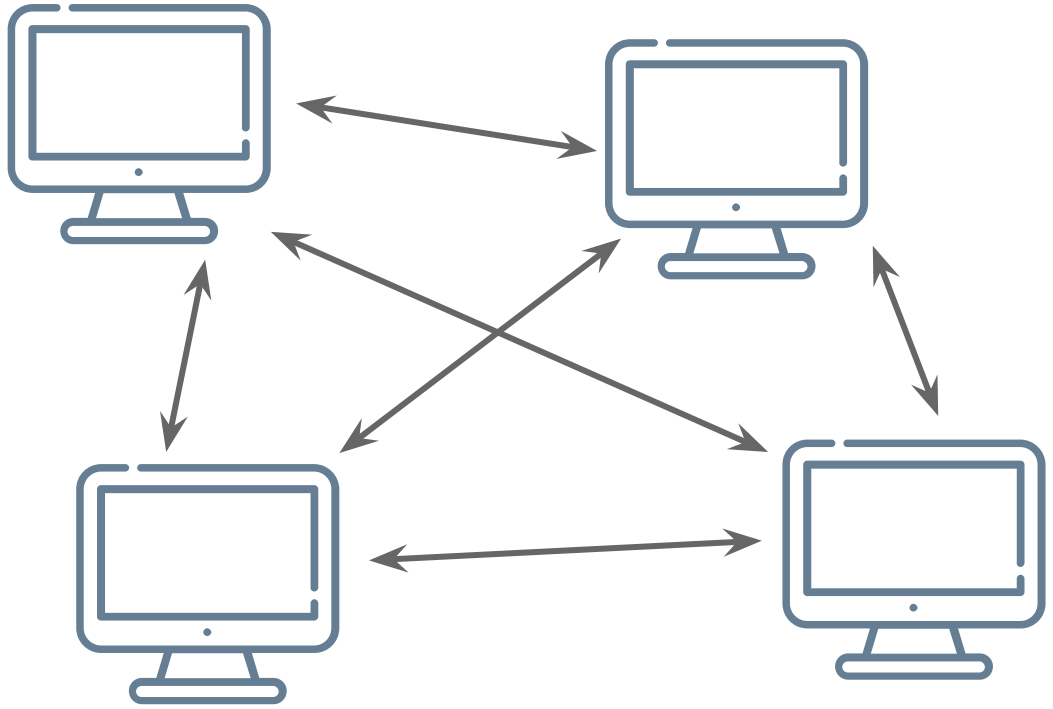
```
# Activating the conda environment and install PyTorch:
conda activate dist_computing
pip3 install torch
```

Agenda

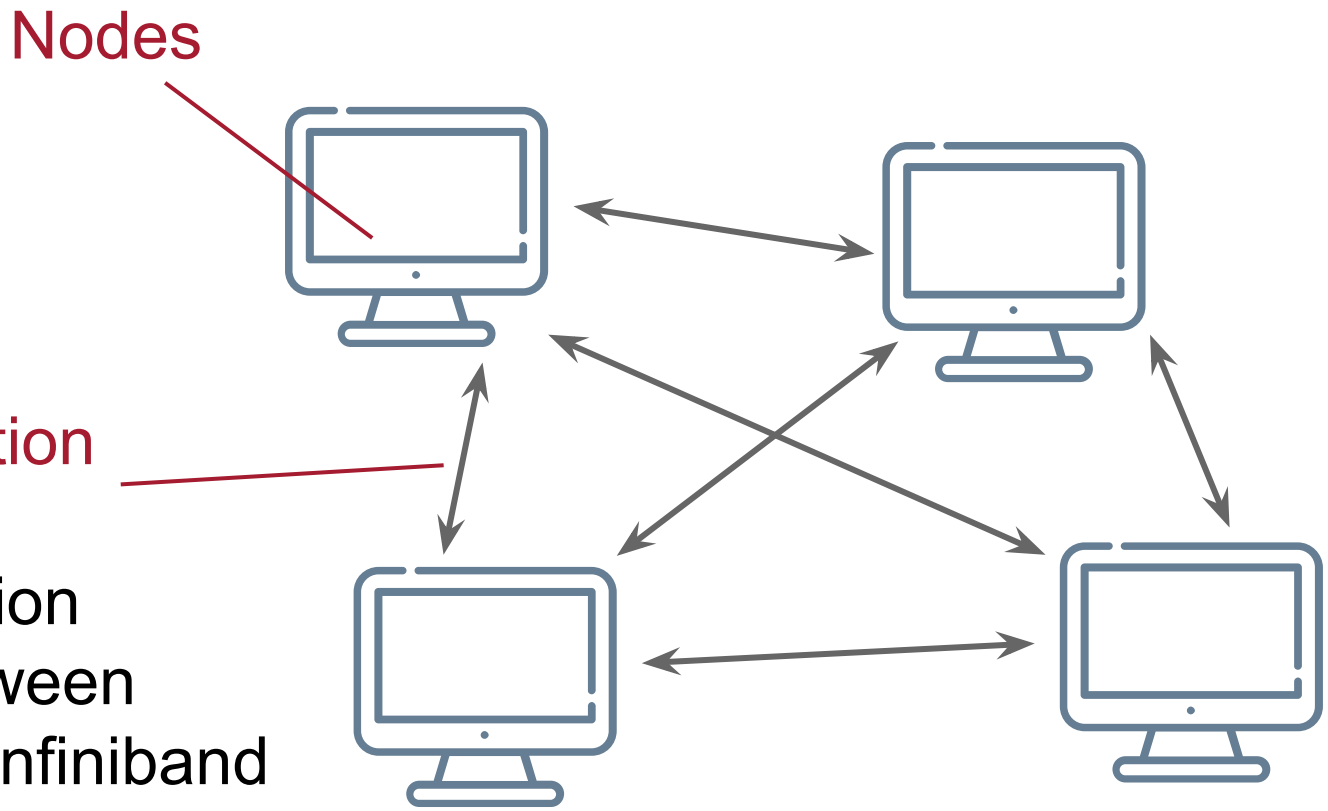
1. The Basics
2. Embarrassingly Parallel Processing
3. Distributed Data Parallel (DDP) Processing
4. Beyond DDP

What is distributed computing?

Multiple computers or nodes working together to solve a problem that is either large to fit in one computer, or takes time to process data with one computer



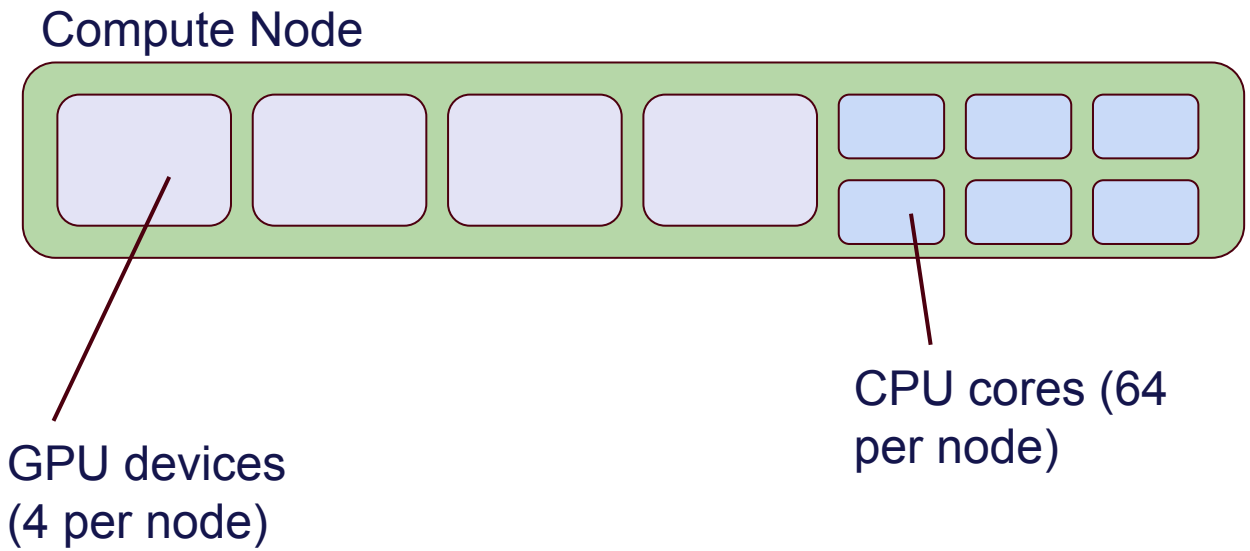
Components of Distributed Computing



Communication
High-speed communication network between nodes (e.g. infiniband network)

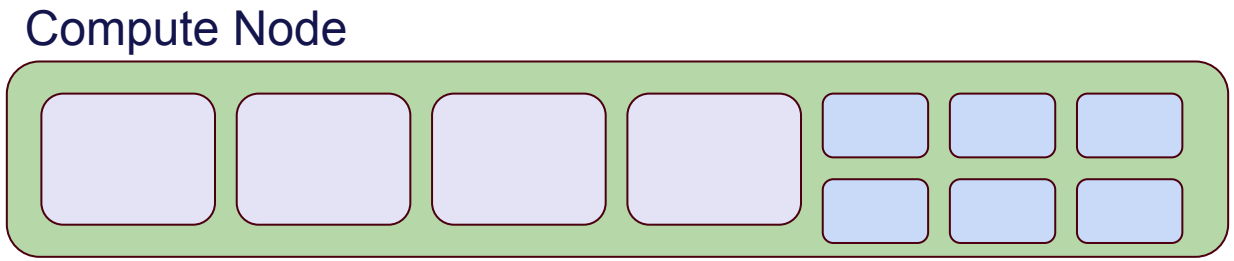
Distributed Software
A library (PyTorch) with software-level communication protocols for coordinating distributed tasks across nodes. (e.g. PyTorch with NCCL or MPI support)

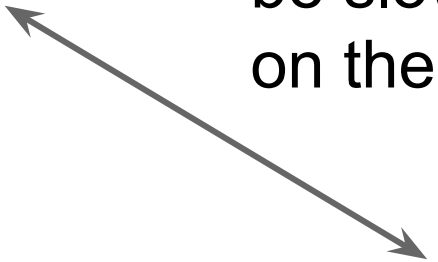
Parallel processing within a node

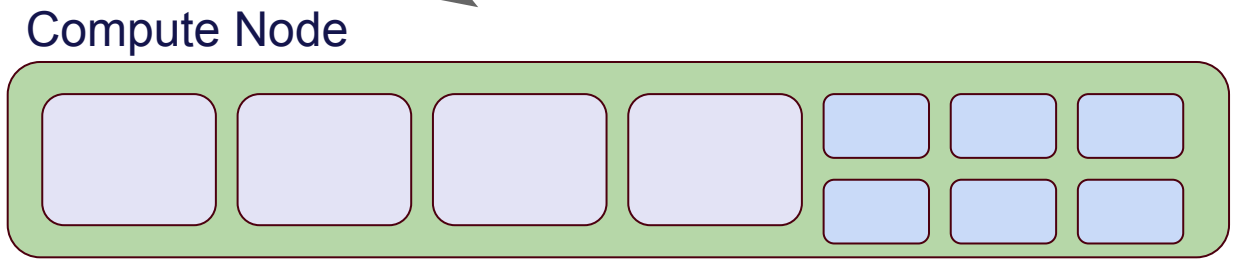


Single-node multi-GPU parallel processing

Multi-node Distributed Computing

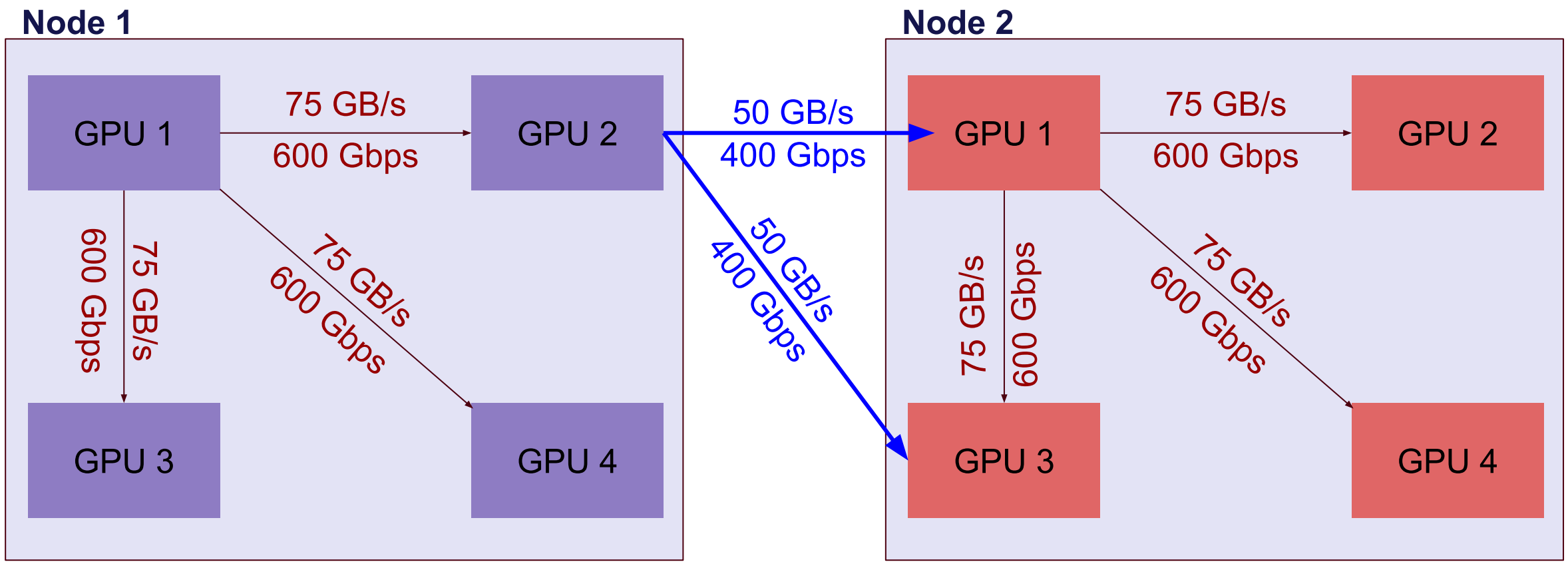






Multi-node distributed computing

Communications between GPUs on different nodes will be slower than within GPUs on the same node



Inside Node (**NVLINK**):
Each GPU talks to other three GPUs at 75 GB/s (single direction). This sums up to 900 GB/s all GPU-GPU bidirectional speed.

75 GB/s * 6 * 2 = 900 GB/s

Outside Node (**InfiniBand Network NDR**):
Each GPU communicates to other GPUs in another node at 400 Gbps (50 GB/s).

Why use distributed computing?

- Scalability
- Resource sharing
- Speed up

Bottlenecks

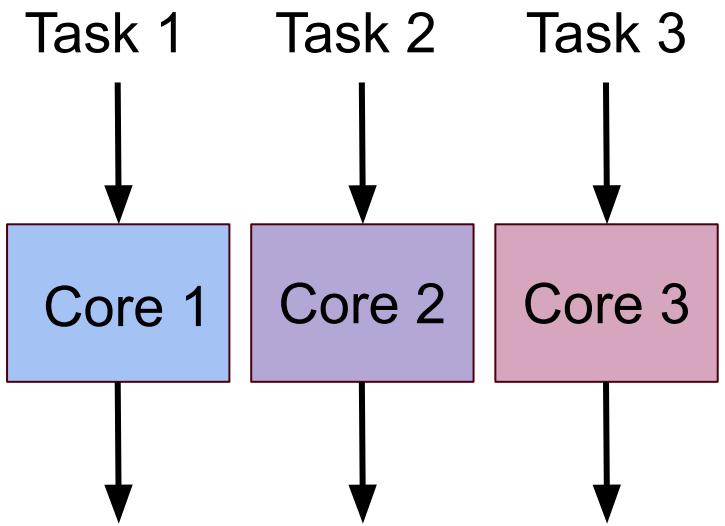
- Memory
  - Training 70B LLM
  - Large neural data processing
- Compute (CPU/GPU)
  - Large # of hyperparameter sweeps
  - Large tensor operations (t-SNE)
- Storage (I/O)
  - Web scraping
- Network & Communication
  - Web scraping

Agenda

1. The Basics
2. Embarrassingly Parallel Processing
3. Distributed Data Parallel (DDP) Processing
4. Beyond DDP

Embarrassingly Parallel Processing

- Easy to parallelize because the problem already consists of **independent tasks**
- **Low or no communication** required between nodes
- High scalability



Embarrassingly Parallel Processing

- Data Parallelism

  Tasks are divided based on the index of the data filename or a loop structure.

- Parameter Sweeps

  Tasks are executed based on different sets of hyperparameters inputs.

- Statistical Simulations

  Tasks are driven by different random number generator seeds or configuration.

SLURM Array Jobs

The easiest way to submit an embarrassingly parallel job on HPC

```
#! /bin/bash
#SBATCH --job-name=job-array
#SBATCH --partition=kempner_requeue
#SBATCH --account=kempner_dev
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=4GB
#SBATCH --time=15:00
#SBATCH --array=1-4
#SBATCH --output=%A_%a.out

module load python

python hyperparameter_tuning.py --task_id $SLURM_ARRAY_TASK_ID
```

A job will be submitted for each element in the array.

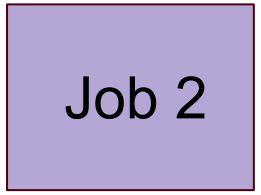
The only difference between jobs is:

SLURM_ARRAY_TASK_ID

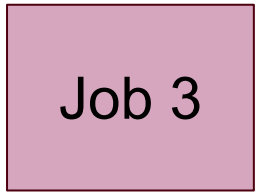
SLURM Array Jobs

```
#! /bin/bash
#SBATCH --job-name=job-array
#SBATCH --partition=kempner_requeue
#SBATCH --account=kempner_dev
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=4GB
#SBATCH --time=15:00
#SBATCH --array=1-4
#SBATCH --output=%A_%a.out

module load python

python hyperparameter_tuning.py --task_id $SLURM_ARRAY_TASK_ID
```

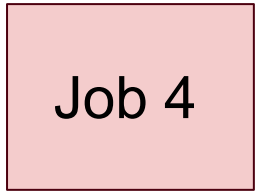

```
python my_code.py --task_id 1
```




```
python my_code.py --task_id 2
```



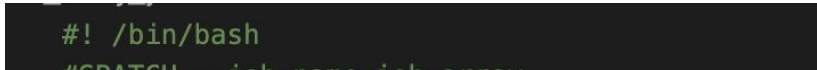

```
python my_code.py --task_id 3
```




```
python my_code.py --task_id 4
```

SLURM Array Jobs (with limiting active jobs)

```
#! /bin/bash
#SBATCH --job-name=job-array
#SBATCH --partition=kempner_requeue
#SBATCH --account=kempner_dev
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=4GB
#SBATCH --time=15:00
#SBATCH --array=1-12%2
#SBATCH --output=%A_%a.out

module load python

python hyperparameter_tuning.py --task_id $SLURM_ARRAY_TASK_ID
```



Job 1 `python my_code.py --task_id 1`

Job 2 `python my_code.py --task_id 2`

Only two jobs will run simultaneously.

```
[nkhoshnevis@holylogin01 array-job]$ squeue -u nkhoshnevis
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
 50590187_[3-12%2] kempner_r job-arra nkhoshne PD       0:00      1 (JobArrayTaskLimit)
       50590187_1 kempner_r job-arra nkhoshne  R       0:24      1 holygpu8a19606
       50590187_2 kempner_r job-arra nkhoshne  R       0:24      1 holygpu8a19606
```

Array Jobs

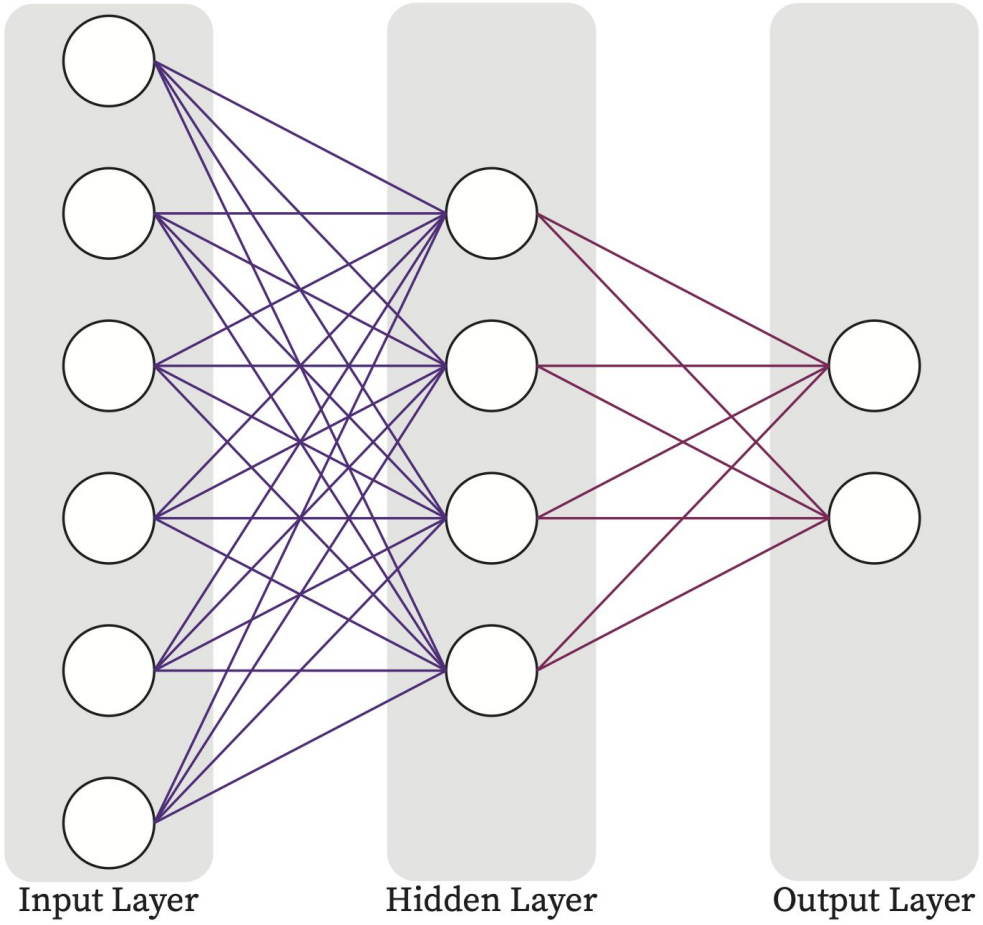
```
#! /bin/bash
#SBATCH --job-name=job-array
#SBATCH --account= kempner_undergrads
#SBATCH --output=%A_%a.out
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gpus-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=10:00
#SBATCH --mem=4GB
#SBATCH --partition=kempner_requeue
#SBATCH --array=1-4

module load python

python hyperparameter_tuning.py --task_id $SLURM_ARRAY_TASK_ID
```

How could we map hyperparameter options to task id?

Agenda

1. The Basics
2. Embarrassingly Parallel Processing
3. Distributed Data Parallel Processing
4. Beyond DDP

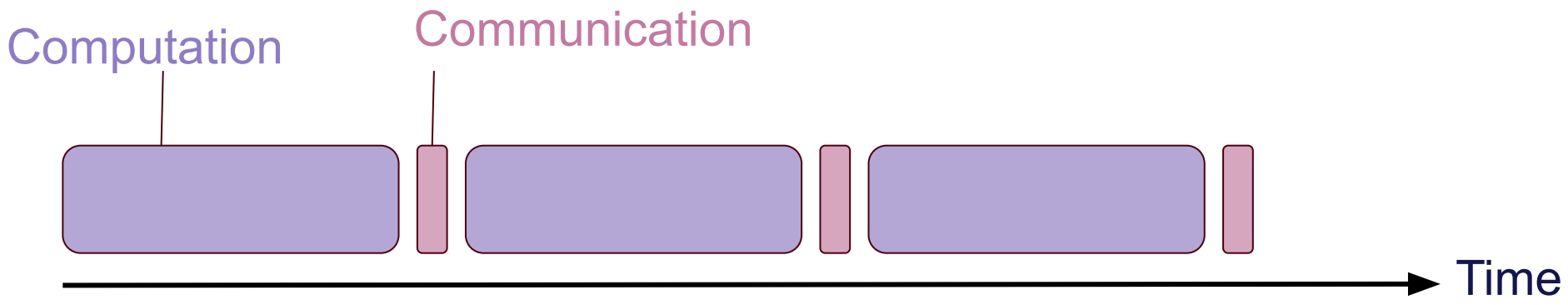
Multi-layer Perceptron



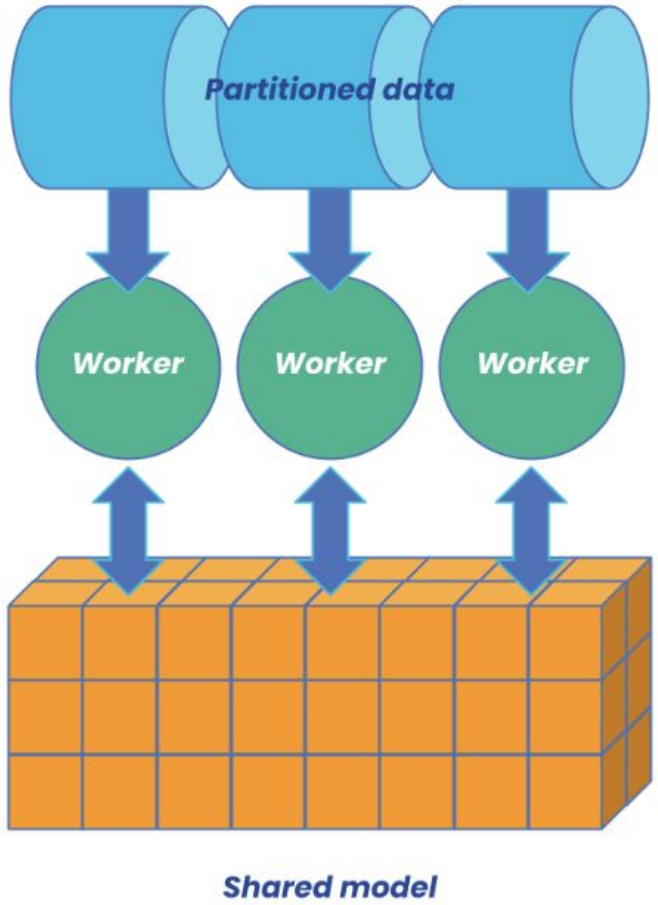

```python
class MLP(nn.Module):
  def __init__(self, in_feature, hidden_units, out_feature):
    super().__init__()

    self.hidden_layer = nn.Linear(in_feature, hidden_units)
    self.output_layer = nn.Linear(hidden_units, out_feature)

  def forward(self, x):
    x = self.hidden_layer(x)
    x = self.output_layer(x)
    return x
```

Coarse-grained parallelism

- Mostly independent tasks
- Occasional communication between processes
- High scalability



Distributed Data Parallel Processing



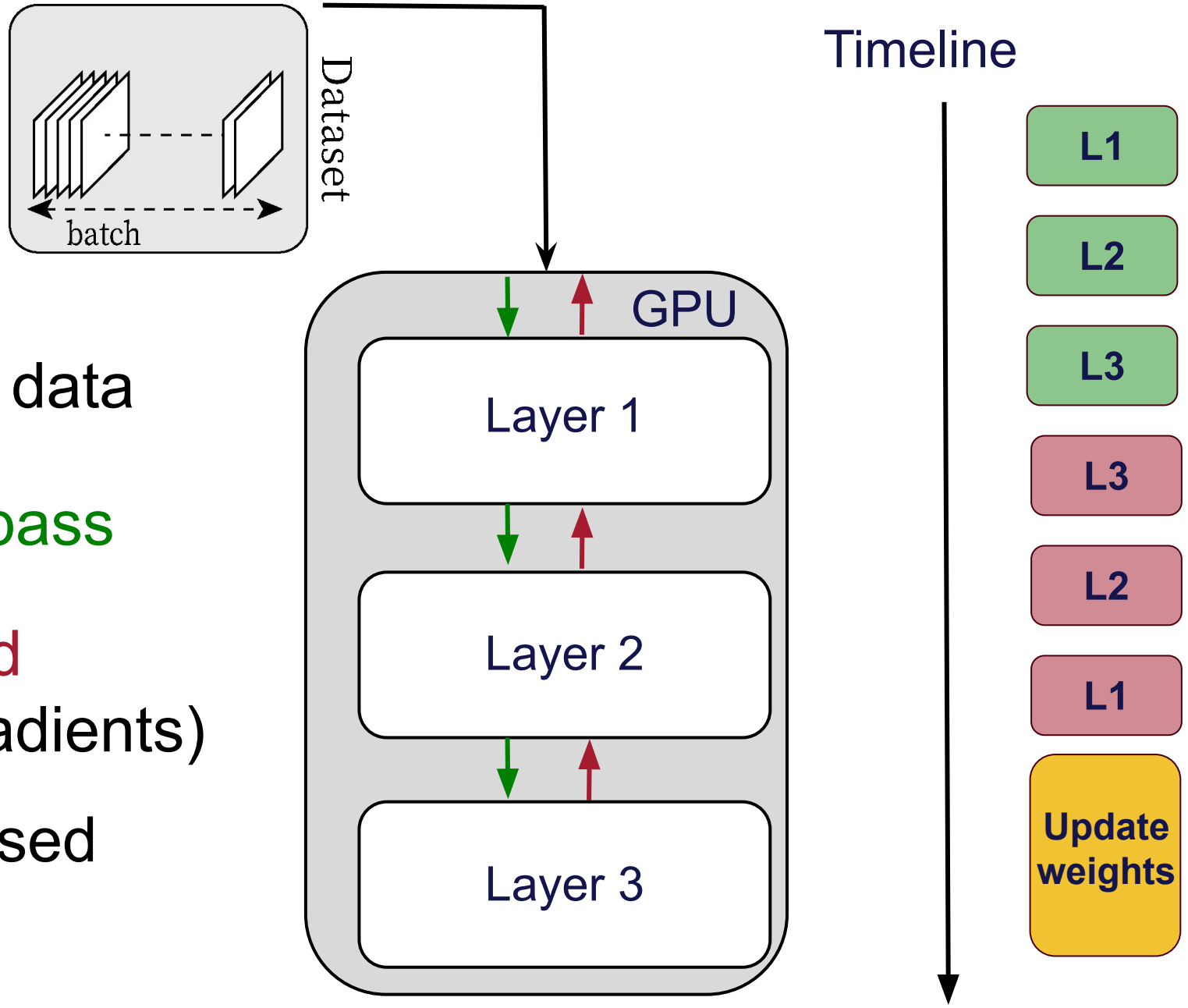
Most common approach to distributed training in machine learning

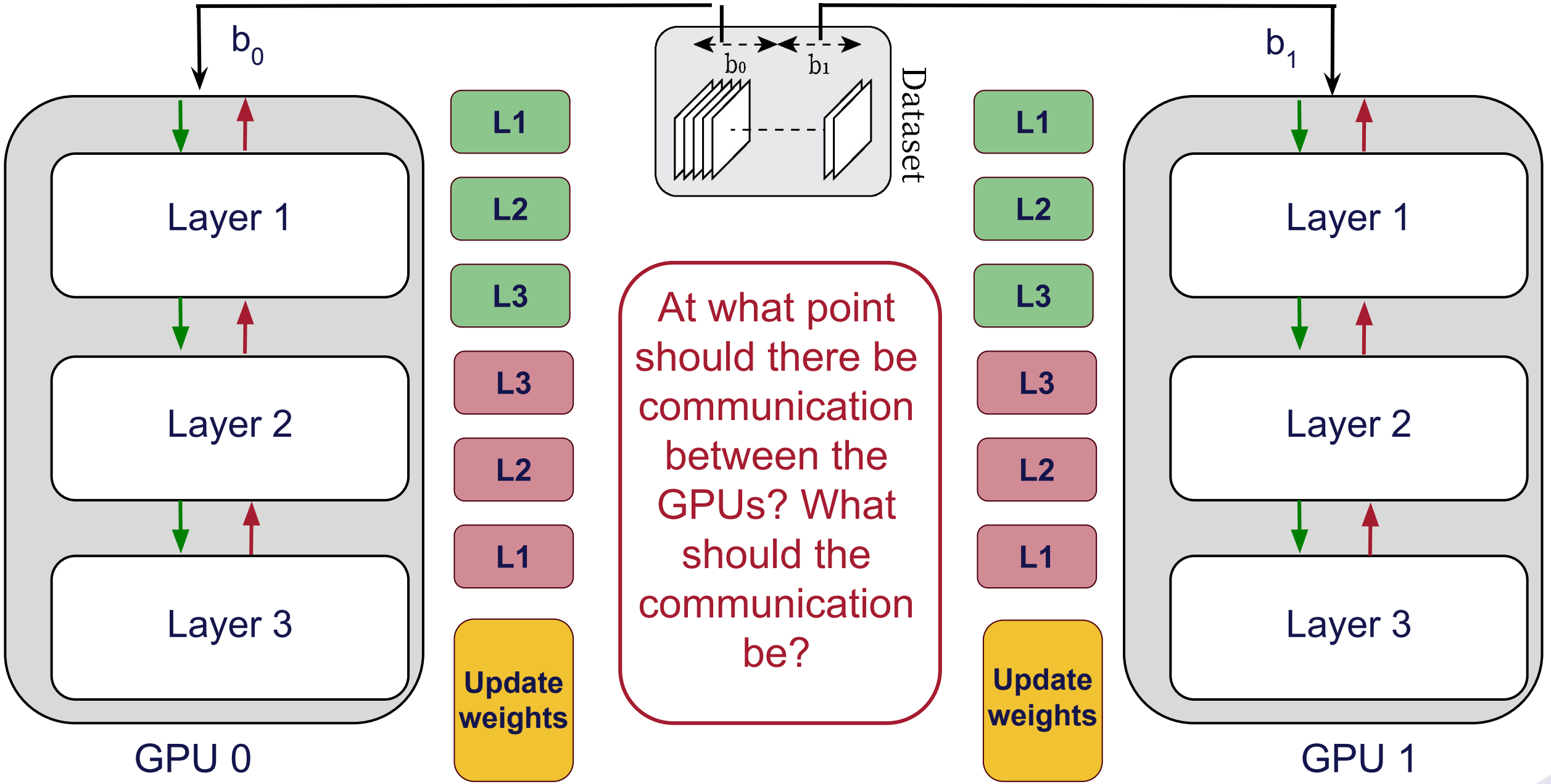
Each GPU trains a copy of the model. Dataset is split into different batches of data on each GPU

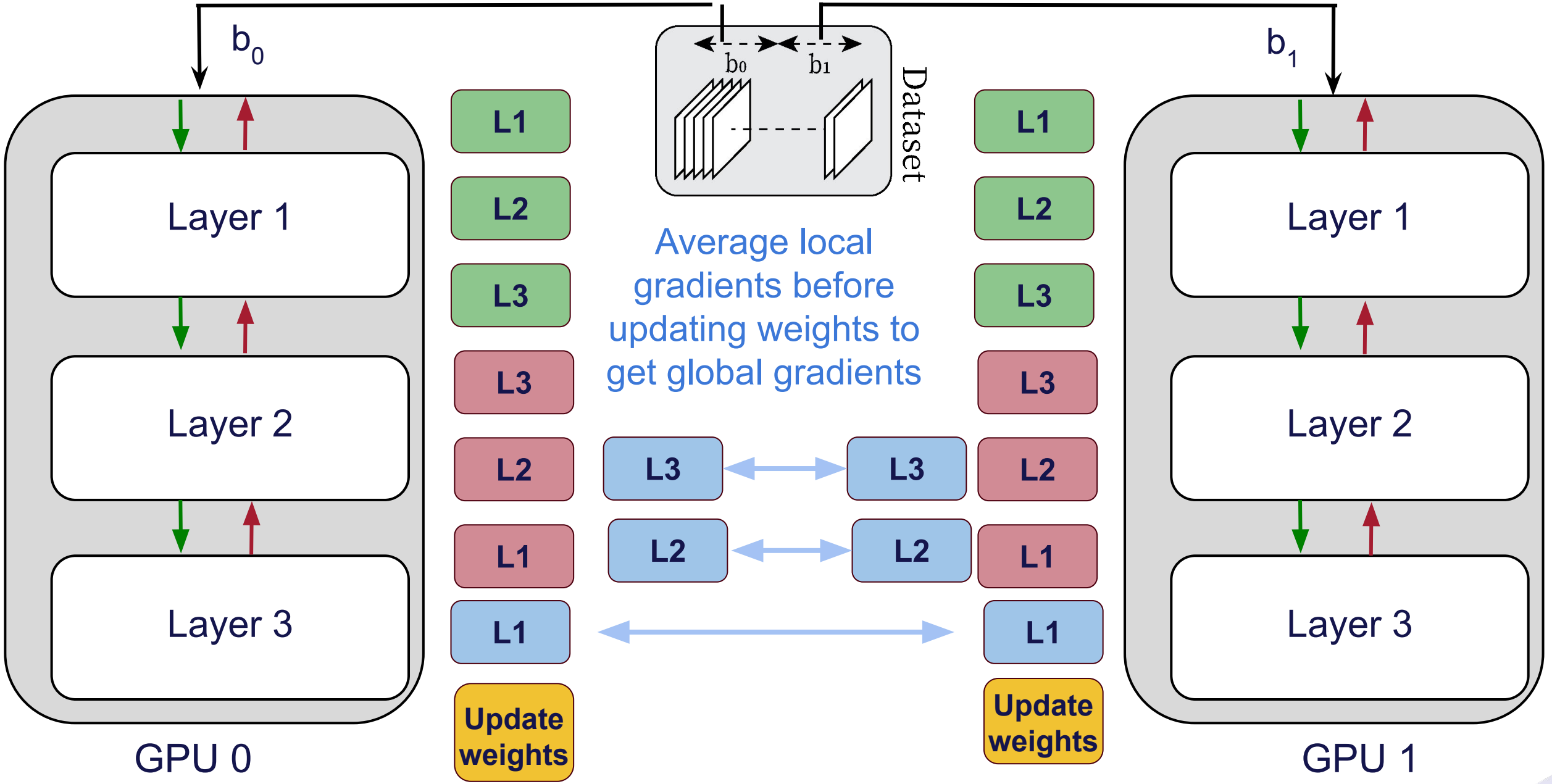
https://www.anyscale.com/blog/what-is-distributed-training

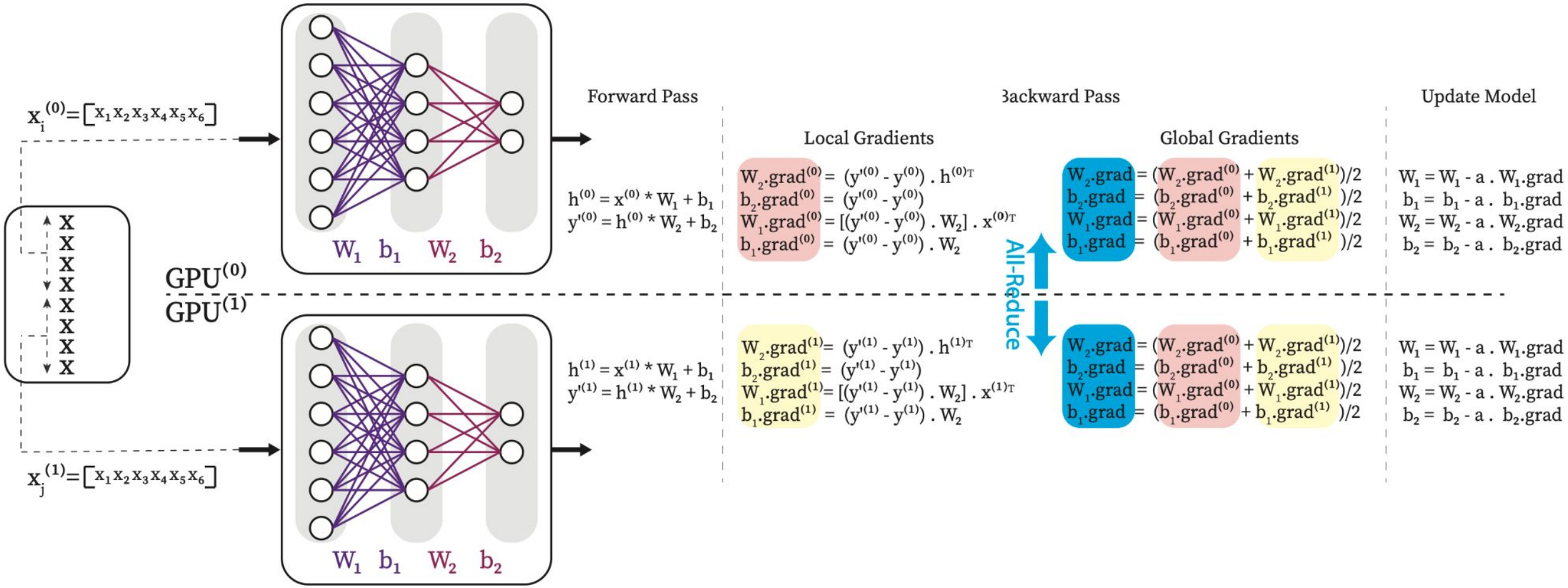
Single GPU MLP Training

1) Model gets batch of data
2) Computes forward pass
3) Computes backward pass (computing gradients)
4) Updates weights based on gradients




b0
b1
b0
b1
Dataset
Layer 1
Layer 2
Layer 3
L1
L2
L3
L3
L2
L1
Update weights
At what point should there be communication between the GPUs? What should the communication be?
L1
L2
L3
L3
L2
L1
Update weights
Layer 1
Layer 2
Layer 3
GPU 0
GPU 1

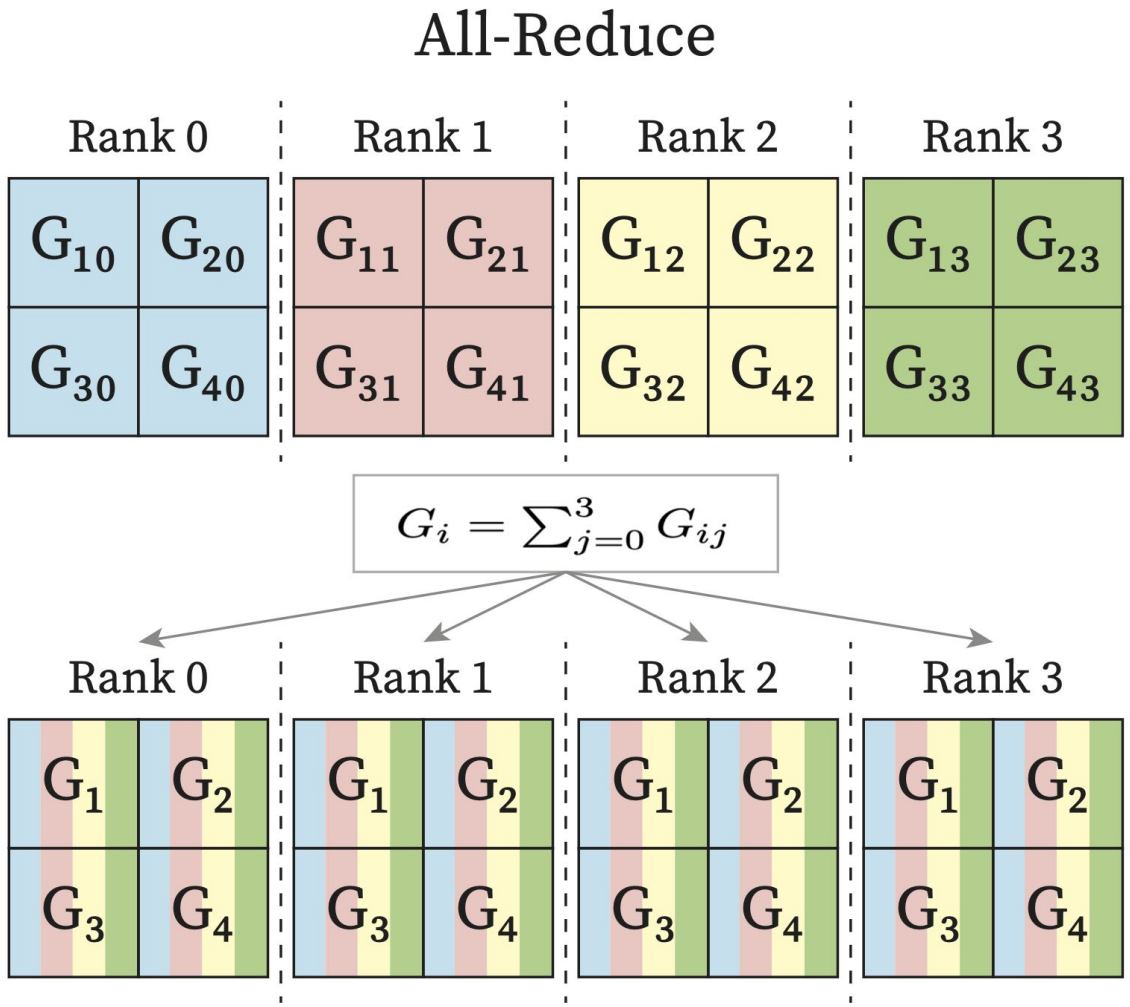

b0
b1
b0
b1
Dataset
Layer 1
Layer 2
Layer 3
L1
L2
L3
L3
L2
L1
L1
Update weights
Average local gradients before updating weights to get global gradients
L3
L3
L2
L2
L1
L1
L1
L2
L3
L3
L2
L1
Update weights
Layer 1
Layer 2
Layer 3
GPU 0
GPU 1

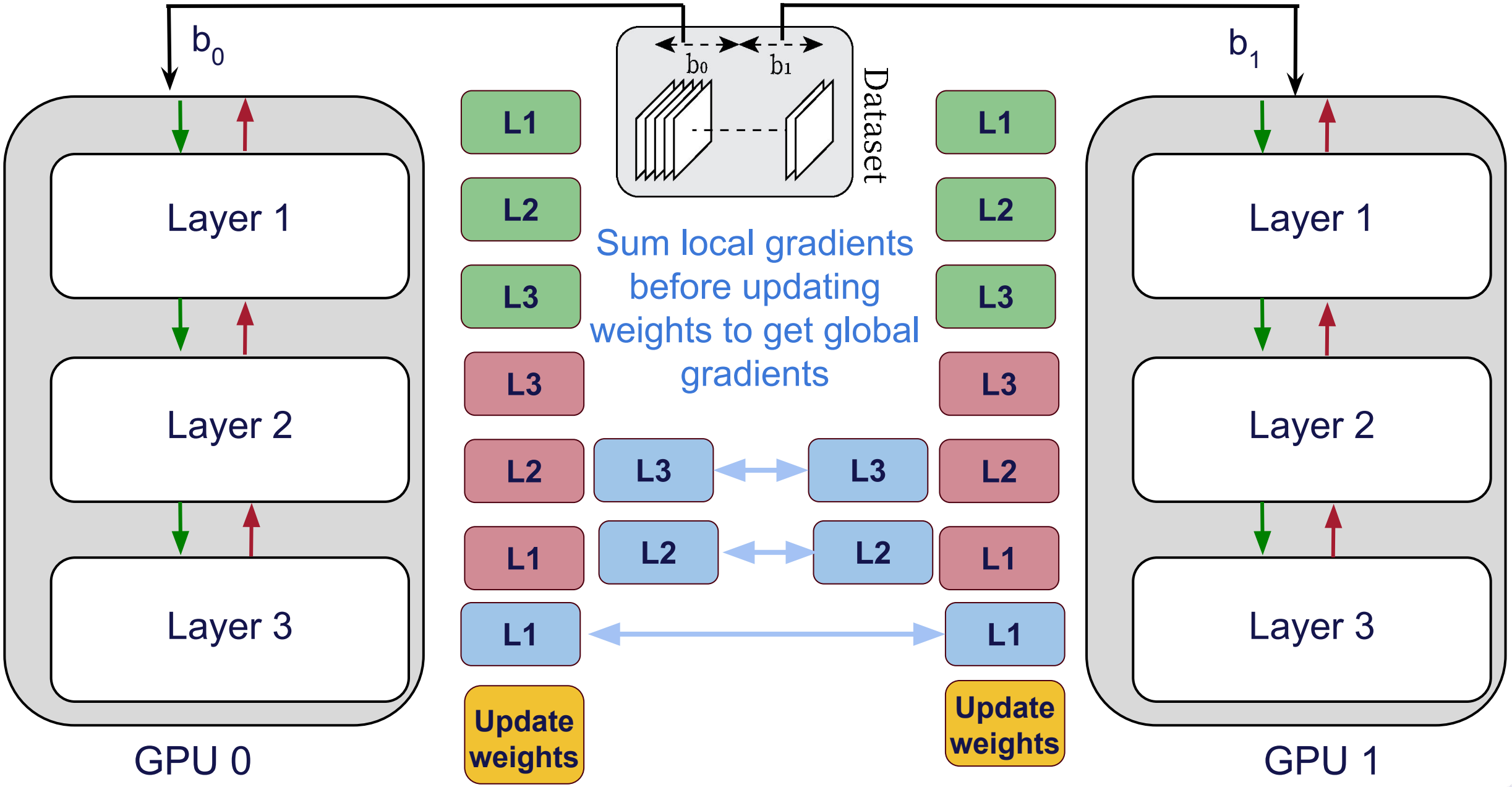

$x_i^{(0)} = [x_1 x_2 x_3 x_4 x_5 x_6]$
$x_j^{(1)} = [x_1 x_2 x_3 x_4 x_5 x_6]$
GPU(0)
GPU(1)
W_1 b_1 W_2 b_2
W_1 b_1 W_2 b_2
Forward Pass
$h^{(0)} = x^{(0)} * W_1 + b_1$
$y'^{(0)} = h^{(0)} * W_2 + b_2$
$h^{(1)} = x^{(1)} * W_1 + b_1$
$y'^{(1)} = h^{(1)} * W_2 + b_2$
Backward Pass
Local Gradients
$W_2.grad^{(0)} = (y'^{(0)} - y^{(0)}) . h^{(0)T}$
$b_2.grad^{(0)} = (y'^{(0)} - y^{(0)})$
$W_1.grad^{(0)} = [(y'^{(0)} - y^{(0)}) . W_2] . x^{(0)T}$
$b_1.grad^{(0)} = (y'^{(0)} - y^{(0)}) . W_2$
$W_2.grad^{(1)} = (y'^{(1)} - y^{(1)}) . h^{(1)T}$
$b_2.grad^{(1)} = (y'^{(1)} - y^{(1)})$
$W_1.grad^{(1)} = [(y'^{(1)} - y^{(1)}) . W_2] . x^{(1)T}$
$b_1.grad^{(1)} = (y'^{(1)} - y^{(1)}) . W_2$
All-Reduce
Global Gradients
$W_2.grad = (W_2.grad^{(0)} + W_2.grad^{(1)})/2$
$b_2.grad = (b_2.grad^{(0)} + b_2.grad^{(1)})/2$
$W_1.grad = (W_1.grad^{(0)} + W_1.grad^{(1)})/2$
$b_1.grad = (b_1.grad^{(0)} + b_1.grad^{(1)})/2$
$W_2.grad = (W_2.grad^{(0)} + W_2.grad^{(1)})/2$
$b_2.grad = (b_2.grad^{(0)} + b_2.grad^{(1)})/2$
$W_1.grad = (W_1.grad^{(0)} + W_1.grad^{(1)})/2$
$b_1.grad = (b_1.grad^{(0)} + b_1.grad^{(1)})/2$
Update Model
$W_1 = W_1 - a . W_1.grad$
$b_1 = b_1 - a . b_1.grad$
$W_2 = W_2 - a . W_2.grad$
$b_2 = b_2 - a . b_2.grad$
$W_1 = W_1 - a . W_1.grad$
$b_1 = b_1 - a . b_1.grad$
$W_2 = W_2 - a . W_2.grad$
$b_2 = b_2 - a . b_2.grad$

How does that communication actually happen?

NVIDIA Collective Communication Library (NCCL, pronounced “NICKEL”) is used as backend in distributed strategies for NVIDIA GPUs

Has various NCCL collective communication primitives




b0
b1
Dataset
b0
b1
Layer 1
Layer 2
Layer 3
GPU 0
GPU 1
L1
L2
L3
L3
L2
L3
L3
L2
L1
L1
Sum local gradients before updating weights to get global gradients
L1
L2
L3
L3
L2
L1
L1
Update weights
Update weights

So how do we get this working in Pytorch and on the cluster?

1) Add some logistics of set-up

2) Wrap model in DDP
   from torch.nn.parallel import DistributedDataParallel as DDP
   model = DDP(model, device_ids=[device])

3) Use DistributedSampler
   from torch.utils.data.distributed import DistributedSampler
   Use as argument in dataloader

Exercise: Try out DDP yourself

1) Look at DDP example (2.2) here: https://github.com/KempnerInstitute/examples/tree/main/distributed-mlp

2) Increase the number of epochs to 3 and print a parameter on every epoch: print(model.module.hidden_layer.bias).

3) Run & look at outputs. Do they make sense?

4) Edit the scripts to run on 4 GPUs total, 2 each on 2 nodes. Run and look at outputs. Do the ranks and device ids make sense?

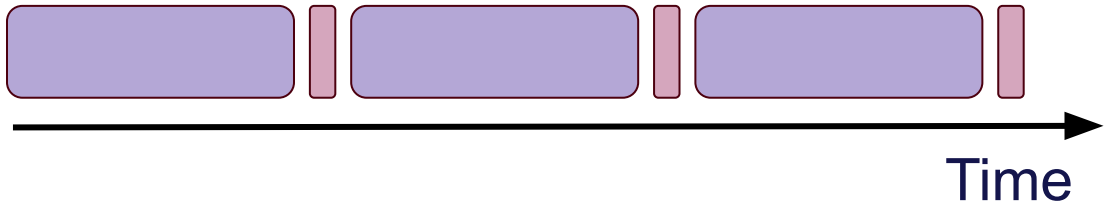
Agenda

1. The Basics
2. Embarrassingly Parallel Processing
3. Distributed Data Parallel Processing
4. Beyond DDP

Coarse vs fine-grained parallelism

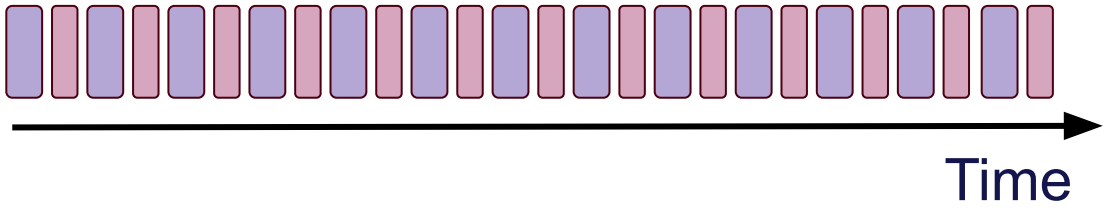
Coarse-grained

- Mostly independent tasks
- Occasional communication between processes
- High scalability



Fine-grained

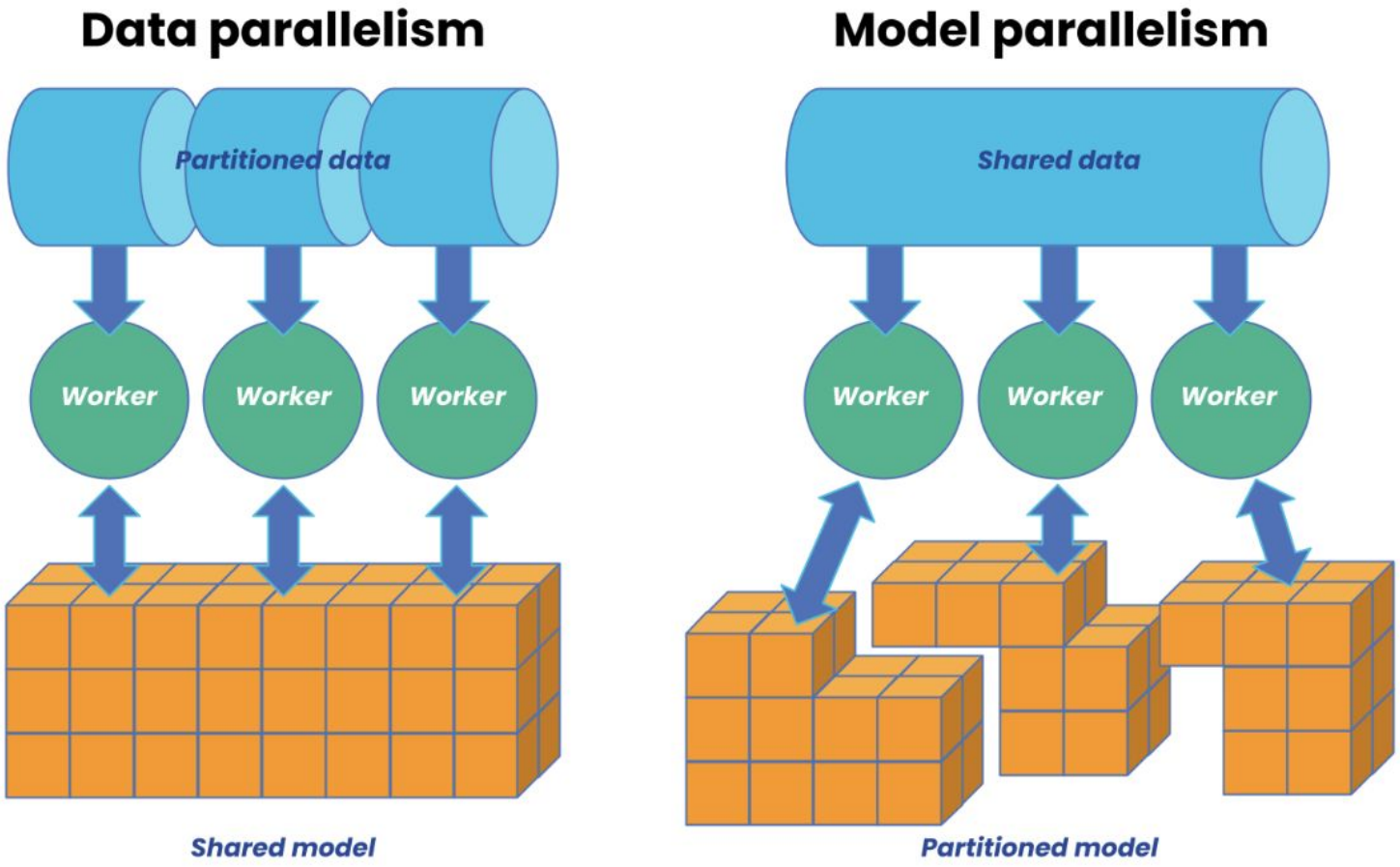
- Highly dependent tasks
- Frequent communication between processes
- Scalability limited by communication overhead



Computation

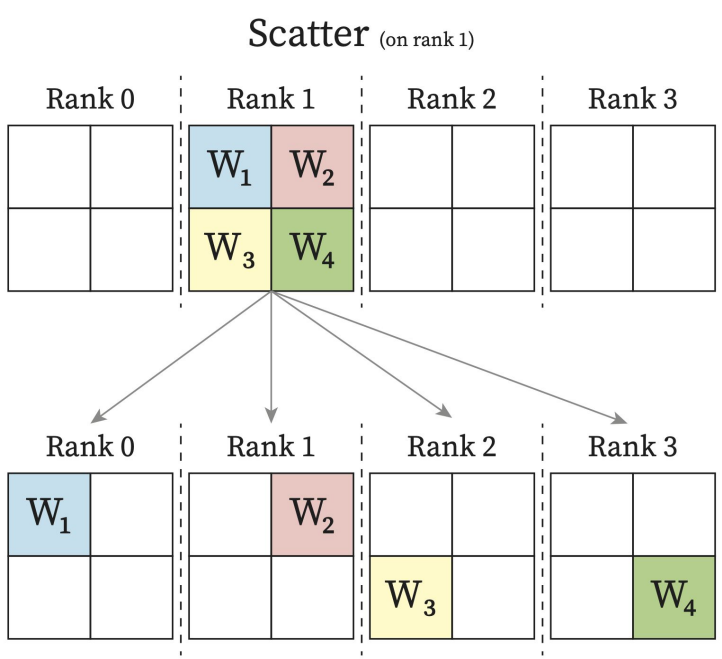
Communication

Model Parallelism

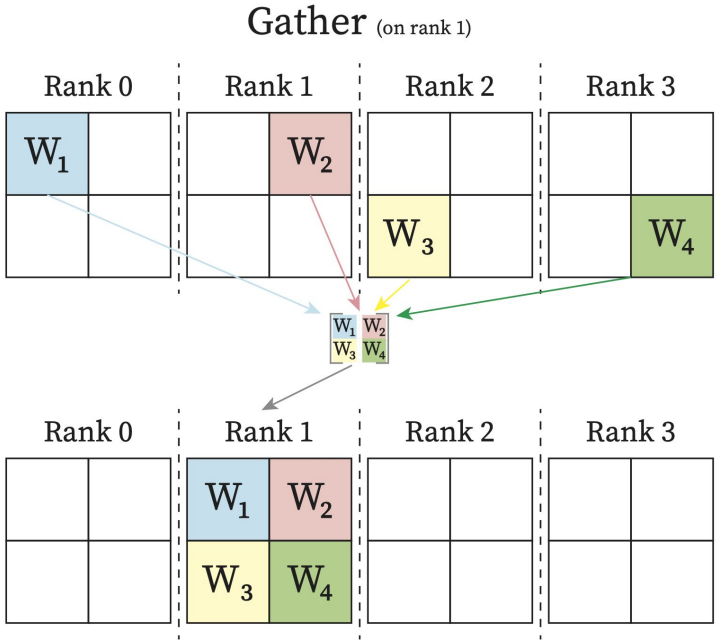


https://www.anyscale.com/blog/what-is-distributed-training

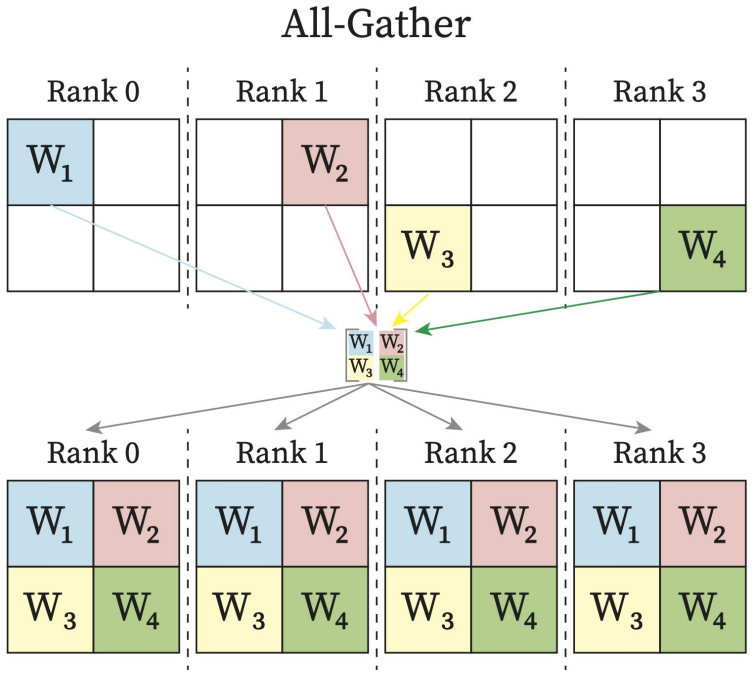
Other NCCL Collective Primitives



Scatter: From one rank data will be distributed across ranks.

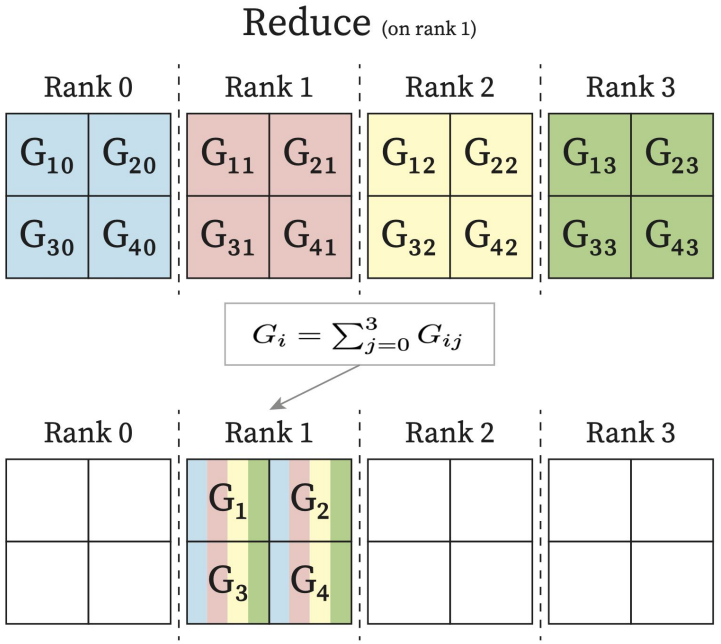


Gather: One rank will receive the aggregation of data from all ranks.

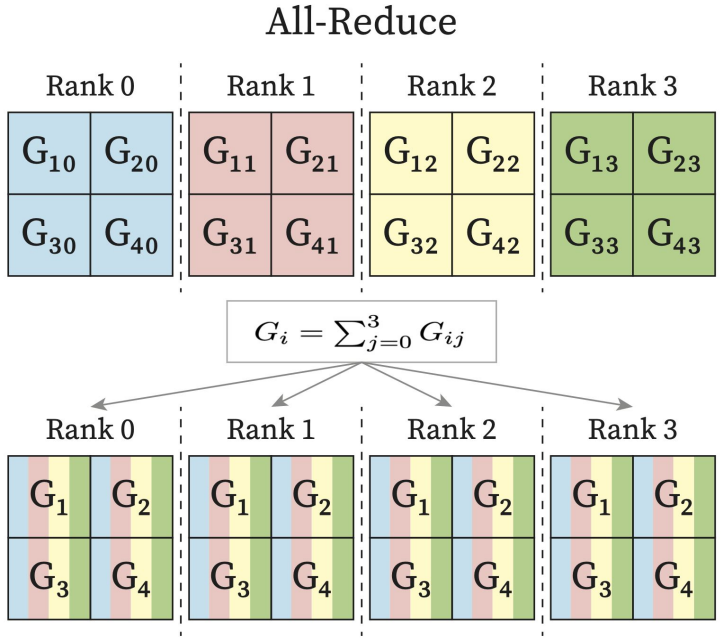


All-Gather: Each rank receives the aggregation of data from all ranks in the order of the ranks.

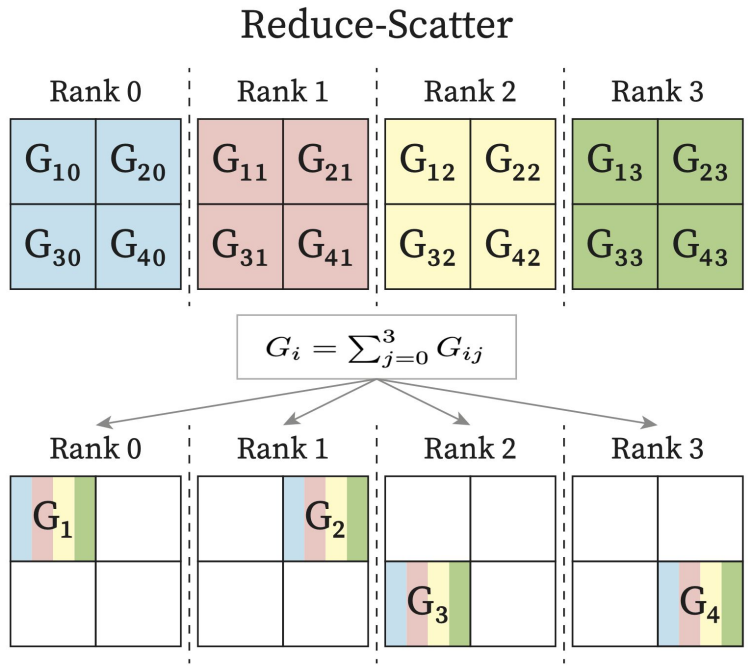
Other NCCL Collective Primitives



Reduce: One rank receives the reduction of input values across ranks.



All-Reduce: Each rank receives the reduction of input values across ranks.



Reduce-Scatter: Input values are reduced across ranks, with each rank receiving a subpart of the result.

Thank you