PyTorch

Distributed Data Parallel Training

Jeong-Sik Lee
ICVS Lab, Yeungnam University

Reference

Pytorch Official Tutorials

Overview of DataParallel & DistributedDataParallel(DDP)

Pytorch Official Example

Basic code for DDP

CS 329S Lecture Slide

Concept of DDP and AllReduce

Yolo v5 Code

Practical DDP code

Overview

This is the overview page for the torch.distributed package. As there are more and more documents, examples and tutorials added at different locations, it becomes unclear which document or tutorial to consult for a specific problem or what is the best order to read these contents. The goal of this page is to address this problem by categorizing documents into different topics and briefly describe each of them. If this is your first time building distributed training applications using PyTorch, it is recommended to use this document to navigate to the technology that can best serve your use case.

torch.distributed

- Distributed Data-Parallel Training
- RPC-Based Distributed Training
- Collective Communication

Data Parallel Training

DataParallel (DP)
Use single-mahcine multi-GPU, with the minimum code change.

DistributedDataParalle (DDP)
Use single-mahcine multi-GPU, further speed up training and are willing to write a little more code to set it up.

DistributedDataParalle and **launching script**
Use multi-mahcine, scale across machine boundaries.

General Distributed Training

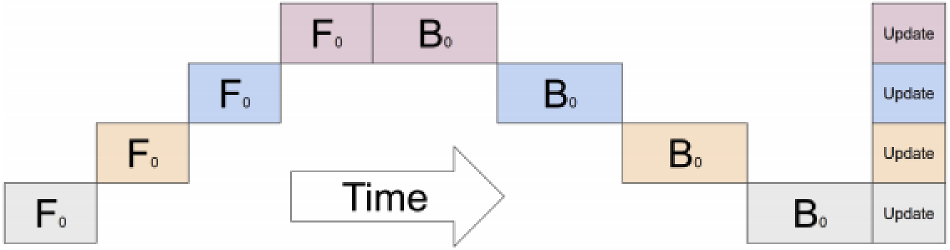
Many training paradigms do not fit into data parallelism, e.g., parameter server paradigm, distributed pipeline parallelism, reinforcement learning applications with multiple observers or agents, etc. The torch.distributed.rpc aims at supporting general distributed training scenarios.

RPC
Supports running a given function on a remote worker.

RRef
Helps to manage the lifetime of a remote object.

Distributed Autograd
Extends the autograd engine beyond machine boundaries.

Distributed Optimizer
Automatically reaches out to all participating workers to update parameters.



The figure represents a model with 4 layers placed on 4 different GPUs (vertical axis). The horizontal axis represents training this model through time demonstrating that only 1 GPU is utilized at a time (image source).

DataParallel

The DataParallel package enables single-machine multi-GPU parallelism with the lowest coding hurdle. It only requires a one-line change to the application code. The tutorial Optional: Data Parallelism shows an example. The caveat is that, although DataParallel is very easy to use, it usually does not offer the best performance. This is because the implementation of DataParallel replicates the model in every forward pass, and its single-process multi-thread parallelism naturally suffers from GIL contentions.

CLASS torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)

Implements data parallelism at the module level.

This container parallelizes the application of the given module by splitting the input across the specified devices by chunking in the batch dimension (other objects will be copied once per device). In the forward pass, the module is replicated on each device, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are summed into the original module.

The batch size should be larger than the number of GPUs used.

DataParallel

CLASS torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)

Parameters

- module (Module) – module to be parallelized
- device_ids (list of python:int or torch.device) – CUDA devices (default: all devices)
- output_device (int or torch.device) – device location of output (default: device_ids[0])

Example:

```
device_ids = [0, 1, 2]
model = model.to(torch.device(f“cuda{device_ids[0]}”)
net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
output = net(input_var)
```

DistributedDataParallel

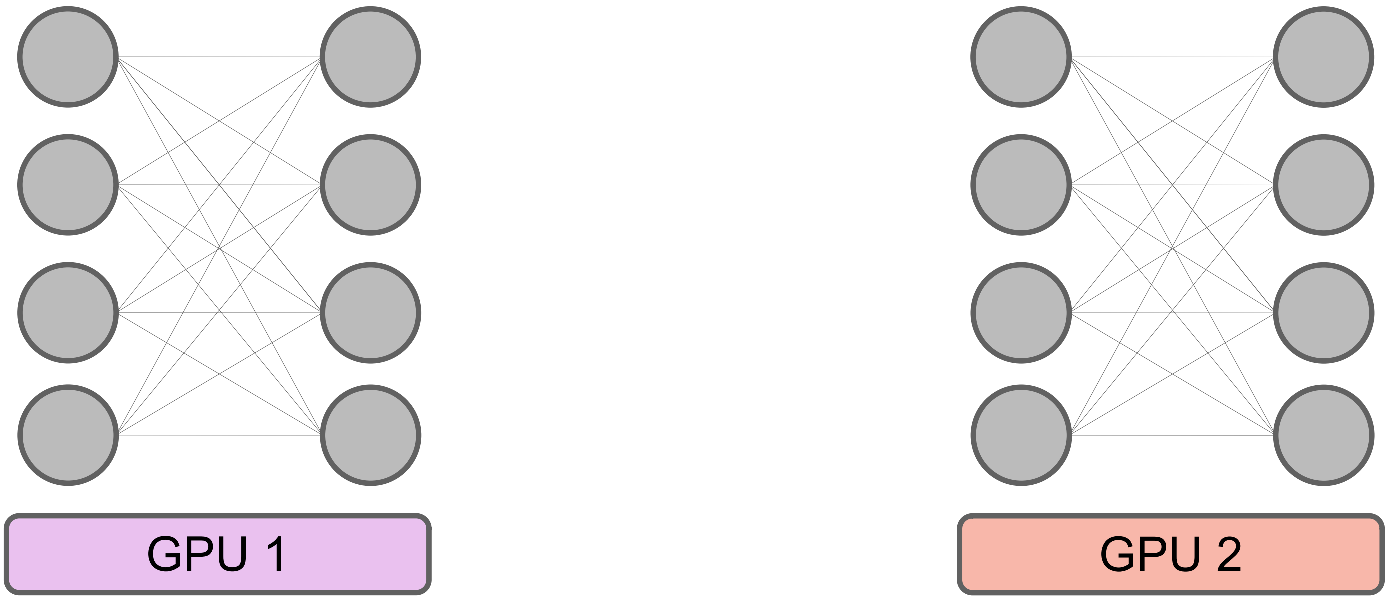
Compared to DataParallel, DistributedDataParallel requires one more step to set up, i.e., calling init_process_group. DDP uses multi-process parallelism, and hence there is no GIL contention across model replicas. Moreover, the model is broadcast at DDP construction time instead of in every forward pass, which also helps to speed up training. DDP is shipped with several performance optimization technologies. For a more in-depth explanation, please refer to this DDP paper (VLDB'20).

Split the data across devices

- each device sees a fraction of the batch
- each device replicates the model
- each device replicate the optimizer

DistributedDataParallel

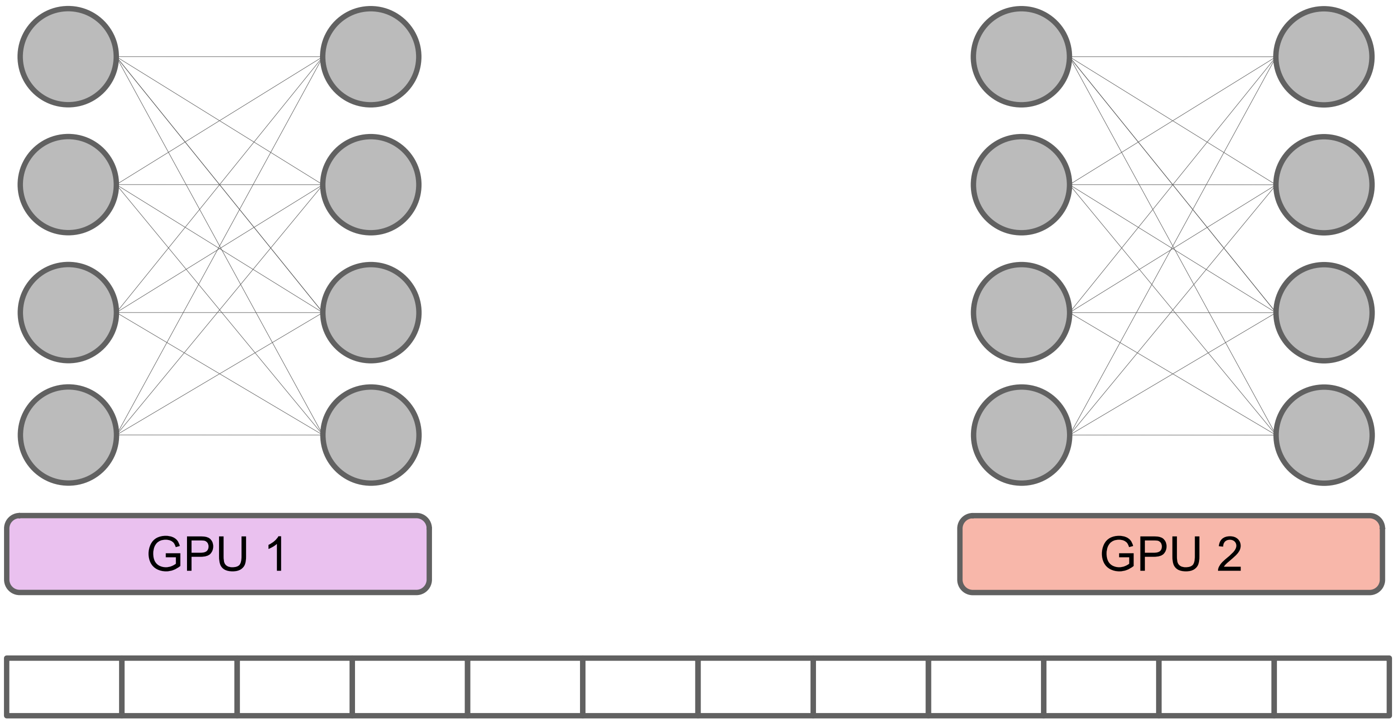
Replicate model across devices



Credit: CS329S

DistributedDataParallel

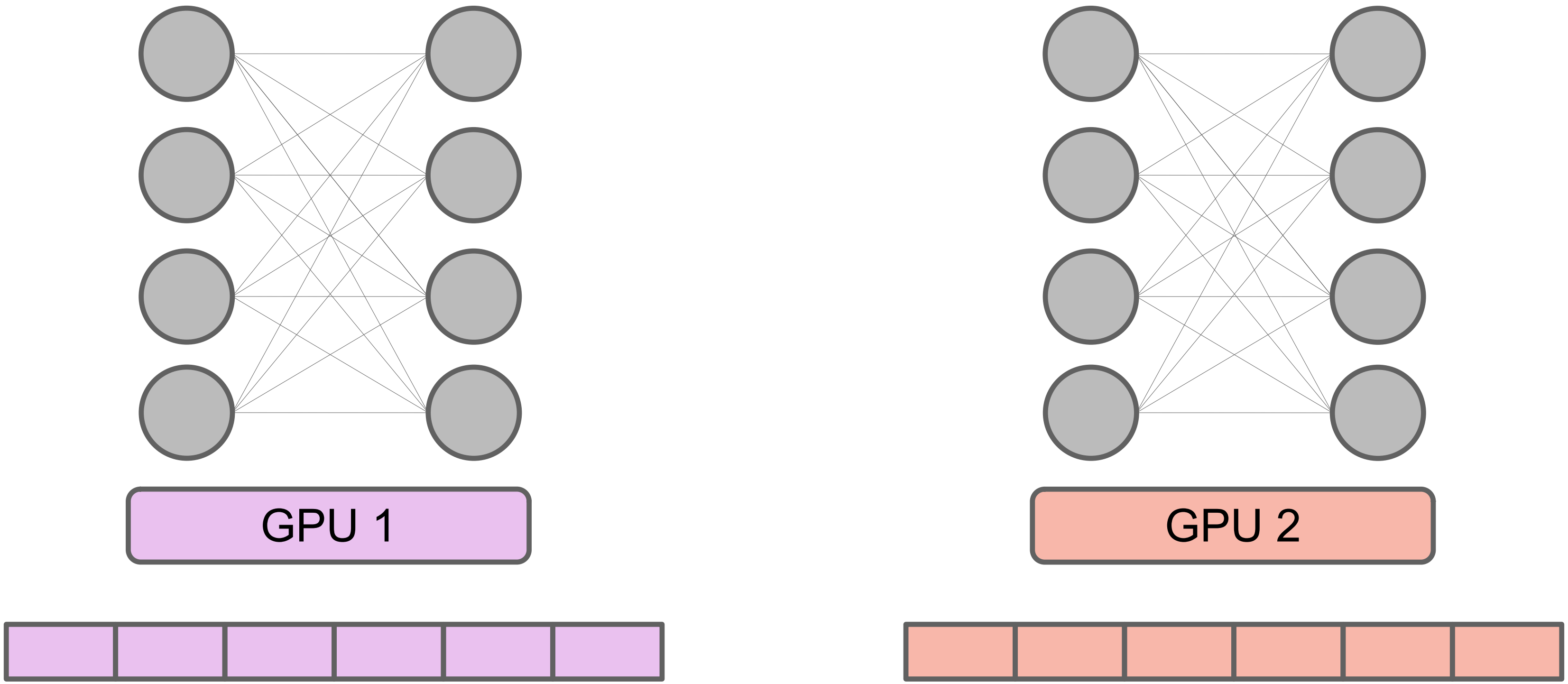
To push in a batch of data



Credit: CS329S

DistributedDataParallel

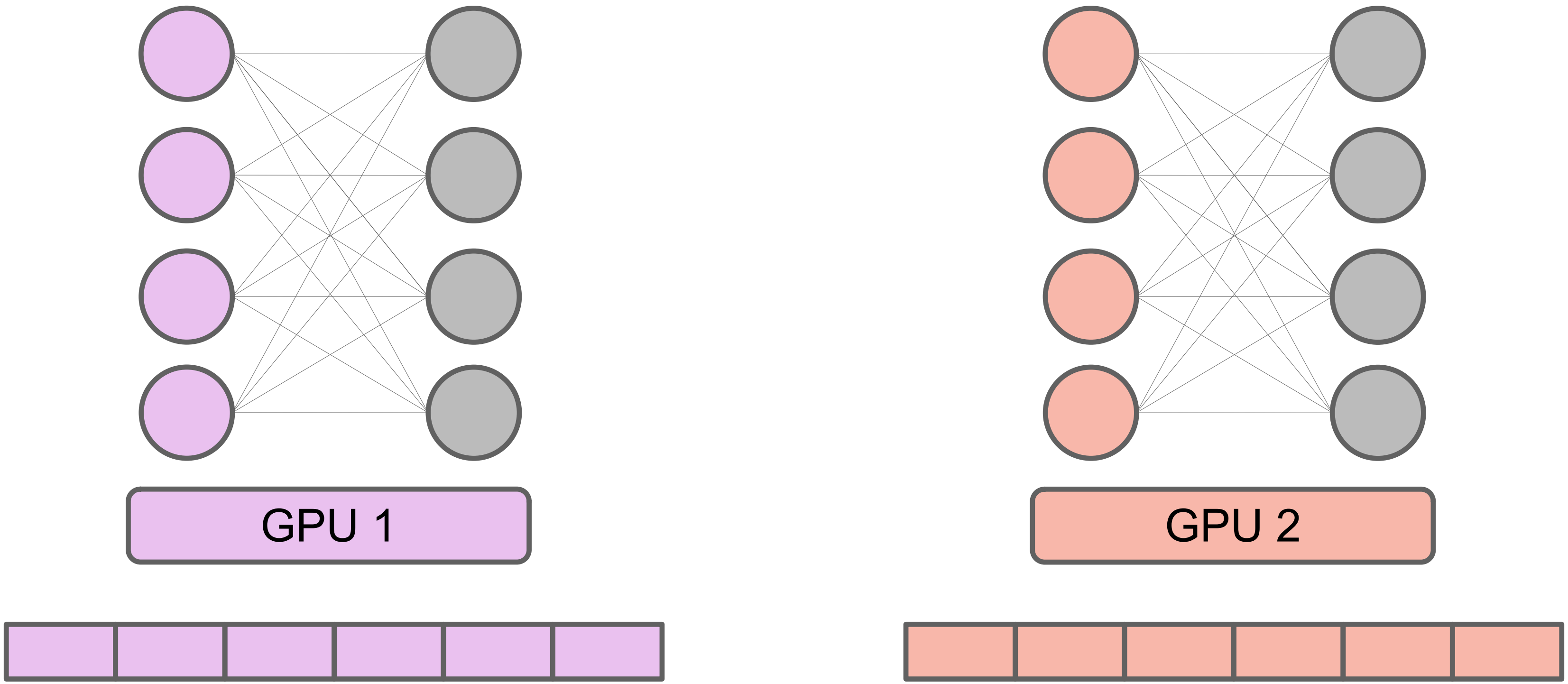
Split batch across device



Credit: CS329S

DistributedDataParallel

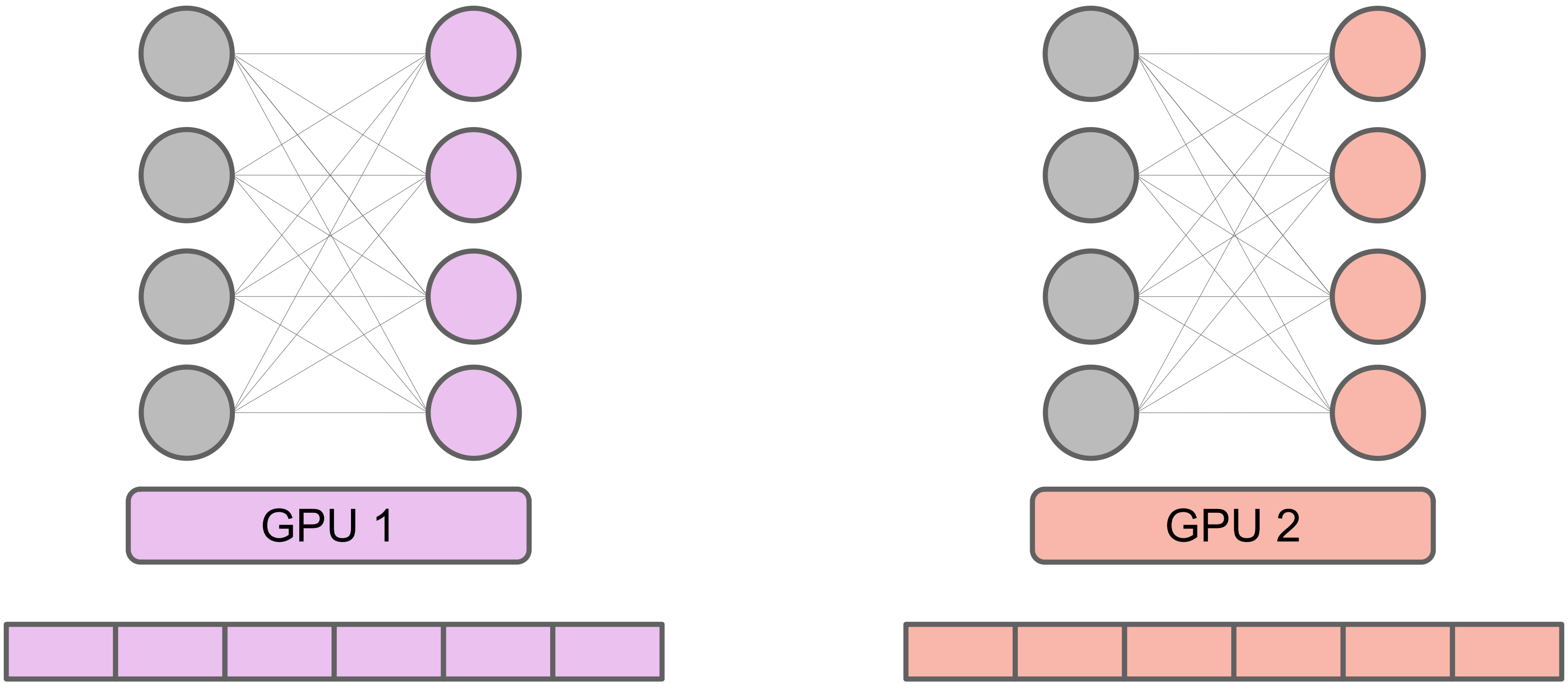
Parallel forward passes



Credit: CS329S

DistributedDataParallel

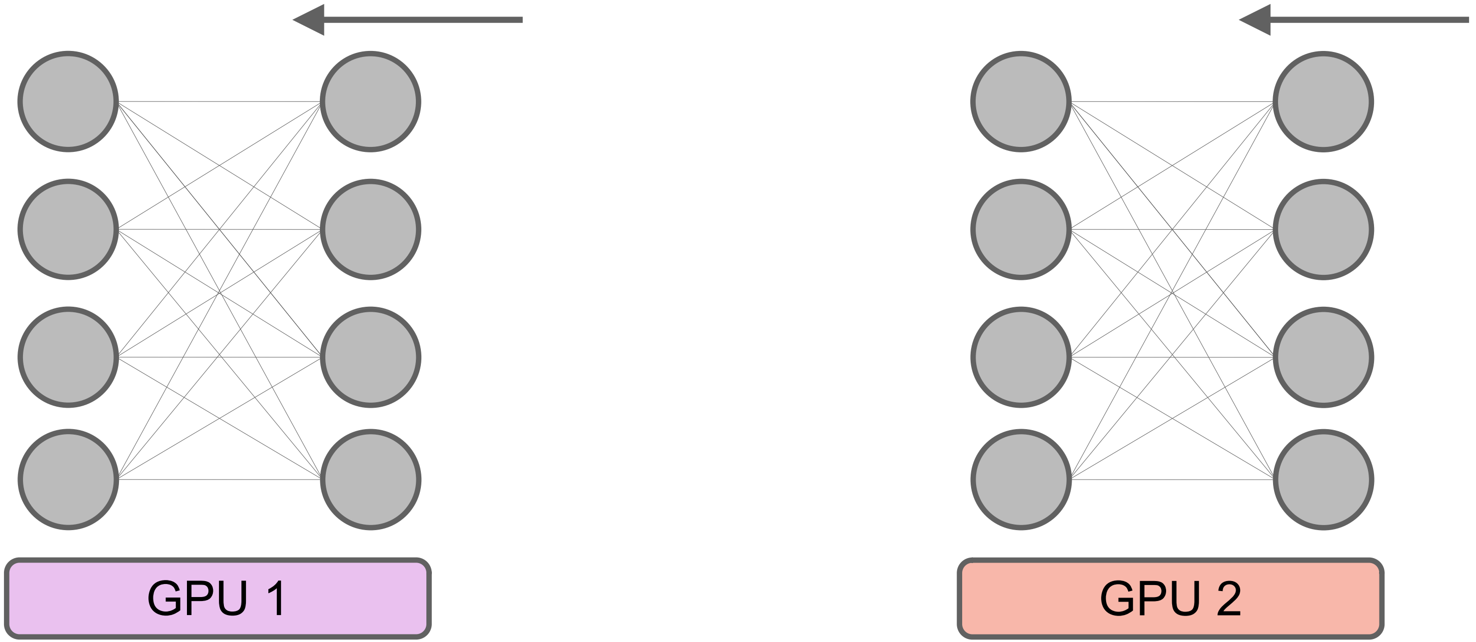
Split batch across device



Credit: CS329S

DistributedDataParallel

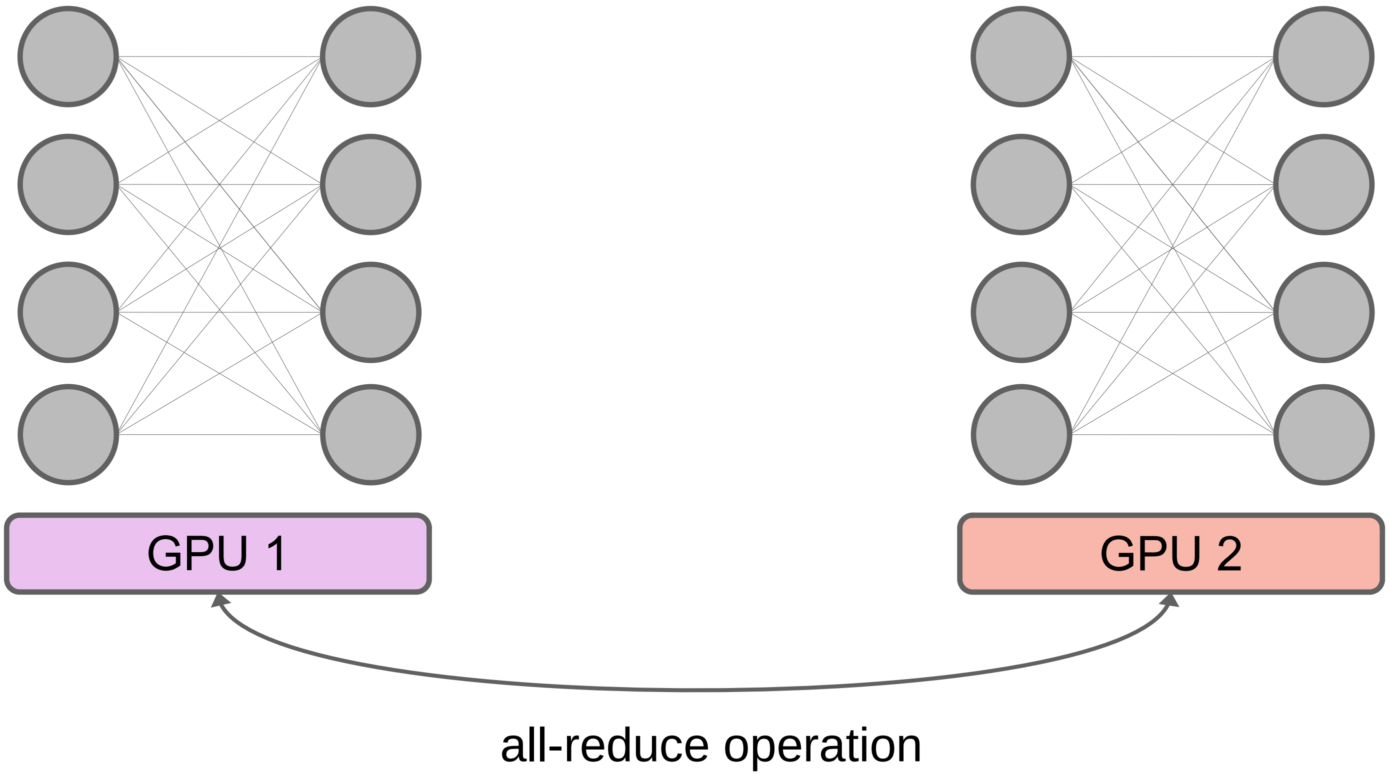
Backpropagate gradients



Credit: CS329S

DistributedDataParallel

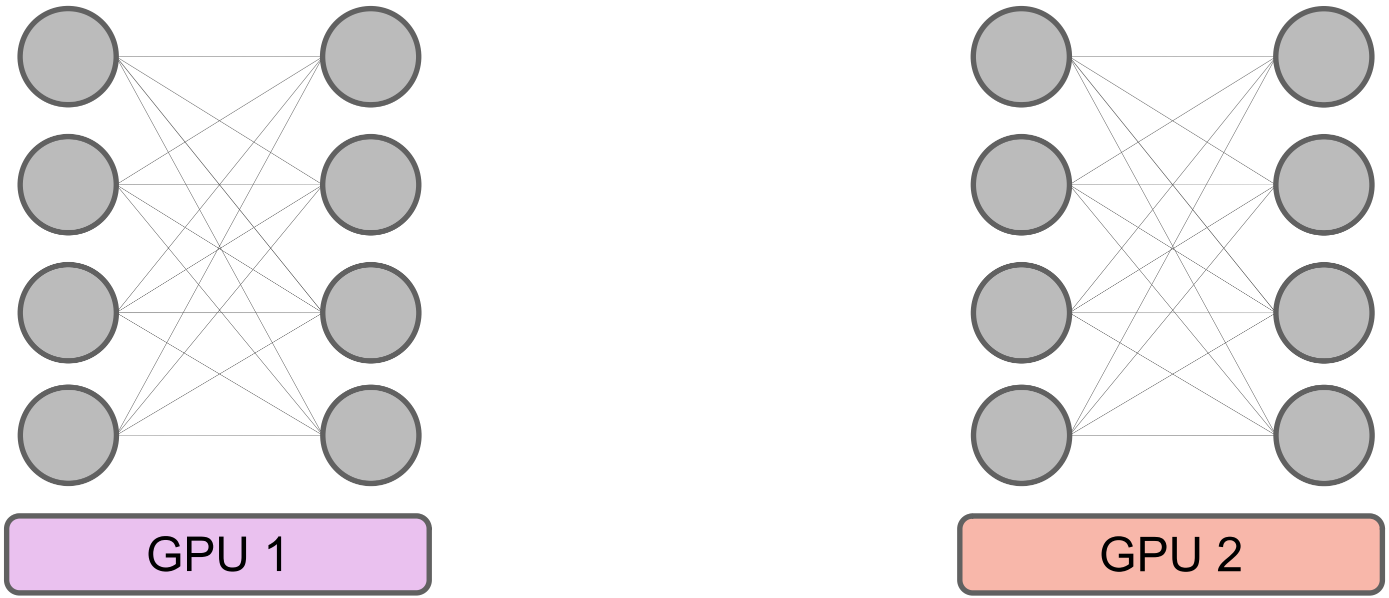
Do all-reduce operation



Credit: CS329S

DistributedDataParallel

All devices do the same gradient updates, all parameters stay synchronized!



Credit: CS329S

DistributedDataParallel: Collective Communication

Collective communication is communication that involves a group of processing elements (termed nodes in this entry) and effects a data transfer between all or some of these processing elements. Data transfer may include the application of a reduction operator or other transformation of the data. Collective communication functionality is often exposed through library interfaces or language constructs. Collective communication is a natural extension of the message-passing paradigm.

- Encyclopedia of Parallel Computing, Springer

Message Passing Interface (MPI)
Sets standard + CPU-CPU communication

Nvidia Collective Communications Library (nccl)
Follows MPI standard for GPU-GPU communication

Facebook Gloo
Optimized for ML: CPU-CPU / GPU-GPU communication

Credit: CS329S

DistributedDataParallel: All-Reduce

AllReduce is the primitive communication API used by DistributedDataParallel to compute gradient summation across all processes. It is supported by multiple communication libraries, including NCCL [2], Gloo [1], and MPI [4]. The AllReduce operation expects each participating process to provide an equally-sized tensor, collectively applies a given arithmetic operation (e.g., sum, prod, min, max) to input tensors from all processes, and returns the same result tensor to each participant. A naive implementation could simply let every process broadcast its input tensor to all peers and then apply the arithmetic operation independently. However, as AllReduce has significant impact on distributed training speed, communication libraries have implemented more sophisticated and more efficient algorithms, such as ring-based AllReduce [2] and tree-based AllReduce [23]. As one AllReduce operation cannot start until all processes join, it is considered to be a synchronized communication, as opposed to the P2P communication used in parameter servers [27].

PyTorch Distributed: Experiences on AcceleratingData Parallel Training

DistributedDataParallel: All-Reduce

All-reduce operation

p processes

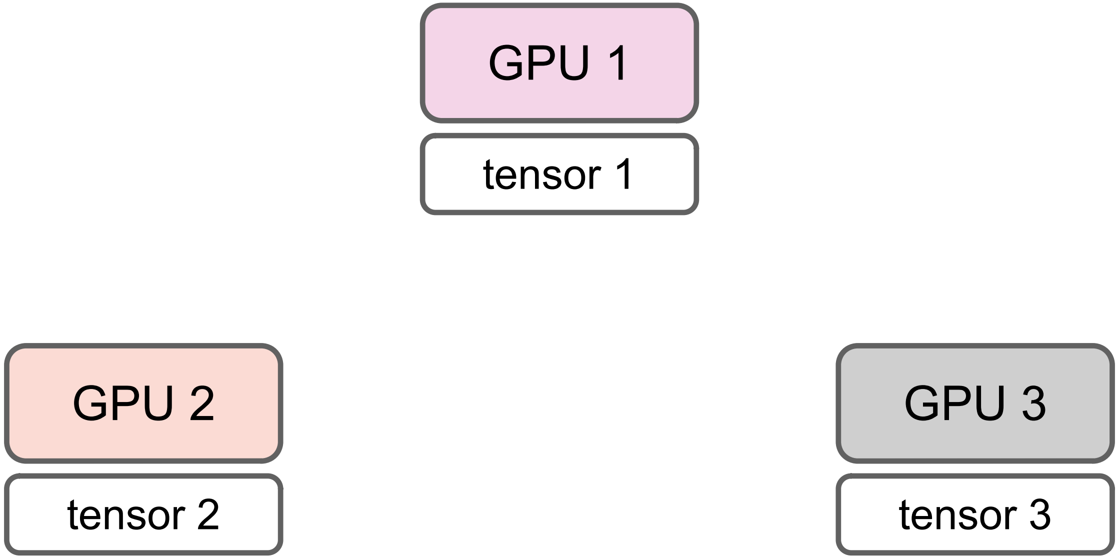
Each process has tensor of size n

Tensors aggregated (e.g. sum)

Result returned to each process

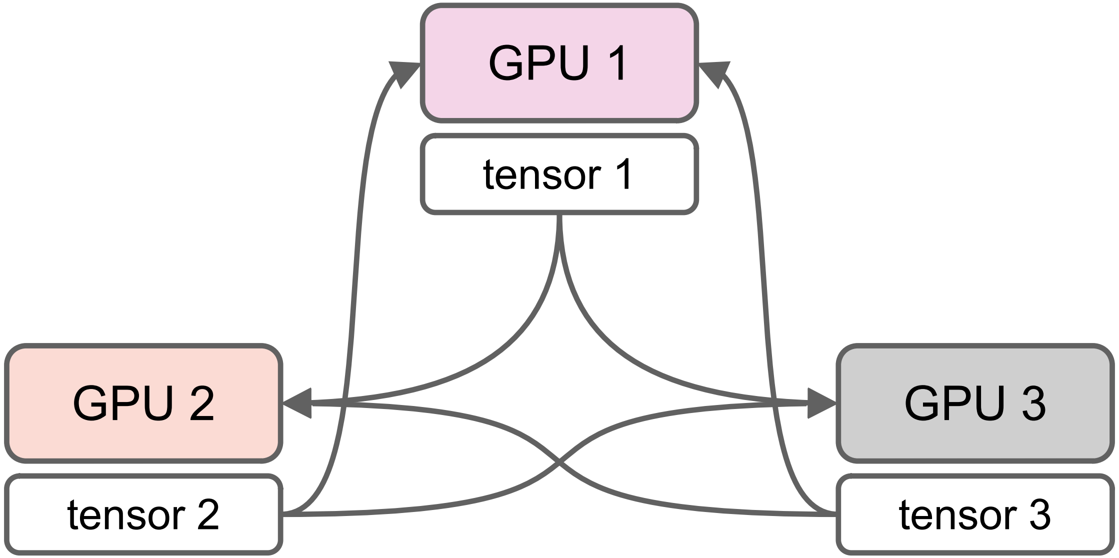
Credit: CS329S

DistributedDataParallel: All-Reduce



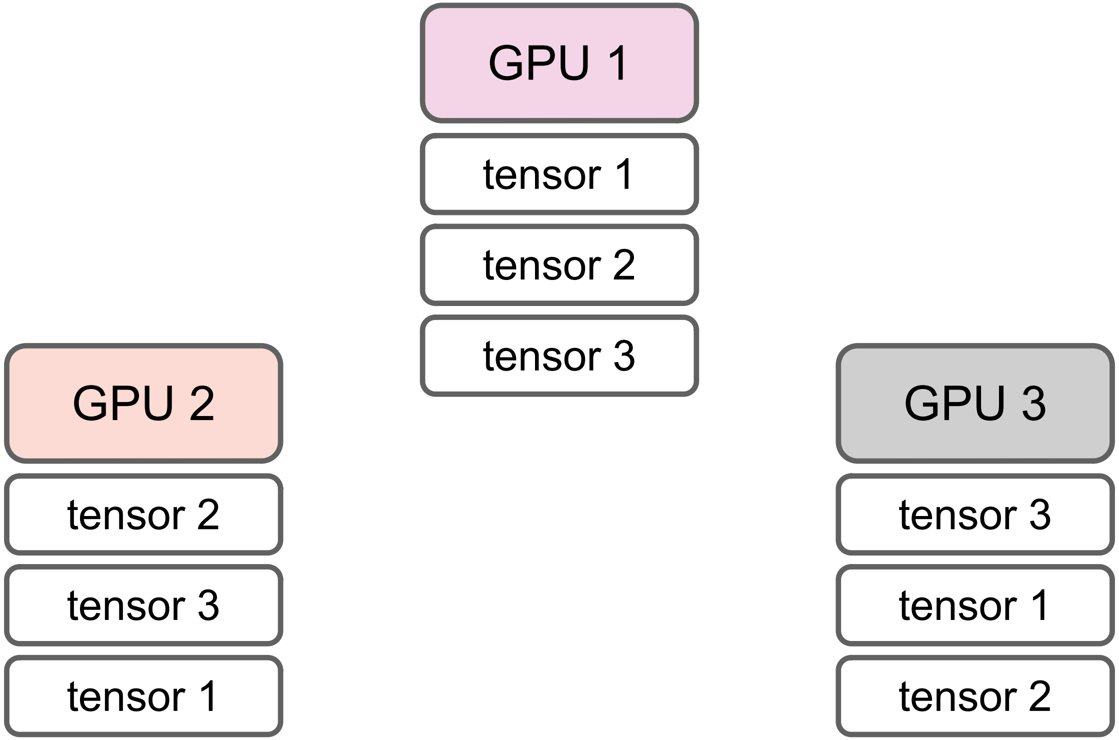
Credit: CS329S

DistributedDataParallel: All-Reduce



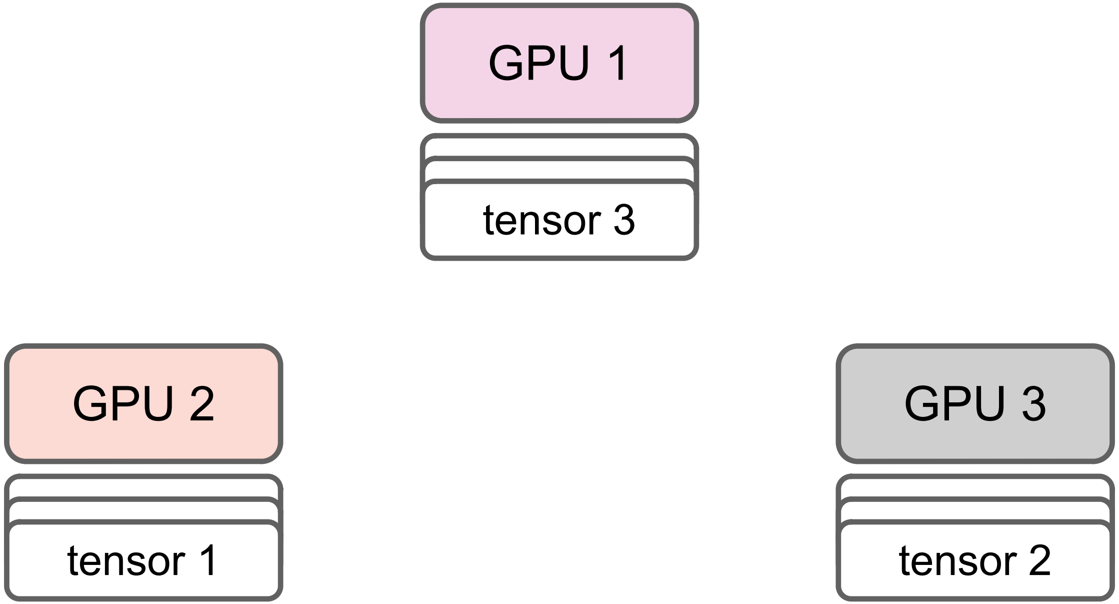
Credit: CS329S

DistributedDataParallel: All-Reduce



Credit: CS329S

DistributedDataParallel: All-Reduce



Credit: CS329S

DistributedDataParallel: All-Reduce

There are another all-reduce operations (e.g. ring all-reduce)

nVidia Collective Communications Library (nccl)

Credit: CS329S

DataParrel vs. DistributedDataParallel

DistributedDataParallel:
  multi-processing where a process a process is created for each GPU

DataParallel:
  Multi-threading, there are performance overhead caused by GIL of Python interpreter.

> The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code. Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an "infamous" feature of Python.
>
> \- Real Python

Topologies

Node: each node consist of multiple GPU devices.

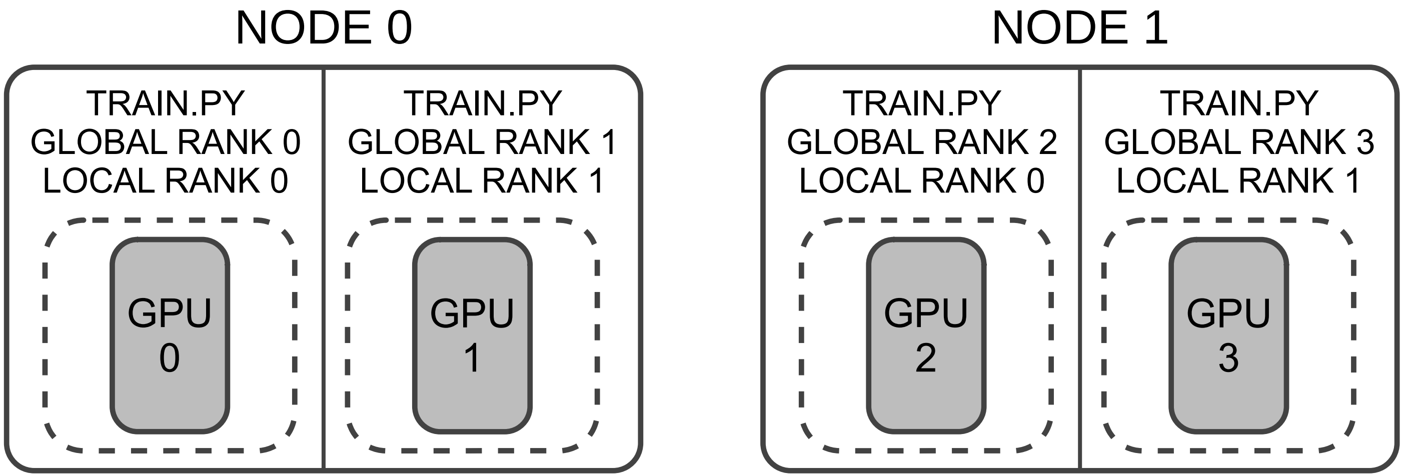
each node can run multiple copies of the DDP application, each of which processes its models on multiple GPUs.

World Size:

The total number of application processes running across all the nodes at one time is called the World Size

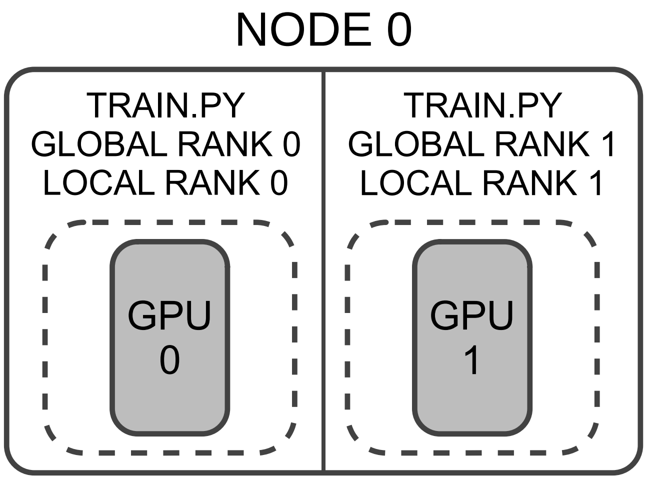
Local Rank:

Each application process is assigned two IDs: a local rank, and global rank



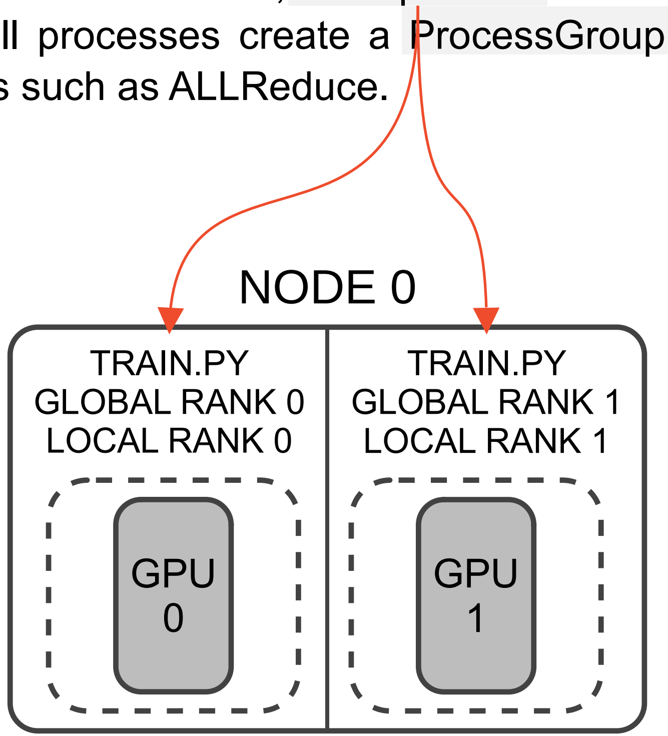
Launching a DDP

Independent of how a DDP application is launched, each process needs a mechanism to know its global and local ranks. Once this is known, all processes create a ProcessGroup that enables them to participate in collective communication operations such as ALLReduce.



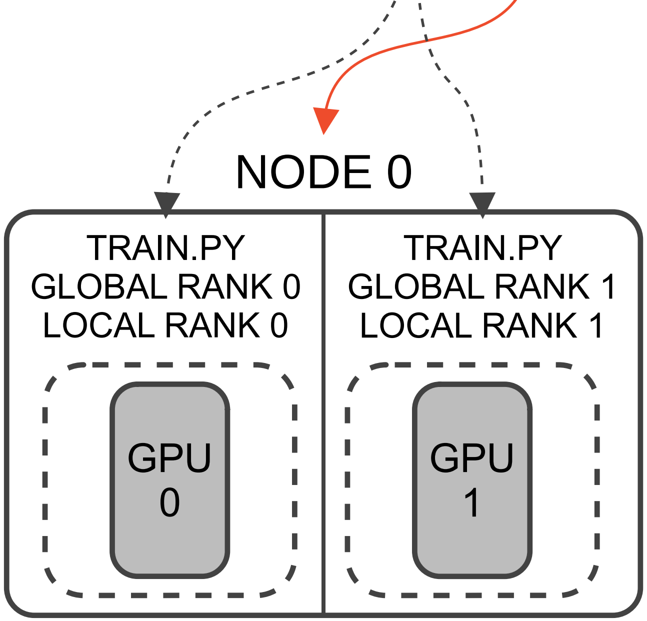
Launching a DDP

Independent of how a DDP application is launched, each process needs a mechanism to know its global and local ranks. Once this is known, all processes create a ProcessGroup that enables them to participate in collective communication operations such as ALLReduce.



Launching a DDP

Independent of how a DDP application is launched, each process needs a mechanism to know its global and local ranks. Once this is known, all processes create a ProcessGroup that enables them to participate in collective communication operations such as ALLReduce.



Launching a DDP

Conventient way to start multiple DDP processes: distributed launch.py script provided with PyTorch

Example:

```
$ python -m torch.distributed.launch.py --options… [python_train_script.py]
```

When the DDP applicatoin is started via launch.py, it passes the world size, global rank, master address and master port via environment variables and the local rank as a command-line parameter to each instance.

- Master address, Master port?

  They are needed when we launch the process with multiple nodes. If the nodes are different server sharing the same network, AllReduce is performed through the corresponding address and port.

Launching a DDP

Conventient way to start multiple DDP processes: distributed launch.py script provided with PyTorch

Example:

```
$ python -m torch.distributed.launch.py --options… [python_train_script.py]
```

CONVENTION for launch.py

1. It must provide an entry-point function for a single worker.

    ex) It should not launch subprocesses using `torch.multiprocessing.spawn`

2. It must use environment variables for initializing the process group.

Argument passing convention

The DDP application takes two command-line arguments:

1. `--local_rank:` This is passed in via launch.py
2. `--local_world_rank:` This is passed in explicitly and is typically either 1 or the number of GPUs per node.

Example:

```
import argparse

If __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--local_rank", type=int default=0)
    parser.add_argument("--local_world_size", type=int, default=1)
    args = parser.parse_args()
    spdm_main(args.local_world_size, args.local_rank)
```

Argument passing convention

Example:

```
def spdm_main(local_world_size, local_rank):
    env_dict = {
        key: os.environ[key]
        for key in (“MASTER_ADDR”, “MASTER_PORT”, “RANK”, “WORLD_SIZE”)
    }
    print(f”[{os.getpid()}] Initializing process group with: {env_dict}”)
    dist.init_process_group(backend=”nccl”)
  print(
    f"[{os.getpid()}] world_size = {dist.get_world_size()}, "
    + f"rank = {dist.get_rank()}, backend={dist.get_backend()}"
  )

  demo_basic(local_world_size, local_rank)

  # Tear down the process group
  dist.destroy_process_group()
```

Argument passing convention

Example:

```
def demo_basic(local_world_size, local_rank):

    # setup devices for this process. For local_world_size = 2, num_gpus = 8,
    # rank 1 uses GPUs [0, 1, 2, 3] and
    # rank 2 uses GPUs [4, 5, 6, 7].
    n = torch.cuda.device_count() // local_world_size
    device_ids = list(range(local_rank * n, (local_rank + 1) * n))

    print(
        f"[{os.getpid()}] rank = {dist.get_rank()}, "
        + f"world_size = {dist.get_world_size()}, n = {n}, device_ids = {device_ids}"
    )

    model = ToyModel().cuda(device_ids[0])
    ddp_model = DDP(model, device_ids)
    # Training code… (forward, backward, step...)
```

Launching a DDP using mp.spawn

`torch.multiprocessing.spawn():`

can be used to spawn multiple process.

torch.multiprocessing.spawn(fn, args=(), nprocs=1, joint=True, daemon=False, start_method='spawn')

Spawn nprocs processes that run fn with args.

Parameters

- fn (function)

Function is called as the entrypoint of the spawned process. This function must be defined at the top level of a module so it can be pickled and spawned. This is a requirement imposed by multiprocessing. The function is called as fn(i, *args), where i is process index and args is the passed through tuple of arguments

rank

Barrier

`torch.distributed.barrier()`: Synchroizes all processes.

> In parallel computing, a barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.
>
> \- Wikipedia

Example:

```python
@contextmanager
def torch_distributed_zero_first(local_rank: int):
    """
    Decorator to make all processes in distributed training wait for each local_master to do something.
    """
    if local_rank not in [-1, 0]:
        torch.distributed.barrier()
    yield
    if local_rank == 0:
        torch.distributed.barrier()
```

Barrier

Example:

```
@contextmanager
def torch_distributed_zero_first(local_rank: int):
    """
    Decorator to make all processes in distributed training wait for each local_master to do something.
    """
    if local_rank not in [-1, 0]:
        torch.distributed.barrier()
    yield
    if local_rank == 0:
        torch.distributed.barrier()
…

with torch_distributed_zero_first(rank):
    attempt_download(weights)    # download if not found locally
```

Barrier, DistributedSampler

Example:

```
def create_dataloader(..., rank=-1, world_size=1, ...):
    # Make sure only the first process in DDP process the dataset first, and the following others can use the cache
    with torch_distributed_zero_first(rank):
        dataset = LoadImagesAndLabels(...)
    ...

    sampler = torch.utils.data.distributed.DistributedSampler(dataset) if rank != -1 else None
    loader = torch.utils.data.DataLoader if image_weights else InfiniteDataLoader
    dataloader = loader(dataset,
                        batch_size=batch_size,
                        num_workers=nw,
                        sampler=sampler,
                        pin_memory=True,
                        collate_fn=LoadImagesAndLabels.collate_fn4 if quad else LoadImagesAndLabels.collate_fn)
    return dataloader, dataset
```

DistributedSampler

CLASS torch.utils.distributed.DistributedSampler(dataset, num_replicas=None, rank=None, shuffle=True, seed=0, drop_last=False)

Parameters

dataset – Dataset used for sampling

num_replicas – Number of processes participating in distributed training. By default, world_size is retrieved from the current distributed group.

Rank – Rank of the current process within num_replicas. By default, rank is retrieved from the current ditributed group.

Example:

```
>>> sampler = DistributedSampler(dataset) if is_distributed else None
>>> loader = DataLoader(dataset, shuffle=(sampler is None),
...                     sampler=sampler)
>>> for epoch in range(start_epoch, n_epochs):
...     if is_distributed:
...         sampler.set_epoch(epoch)
...     train(loader)
```

Save Checkpoint

CLASS torch.nn.parallel.DistributedDataParallel(module, device_ids=None, output_device=None, ...)

CLASS torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)

Example:

```
class DistributedDataParallel(nn.Module):
    def __init__(self, module, ...):
        …
        self.module = module
        …

>>> model = DDP(model, …)
>>> model.state_dict()
>>> model.module.state_dict()
```

Which one is real parameters of model?

Save Checkpoint

Example:

```
def is_parallel(model):
    return type(model) in (nn.parallel.DataParalle, nn.paralle.DistributedDataParallel)

>>> state_dict = model.module.state_dict() if is_parallel(model) else model.state_dict()
```