

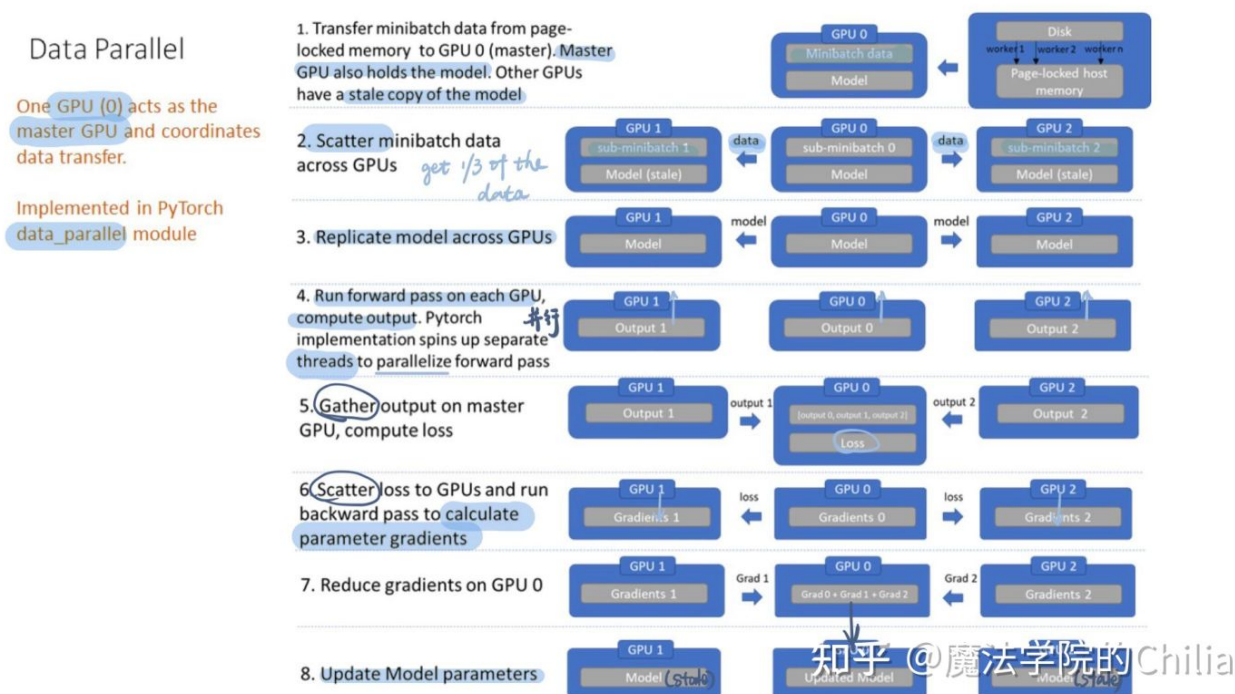
# Pytorch并行训练

## 1. torch.nn.DataParallel

只需要一行代码就可以实现DataParallel:

```
if torch.cuda.device_count() > 1: #判断是不是有多个GPU
    print("Let's use", torch.cuda.device_count(), "GPUs!")
    model = nn.DataParallel(model, device_ids=range(torch.cuda.device_count()))
```

DataParallel方法虽然代码非常简单，但是GPU会出现负载不均衡的问题（“一个gpu干活，其它gpu看戏”）。第一个GPU（12GB）可能占用了10GB，剩余的GPU却只使用了2GB -- 这是极大的浪费。那么，为什么会出现这样的现象呢？我们来仔细研究一下下面这个流程图就明白了：



我们首先需要指定一个GPU(0)作为master GPU，它必须承担最多的负载。

第一步：GPU(0)从锁页内存(分配主机内存时锁定该页，让其**不与磁盘交换**。必须保证自己内存空间充足的时候才用锁页内存！)中取得一个minibatch。

第二步：GPU(0)把这个minibatch去scatter到其他GPU上，每个GPU拿到1/n的数据，这就是data parallel的定义。

第三步：GPU(0)把模型也分配到其他GPU上，现在模型是同步的

第四步：在每个GPU上并行的做forward pass，得到每个sub-minibatch的输出

第五步：GPU(0)把所有GPU的输出gather到自己这里，计算loss

第六步：GPU(0)把loss scatter到其他GPU，分别进行backward pass计算梯度

第七步：GPU(0)把梯度集中起来进行reduce，计算出梯度的总和

第八步：GPU(0)用这个梯度更新自己的模型。而其他模型没有更新，是stale版本。

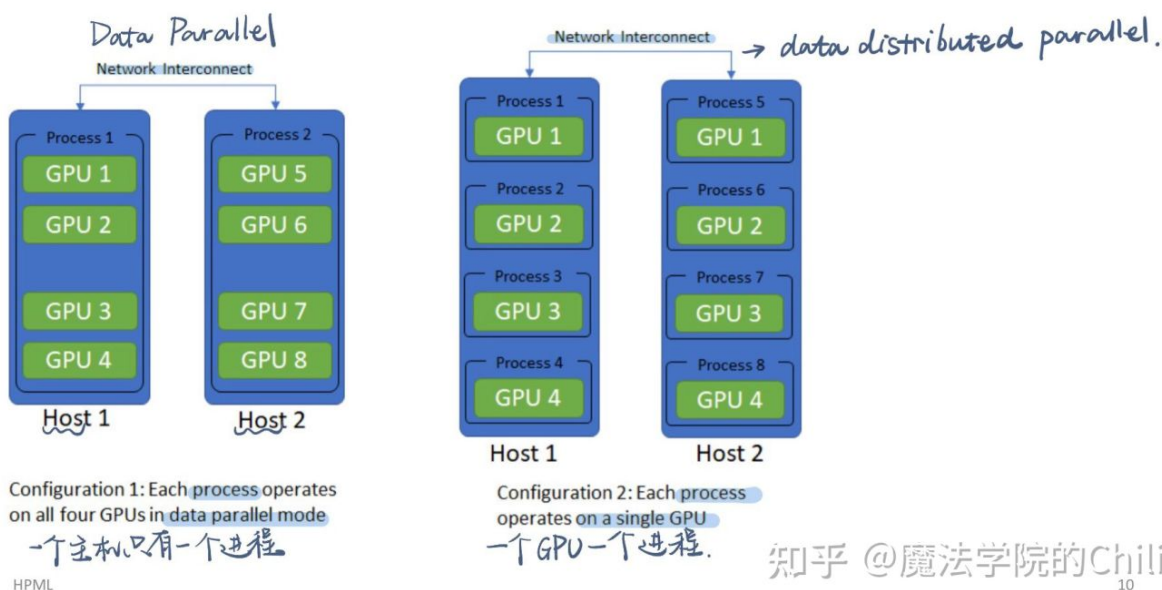
我们可以看到，nn.DataParallel做了很多不必要的操作，导致其性能不佳：

- 第一步的时候，从page-lock memory先取数据到GPU(0),然后由GPU(0)再进行分发。为什么不能直接从page-lock memory中分发到各个GPU呢？
- 第五步的时候，GPU(0)拿到每个GPU的output然后计算loss，那么为什么不能每个GPU自己计算自己的loss呢？
- 最后一步参数的更新是在GPU(0)上进行的，此时其他GPU上的模型和GPU (0) 不再同步。所以每次都需要broadcast 模型进行显式的同步。

由于loss的计算、梯度的reduce、模型的更新都是在GPU(0)上进行，所以造成了第一个GPU的负载远远大于剩余其他的GPU。

## 2. torch.nn.DistributedDataParallel

真正使用了多进程（process），而DataParallel只是使用了多线程（thread）：



DataParallel: 一个host一个process; DistributedDataParallel: 一个GPU一个process

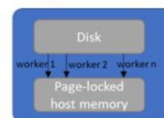
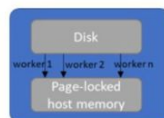
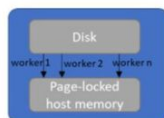
DistributedDataParallel是对每个GPU使用一个**进程**，适用于单机多卡和多机多卡的场景。它的主要工作流程如下图所示，可以看到相比DataParallel做了很多的优化：

## Distributed Data Parallel

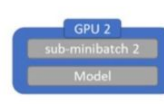
No master GPUs

Implemented in PyTorch  
DistributedDataParallel  
module

1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data



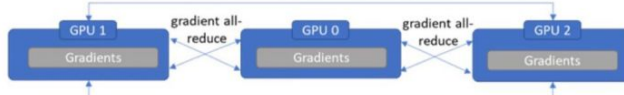
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output



4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation



5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required



知乎 @魔法学院的Chilia

每个GPU都有一个process，分别计算在这个GPU上的loss、gradient，然后用ring-all-reduce的方式进行聚合。得到综合的gradient之后同时更新本GPU上的模型参数，所以模型一直都是同步的，不需要手动同步。

DistributedDataParallel is proven to be significantly faster than torch.nn.DataParallel for single-node multi-GPU data parallel training.

调用过程：

(1) 初始化：sync the processes

```
#初始化使用nccl后端
torch.distributed.init_process_group(backend="nccl")
```

Gloo（用于CPU）和NCCL（用于GPU）都是用于多个node之间通信的库。从下面的图中我们可以看到，torch.nn.parallel.DistributedDataParallel是依赖于torch.distributed的，而torch.distributed提供的是多机通信的一些原语。它依赖于Gloo/NCCL/MPI backend。

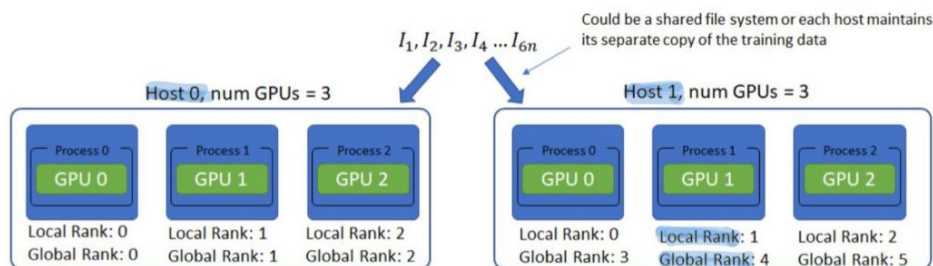
而torch.nn.DataParallel根本就是虚假的分布式！因为它只能使用于单个node，而且是单进程(single process)的，它的并行化全部依赖于线程(thread)。



知乎 @魔法学院的Chilia

(2) 使用DistributedSampler

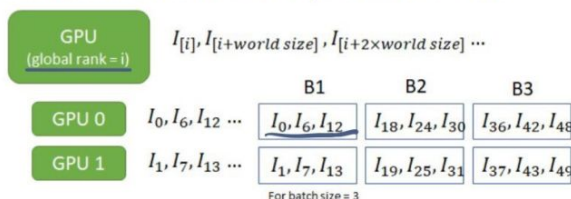
多gpu训练是，我们希望同一时刻在每个gpu上的数据是不一样的，这样相当于batch size扩大了N倍，因此起到了加速训练的作用。DistributedSampler用于把锁页内存的一个minibatch再进行切分，分配到不同的GPU进程中去。要保证这样的划分是没有overlap的。



Worldsize:  $\text{num Hosts} \times \text{num GPUs per host} = 2 \times 3 = 6$

Each process knows its **local rank** and **host number**. Can calculate **global rank** from this info.

$\text{Global rank} = \text{local rank} + \text{num GPUs per host} \times \text{host number}$



```
def __init__(self):
    # deterministically shuffle based on epoch
    g = torch.Generator()
    g.manual_seed(self.epoch)
    indices = torch.randperm(len(self.dataset), generator=g).tolist()

    # add extra samples to make it evenly divisible
    indices += indices[:self.total_size - len(indices)]
    assert len(indices) == self.total_size

    # subsample
    indices = indices[self.rank:self.total_size:self.num_replicas]
    assert len(indices) == self.num_samples
    print('len indices: {}'.format(len(indices)))
    return iter(indices)
```

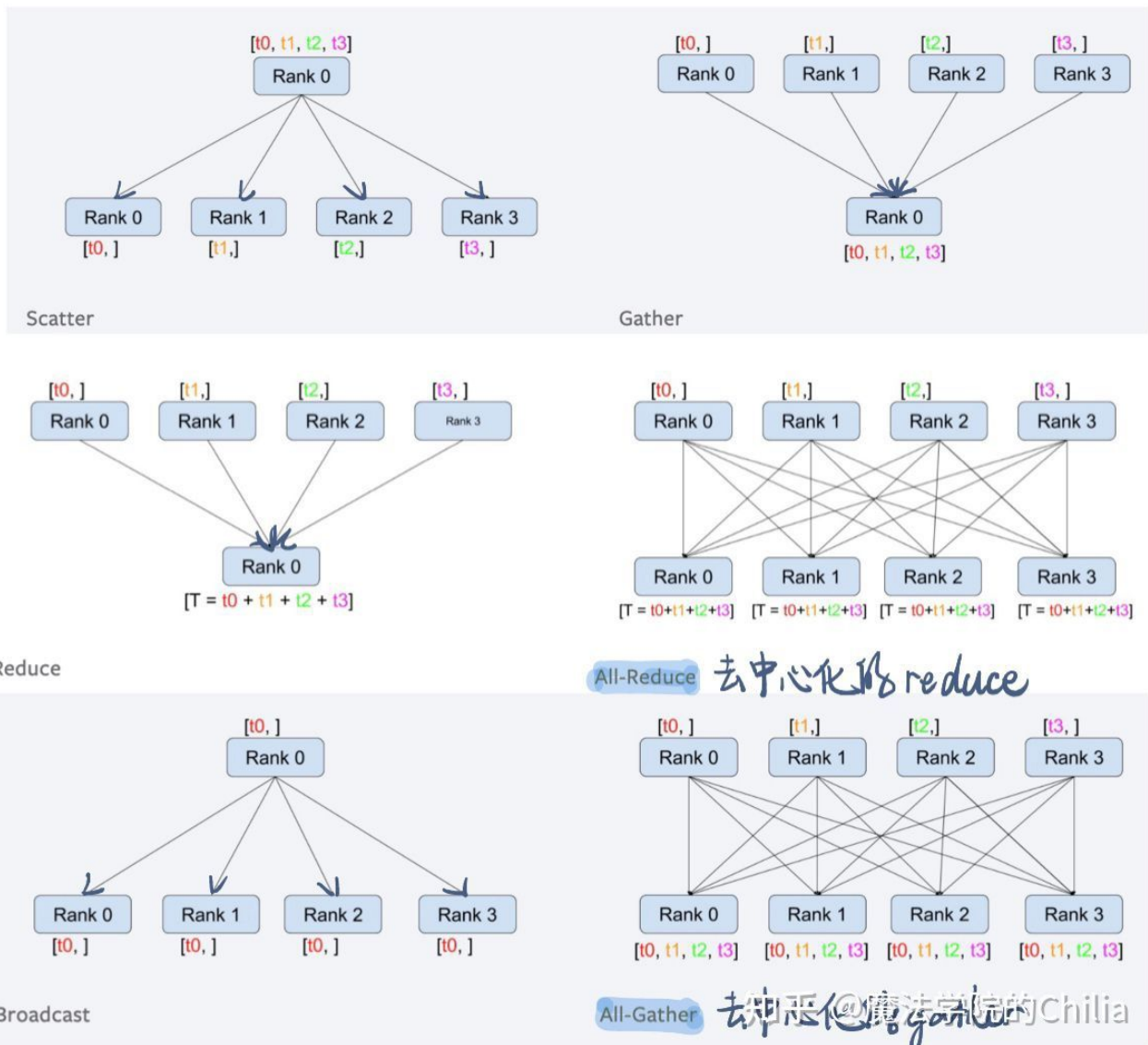
DistributedSampler (distributed.py)

每个进程都知道自己的local rank 和 host number,就可以推算出自己的global rank, 然后获得自己应有的那一份 sub-minibatch数据，保证每个进程拿到的数据没有重叠。

### 3. torch.distributed collectives原语

- **dist.broadcast**(tensor,src,group): 对于group中的所有GPU编号，把tensor从src GPU分发到其他的GPU process中
- **dist.reduce**(tensor,dst,op,group): Applies op to all tensor in group and store the result in dst.
- **dist.all\_reduce**(tensor,op,group): Same as reduce, but the result is stored in all processes.
- **dist.scatter**(tensor,src,scatter\_list,group): copy the ith tensor scatter\_list[i] to the ith process.
- **dist.gather**(tensor, dst, gather\_list, group): copy tensor from all processes in group in dst
- **dist.all\_gather**(tensor\_list,tensor,group): copy tensor from all processes to tensor\_list, on all processes.





下面我们来看一个例子：

```
import torch
import argparse
from torch import distributed as dist

print(torch.cuda.device_count()) # 打印gpu数量
torch.distributed.init_process_group(backend="nccl") # 并行训练初始化, 'nccl'模式
print('world_size', torch.distributed.get_world_size()) # 打印当前进程数

# 下面这个参数需要加上, torch内部调用多进程时, 会使用该参数, 对每个gpu进程而言, 其local_rank都是不同的;
parser.add_argument('--local_rank', default=-1, type=int)
args = parser.parse_args()
torch.cuda.set_device(args.local_rank) # 设置gpu编号为local_rank;此句也可能看出local_rank的值是什么
```

```

'''
多卡训练加载数据：
注意shuffle与sampler是冲突的，并行训练需要设置sampler，此时务必
# 要把shuffle设为False。但是这里shuffle=False并不意味着数据就不会乱序了，而是乱序的方式交给
# sampler来控制，实质上数据仍是乱序的。
'''

train_sampler = torch.utils.data.distributed.DistributedSampler(My_Dataset)
dataloader = torch.utils.data.DataLoader(ds,
                                         batch_size=batch_size,
                                         shuffle=False,
                                         num_workers=16,
                                         pin_memory=True, ##锁页内存
                                         drop_last=True,
                                         sampler=self.train_sampler)

'''
多卡训练的模型设置：
'''

def average_gradients(model): ##每个gpu上的梯度求平均
    size = float(dist.get_world_size())
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op = dist.reduce_op.SUM)
        param.grad.data /= size

My_model = My_model.cuda(args.local_rank) # 将模型拷贝到每个gpu上.直接.cuda()也行，因为多进
程时每个进程的device号是不一样的
My_model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(My_model) # 设置多个gpu的BN同步
My_model = torch.nn.parallel.DistributedDataParallel(My_model,
                                                    device_ids=[args.local_rank],
                                                    output_device=args.local_rank,
                                                    find_unused_parameters=False,
                                                    broadcast_buffers=False)

'''开始多卡训练：'''
for epoch in range(200):
    train_sampler.set_epoch(epoch) # 这句莫忘，否则相当于没有shuffle数据
    My_model.train()
    for idx, sample in enumerate(dataloader):
        inputs, targets = sample[0].cuda(local_rank, non_blocking=True),
sample[1].cuda(local_rank, non_blocking=True)
        opt.zero_grad()
        output = My_model(inputs)
        loss = My_loss(output, targets)
        loss.backward()
        average_gradient(My_model) ##计算梯度的平均
        opt.step() ##根据梯度更新模型

'''多卡测试(evaluation):'''
if local_rank == 0:
    My_model.eval()
    with torch.no_grad():
        acc = My_eval(My_model)

```

```
torch.save(My_model.module.state_dict(), model_save_path)
dist.barrier() # 这一句作用是：所有进程(gpu)上的代码都执行到这，才会执行该句下面的代码

'''
其它代码
'''
```

在分布式的GPU上进行SGD，每个GPU在计算完梯度之后，需要把不同GPU上的梯度进行all-reduce，即计算所有GPU上梯度的平均值：`dist.all_reduce(param.grad.data, op=dist.reduce_op.SUM)`，意为我们对每个模型参数的梯度进行all\_reduce, 采用求和的形式计算all\_reduce.

最后一步，运行上述代码的方式（8卡为例）：

```
python3 -m torch.distributed.launch --nproc_per_node=8 DDP.py
```