Distributed Data Parallel

Warning

The implementation of :class:`torch.nn.parallel.DistributedDataParallel` evolves over time. This design note is written based on the state as of v1.4.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]ddp.rst`, line 7); *backlink*

Unknown interpreted text role "class".

:class:`torch.nn.parallel.DistributedDataParallel` (DDP) transparently performs distributed data parallel training. This page describes how it works and reveals implementation details.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]ddp.rst`, line 11); *backlink*

Unknown interpreted text role "class".

Example

Let us start with a simple :class:`torch.nn.parallel.DistributedDataParallel` example. This example uses a :class:`torch.nn.Linear` as the local model, wraps it with DDP, and then runs one forward pass, one backward pass, and an optimizer step on the DDP model. After that, parameters on the local model will be updated, and all models on different processes should be exactly the same.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]ddp.rst`, line 18); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]ddp.rst`, line 18); *backlink*

Unknown interpreted text role "class".

```
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP


def example(rank, world_size):
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)
    # create local model
    model = nn.Linear(10, 10).to(rank)
    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # forward pass
    outputs = ddp_model(torch.randn(20, 10).to(rank))
    labels = torch.randn(20, 10).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()
    # update parameters
    optimizer.step()

def main():
    world_size = 2
    mp.spawn(example,
        args=(world_size,),
        nprocs=world_size,
        join=True)

if __name__=="__main__":
    # Environment variables which need to be
    # set when using c10d's default "env"
    # initialization mode.
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "29500"
    main()
```

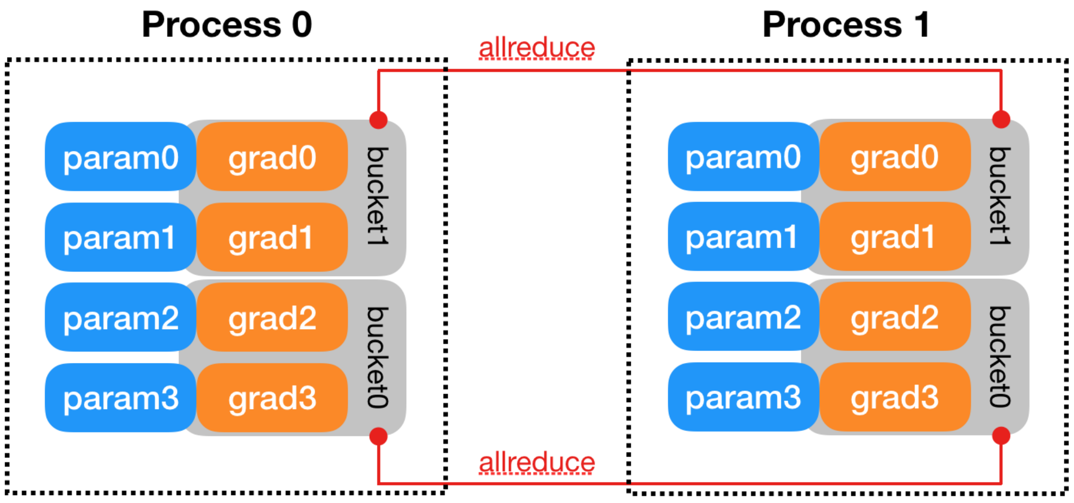
Internal Design

This section reveals how it works under the hood of :class:`torch.nn.parallel.DistributedDataParallel` by diving into details of every step in one iteration.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]ddp.rst`, line 73); *backlink*

Unknown interpreted text role "class".

- **Prerequisite**: DDP relies on c10d `ProcessGroup` for communications. Hence, applications must create `ProcessGroup` instances before constructing DDP.
- **Construction**: The DDP constructor takes a reference to the local module, and broadcasts `state_dict()` from the process with rank 0 to all other processes in the group to make sure that all model replicas start from the exact same state. Then, each DDP process creates a local `Reducer`, which later will take care of the gradients synchronization during the backward pass. To improve communication efficiency, the `Reducer` organizes parameter gradients into buckets, and reduces one bucket at a time. Bucket size can be configured by setting the *bucket_cap_mb* argument in DDP constructor. The mapping from parameter gradients to buckets is determined at the construction time, based on the bucket size limit and parameter sizes. Model parameters are allocated into buckets in (roughly) the reverse order of `Model.parameters()` from the given model. The reason for using the reverse order is because DDP expects gradients to become ready during the backward pass in approximately that order. The figure below shows an example. Note that, the `grad0` and `grad1` are in `bucket1`, and the other two gradients are in `bucket0`. Of course, this assumption might not always be true, and when that happens it could hurt DDP backward speed as the `Reducer` cannot kick off the communication at the earliest possible time. Besides bucketing, the `Reducer` also registers autograd hooks during construction, one hook per parameter. These hooks will be triggered during the backward pass when the gradient becomes ready.
- **Forward Pass**: The DDP takes the input and passes it to the local model, and then analyzes the output from the local model if `find_unused_parameters` is set to `True`. This mode allows running backward on a subgraph of the model, and DDP finds out which parameters are involved in the backward pass by traversing the autograd graph from the model output and marking all unused parameters as ready for reduction. During the backward pass, the `Reducer` would only wait for unready parameters, but it would still reduce all buckets. Marking a parameter gradient as ready does not help DDP skip buckets as for now, but it will prevent DDP from waiting for absent gradients forever during the backward pass. Note that traversing the autograd graph introduces extra overheads, so applications should only set `find_unused_parameters` to `True` when necessary.
- **Backward Pass**: The `backward()` function is directly invoked on the loss `Tensor`, which is out of DDP's control, and DDP uses autograd hooks registered at construction time to trigger gradients synchronizations. When one gradient becomes ready, its corresponding DDP hook on that grad accumulator will fire, and DDP will then mark that parameter gradient as ready for reduction. When gradients in one bucket are all ready, the `Reducer` kicks off an asynchronous `allreduce` on that bucket to calculate mean of gradients across all processes. When all buckets are ready, the `Reducer` will block waiting for all `allreduce` operations to finish. When this is done, averaged gradients are written to the `param.grad` field of all parameters. So after the backward pass, the *grad* field on the same corresponding parameter across different DDP processes should be the same.
- **Optimizer Step**: From the optimizer's perspective, it is optimizing a local model. Model replicas on all DDP processes can keep in sync because they all start from the same state and they have the same averaged gradients in every iteration.



Note

DDP requires `Reducer` instances on all processes to invoke `allreduce` in exactly the same order, which is done by always running `allreduce` in the bucket index order instead of actual bucket ready order. Mismatched `allreduce` order across processes can lead to wrong results or DDP backward hang.

Implementation

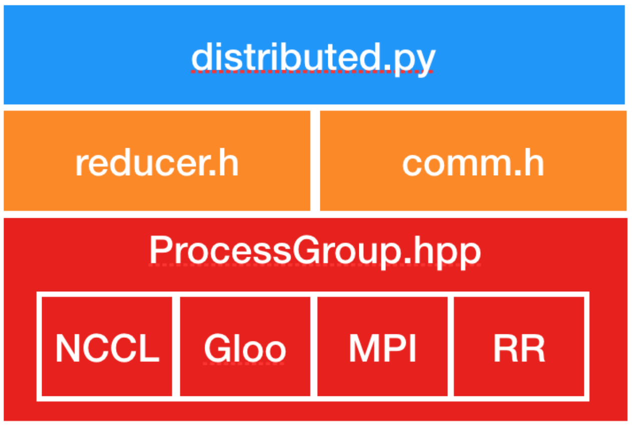
Below are pointers to the DDP implementation components. The stacked graph shows the structure of the code.

ProcessGroup

- ProcessGroup.hpp: contains the abstract API of all process group implementations. The `c10d` library provides 3 implementations out of the box, namely, *ProcessGroupGloo*, *ProcessGroupNCCL*, and *ProcessGroupMPI*. `DistributedDataParallel` uses `ProcessGroup::broadcast()` to send model states from the process with rank 0 to others during initialization and `ProcessGroup::allreduce()` to sum gradients.
- Store.hpp: assists the rendezvous service for process group instances to find each other.

DistributedDataParallel

- distributed.py: is the Python entry point for DDP. It implements the initialization steps and the `forward` function for the `nn.parallel.DistributedDataParallel` module which call into C++ libraries. Its `_sync_param` function performs intra-process parameter synchronization when one DDP process works on multiple devices, and it also broadcasts model buffers from the process with rank 0 to all other processes. The inter-process parameter synchronization happens in `Reducer.cpp`.
- comm.h: implements the coalesced broadcast helper function which is invoked to broadcast model states during initialization and synchronize model buffers before the forward pass.
- reducer.h: provides the core implementation for gradient synchronization in the backward pass. It has three entry point functions:
  - `Reducer`: The constructor is called in `distributed.py` which registers `Reducer::autograd_hook()` to gradient accumulators.
  - `autograd_hook()` function will be invoked by the autograd engine when a gradient becomes ready.
  - `prepare_for_backward()` is called at the end of DDP forward pass in `distributed.py`. It traverses the autograd graph to find unused parameters when `find_unused_parameters` is set to `True` in DDP constructor.


distributed.py
reducer.h
comm.h
ProcessGroup.hpp
NCCL
Gloo
MPI
RR