PyTorch Distributed and Parallel Training - 1

20230405
박현우

Table of Contents

1. Overview of Distributed Training

1. Main components of torch.distributed
2. Basic Modules for Data-Parallel training in PyTorch
3. Main Paradigms of Distributed/Parallel training
4. Summary

1.1 Main components of torch.distributed

Features in `torch.distributed` can be categorized into three main components (i.e. low-level Building blocks):

1. Distributed Data-Parallel Training (DDP)
2. ProcessGroup Backend (RPC)
3. Collective Communication (c10d) library

1.1.1 Distributed Data-Parallel Training (DDP)

Aligned for single-program multiple-data training paradigm.

DDP

1. takes care of gradient communication to keep model replicas synchronized
   (c.f. complete model replication of DP)
2. and overlaps it with the gradient computations to speed up training.

1.1.2 ProcessGroup Backend (RPC)

Aligned for supporting general training structures beyond data-parallel training, such as

1. distributed pipeline parallelism,
2. parameter server paradigm,
3. and combinations of DDP with other training paradigms.

RPC Backend of PyTorch

1. helps manage remote object lifetime
2. and extends the autograd engine beyond machine boundaries.

1.1.3 Collective Communication (c10d) library

Aligned for sending tensors across processes within a group.

c10d offers both

1. Collective Communication APIs
   (e.g., all_reduce and all_gather) (explained later)
2. P2P communication APIs
   (e.g., send and isend).

c10d is the most fundamental library for communication, for example,

1. DDP leverages Collective communications
2. RPC leverages P2P communications.

1.2 Basic Modules for Data-Parallel training

Types of Data-Parallelism and corresponding modules.

<table>
<tr><th></th><th>Single Machine</th><th>Multiple Machine</th></tr>
<tr><td>Single GPU</td><td>Simple PyTorch</td><td rowspan="2">DDP + torchrun
or torch.distributed.elastic.</td></tr>
<tr><td>Multiple GPU</td><td>DataParallel (DP)
or DistributedDataParallel (DDP)</td></tr>
</table>

1.2 Basic Modules for Data-Parallel training

1. Simple PyTorch
   for simplest, small-sized model.
2. DataParallel (DP)
   for speed up training with minimal code changes.
3. DistributedDataParallel (DDP)
   for faster speed than DP but with more code.
4. DDP + torchrun
   for scaling across cluster of machines.
5. torch.distributed.elastic
   for find-grained error handling and dynamic allocation/drop of machines during training.

1.2.1 DataParallel (DP)

The DataParallel package enables single-machine multi-GPU parallelism with the lowest coding hurdle;
only requires a one-line change.

```
class Model(nn.Module):
    # Our model

    def __init__(self, input_size, output_size):
        super(Model, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = self.fc(input)
        print("\tIn Model: input size", input.size(),
              "output size", output.size())

        return output
```

```
model = Model(input_size, output_size)
if torch.cuda.device_count() > 1:
  print("Let's use", torch.cuda.device_count(), "GPUs!")
                                        [10, ...], [10, ...] on 3 GPUs
  model = nn.DataParallel(model)

model.to(device)
```

1.2.1 DataParallel (DP)

Trading-off with simplicity, DP usually does not offer the best performance because

1. it replicates the complete model (i.e. updated model with gradients from all devices) in every forward pass,
2. and its single-process multi-thread parallelism naturally suffers from GIL contention.

NOTE : PyTorch officially NOT recommend using DP in production.

1.2.2 torch.distributed.elastic

DDP doesn’t provide Failure Recovery;
e.g. from OOM or network timeout.

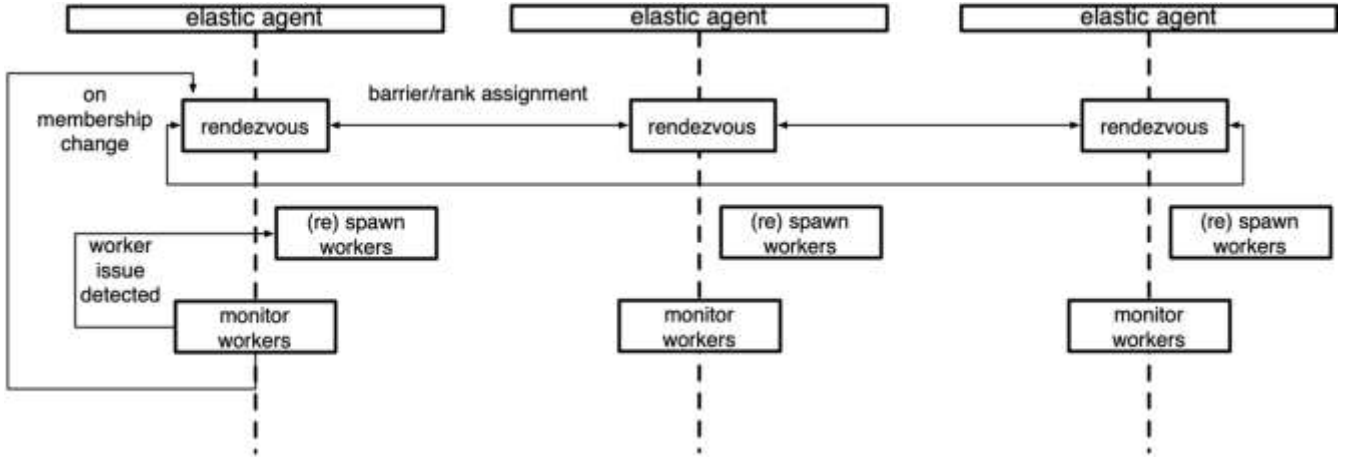
Failures in cluster cannot be handled using a standard try-except construct.
This is because DDP requires all processes to operate in a closely synchronized manner;

More specifically, all `All_Reduce` communications in all processes must match (synchronized).
= Exception in one of the processes causes desynchronization, resulting in cluster crash or hang.

1.2.2 torch.distributed.elastic

Therefore, torch.distributed.elastic provides

1. Fault Tolerance/Handling for cluster
2. Dynamic pool of machines in cluster (Elasticity).



(Further explained in RPC-architecture)

1.3 Main Paradigms of Distributed/Parallel training

i.e. Higher level paradigm or architecture built upon PyTorch modules.

1. Data-Parallel Training (DP, DDP)
2. FSDP
3. RPC-based Distributed Trainings

Simplicity

Scalability,
Flexibility,
Customizability

DP

DDP

FSDP

RPC-based

Limited to Single Machine

Can extend to Multiple Machines

Can extend to large Cluster

Fully Customizable

1.4 Summary

Low Level

High Level

Components		
DDP	RPC Backend	c10d library

Modules		
	Single Machine	Multiple Machines
Single GPU	Simple PyTorch	DDP + torchrun or torch.distributed .elastic.
Multiple GPU	DP or DDP	

Paradigms		
Data-Parallel Training (DP, DDP)	FSDP	RPC-based Distributed Trainings

2. Collective Communications

Collective Communications is to communicate across multiple processes in a cluster.
c.f. Point-to-Point communication

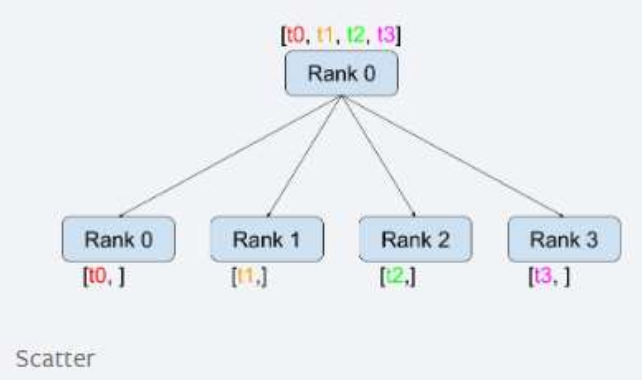
There are 6 types of Collective Communications.

Terminology

1. All processes are in the World.
2. A Group is a subset of all processes.

Thus, multiple groups can reside in the World.

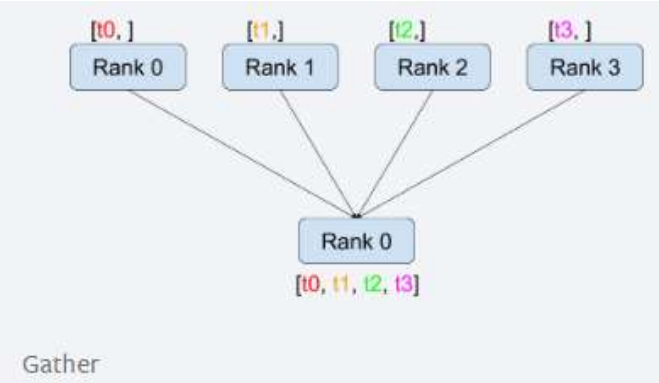
2.1. Scatter, Gather



Scatter

```
dist.scatter(tensor, scatter_list, src, group)
```

Copies the *i*-th tensor `scatter_list[i]` to the *i*-th process.

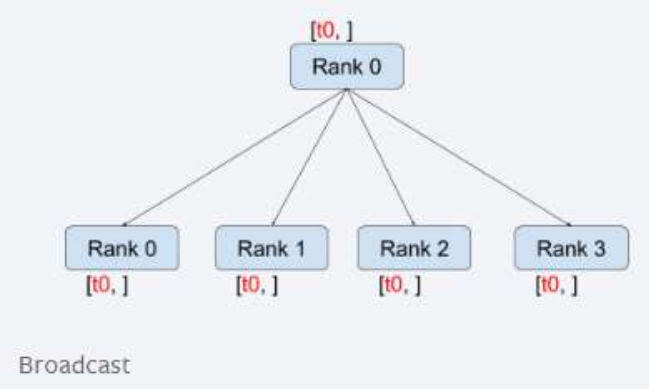


Gather

```
dist.gather(tensor, gather_list, dst, group)
```

Copies `tensor` from all processes in `dst`.

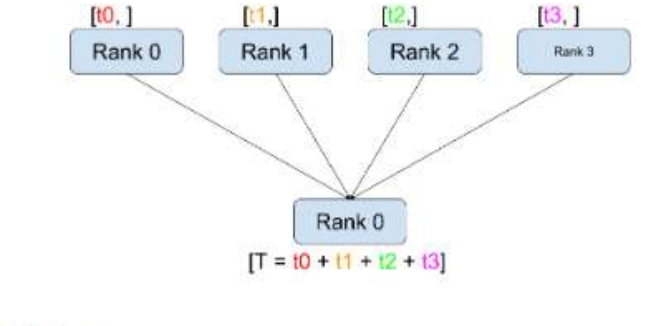
2.2. Broadcast, Reduce



Broadcast

`dist.broadcast(tensor, src, group)`

Copies `tensor` from `src` to all other processes.

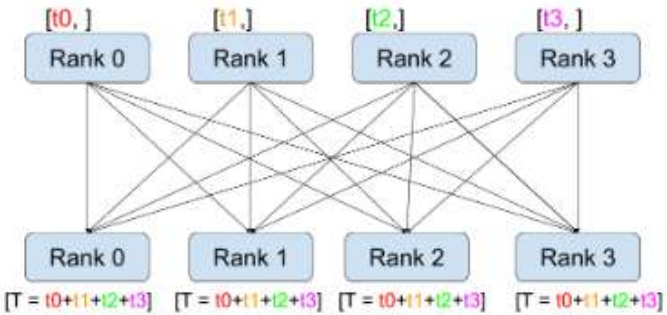


Reduce

`dist.reduce(tensor, dst, op, group)`:

1. Applies `op` to every `tensor`
2. and stores the result in `dst`.

2.3. All-Reduce



All-Reduce

All-Reduce

```
dist.all_reduce(tensor, op, group)
```

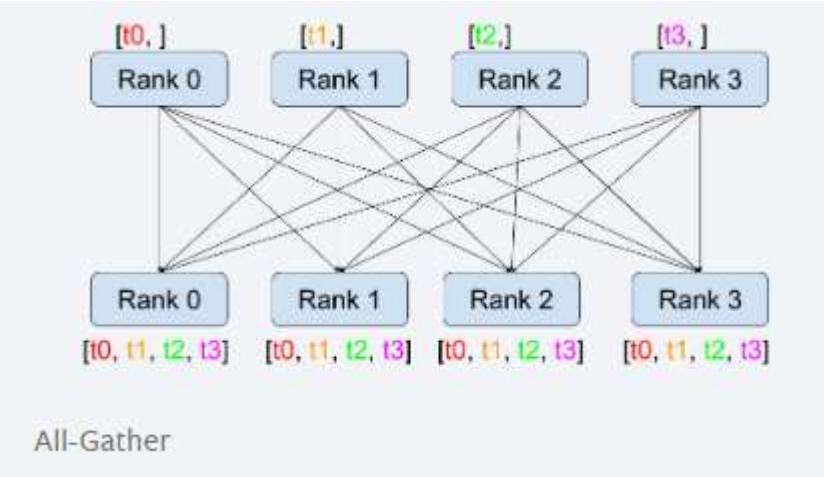
Same as Reduce,
but the result is stored in all processes.

2.3. All-Reduce

Implementation

1. All processes are individually communicating to each other.
   (Have O(N^2) communications)
2. Reduce to one process (Master) and master propagates back to all other processes.
   (Impose severe load on master process)

2.4 All-Gather



All-Gather

`dist.all_gather(tensor_list, tensor, group)`

Copies `tensor` from all processes to `tensor_list`, on all processes.

Simply, All-Reduce without op.

2.5 Notes

All-Reduce and All-Gather communications are Extremely expensive.

Thus, need to

1. decompose these ops into simple ops (Reduce, Broadcast, etc)
2. compute them distributely in parallel.

Thus, other strategies are proposed to solve such communication overload

e.g. Ringed All-Reduce in FSDP.

3. DDP

1. What is DDP
2. How DDP operates
3. Comparison to DP
4. Caveat of DDP
5. How to use DDP
6. DDP with Model parallelism
7. DDP with torchrun (Multi- Machines, Multi- Devices)

3.1. What is DDP

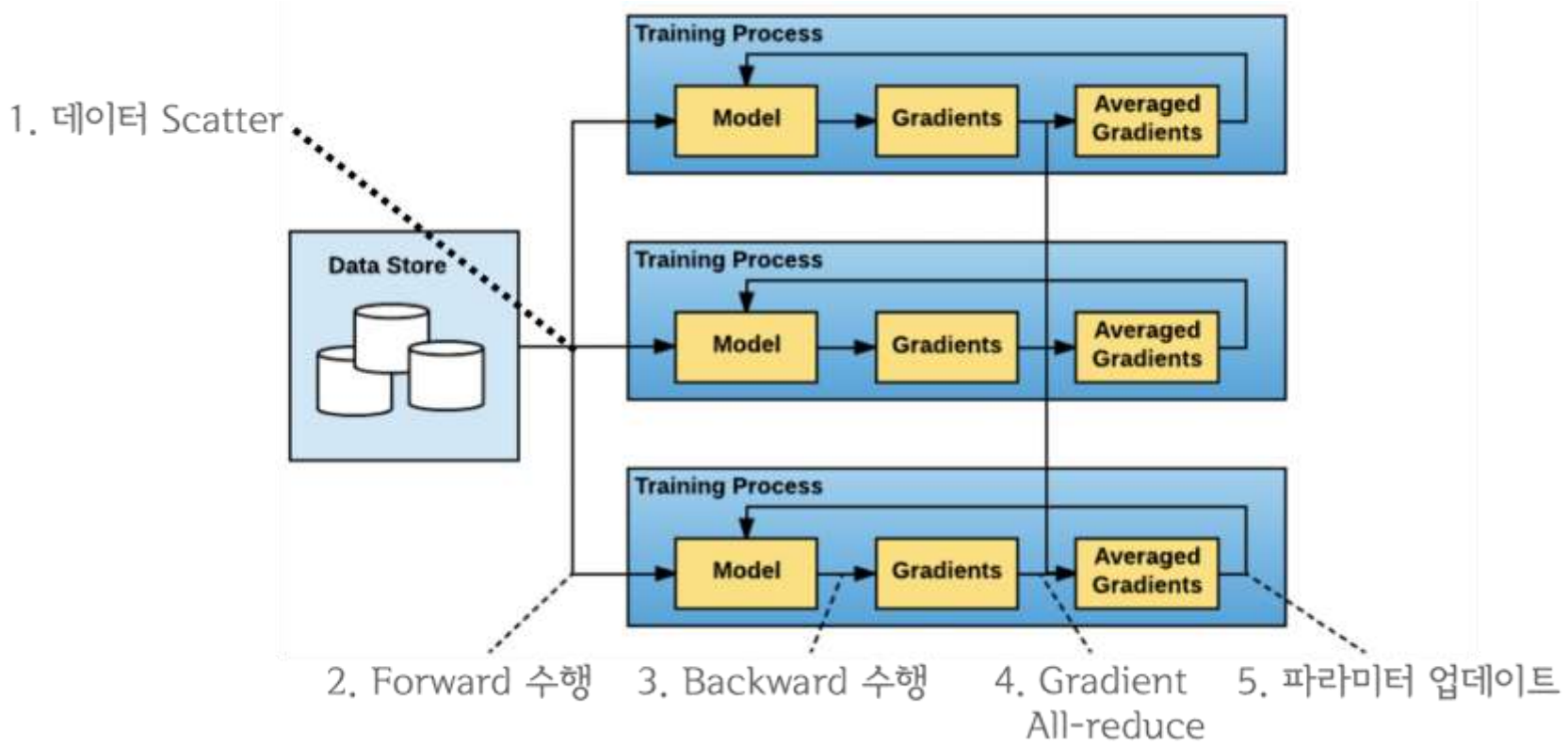
DDP implements data AND model parallelism which can run across multiple machines.

DDP can be run on both in

1. Single Machine - Multiple Devices
2. Multiple Machines - Multiple Devices

in which for latter, torchrun is required.

3.2 How DDP operates



3.2 How DDP operates

Specifically:

1. DDP registers an autograd hook for each parameter (in model.parameters())
2. The hook will fire
   when the corresponding gradient is computed in the backward pass.
3. Then DDP uses that signal to trigger gradient synchronization across processes.
4. Each process updates its own model with given (reduced) gradient.

3.3 Comparison to DP

	DP	DDP
Mechanism	Single-process, multi-thread	Multi-process
Scaling	only works on a single machine.	works for both single- and multi- machine.
Parallelism	only Data parallelism	both Data and Model parallelism

For DP,

1. GIL contention deteriorates multi-threading performance
2. can’t scale on multiple machines
3. can’t train model with size larger than device.

3.3 Comparison to DP

Also, DDP wraps training jobs with Process,
abstracting jobs from underlying devices.

This allows Heterogeneous composition of cluster.

In contrast, DP simply runs training on Thread,
restricting to have Homogenous devices.

3.3 Comparison to DP

Even on a single machine, DP is slower than DDP because

1. GIL contention in multi-threading
   c.f. Multiprocessing is free from GIL
2. DP replicates full model per-iteration
   c.f. Only replicates gradients
3. and additional communication overhead introduced by
   a. scattering inputs
   b. gathering outputs.

      c.f each machine contains own data.

3.4 Caveat of DDP - Skewed Processing Speeds

There are 3 distributed synchronization points in DDP:

1. Constructor
2. Forward pass
3. Backward pass

In ideal, each process would

1. launch the same number of synchronizations
2. and reach these synchronization points in the same order

3.4 Caveat of DDP - Skewed Processing Speeds

Although DDP is allowed to construct Heterogeneous cluster,
DDP doesn’t provide delicate, flexible synchronization policy across processes.

Thus, fast processes might

1. arrive early
2. and timeout while waiting for slower processes. (# elastic)

Such skewed processing speeds can also be occurred by

1. Network delays
2. Resource contentions
3. or unpredictable workload spikes.

3.4 Caveat of DDP - Skewed Processing Speeds

Unlike elastic, DDP cannot recover from such timeout by itself.

Therefore, users are responsible for balancing workload distributions across processes.

Also, user must pass a sufficiently large timeout value when initializing training, so none of processes timeout to die.

Due to such inherent limitation of DDP,
PyTorch recommends to use more advanced strategies for large clusters

e.g. FSDP, RPC-based architectures

3.5 How to use DDP

```python
def demo_basic(rank, world_size):
    print(f"Running basic DDP example on rank {rank}.")
    setup(rank, world_size)

    # create model and move it to GPU with id rank
    model = ToyModel().to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(rank)
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()


def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)
```

To use DDP,
simply wrap model with DDP,
with device_id on which the process will run.

(rank is automatically passed by mp.spawn)

The recommended way to use DDP is
to spawn one process for each model replica,
where a model replica can span multiple devices.

(run_demo 1:1 maps processes to devices)

3.6 DDP with Model Parallelism

```
model = ToyModel().to(rank)
ddp_model = DDP(model, device_ids=[rank])
```

DDP with Model parallelism is the same as with Data parallelism, except device_ids and output_device must NOT be set.

They will be automatically set by forward() of the model or by the user/application.

```
def demo_model_parallel(rank, world_size):
    print(f"Running DDP with model parallel example on rank {rank}.")
    setup(rank, world_size)

    # setup mp_model and devices for this process
    dev0 = (rank * 2) % world_size
    dev1 = (rank * 2 + 1) % world_size
    mp_model = ToyMpModel(dev0, dev1)
    ddp_mp_model = DDP(mp_model)

    loss_fn = nn.MSELoss()
```

3.6 DDP with Model Parallelism

However, although DDP supports Model parallelism, it is not efficient.

For example, DDP depends on All-Reduce / All-Gather to synchronize both Model and update gradients,

which are very expensive operation.

FSDP solves such inefficiencies of DDP for Model parallelism.

3.7 DDP with torchrun (Multi-machines, Multi-devices)

To initialize DDP jobs on all nodes in the cluster,
run torchrun command:
(elastic_ddp.py is same as previous examples)

```
torchrun --nnodes=2 --nproc_per_node=8 --rdzv_id=100 --rdzv_backend=c10d --rdzv_endpoint
=$MASTER_ADDR:29400 elastic_ddp.py
```

meaning that DDP script is running on 2 hosts (nnodes) with 8 processes (nproc_per_node) for each.

3.7 DDP with torchrun (Multi-machines, Multi-devices)

However, to run this torchrun command across cluster,
external Cluster management tools are required.

For torchrun, SLURM is the most commonly used tools.

e.g. `srun --nodes=2 ./torchrun_script.sh`

SLURM is Resource Management/Scheduling tools on Linux cluster.

Nvidia provides DeepOps, Infrastructure automation tools for Kubernetes and Slurm clusters with NVIDIA GPUs.