Distributed Training with PyTorch DDP

Author: Sam Foreman (foremans@anl.gov), Corey Adams (corey.adams@anl.gov)

Pytorch has an additional built-in distributed data parallel package, DDP, short for Distributed Data Parallel. It comes in pytorch 1.6 or higher, and wraps your model (**not** your optimizer, like horovod) and performs computation and communication simultaneously.
DDP implements data parallelism at the module level which can run across multiple machines.

- PyTorch Documentation
- Original paper

A thorough description of the design and implementation can be found in the original paper, and there are many great resources available from PyTorch Distributed Overview — PyTorch Tutorials 1.11.0 documentation

> Distributed Data-Parallel Training (DDP) is a widely adopted single-program multiple-data training paradigm. With DDP, the model is replicated on every process, and every model replica will be fed with a different set of input data samples. DDP takes care of gradient communication to keep model replicas synchronized and overlaps it with the gradient computations to speed up training.

Overview

> DistributedDataParallel (DDP) implements data parallelism at the module level which can run across multiple machines. Applications using DDP should spawn multiple processes and create a single DDP instance per process. DDP uses collective communications in the torch.distributed package to synchronize gradients and buffers. More specifically, DDP registers an autograd hook for each parameter given by `model.parameters()` and the hook will fire when the corresponding gradient is computed in the backward pass. Then DDP uses that signal to trigger gradient synchronization across processes. Please refer to DDP design note for more details[1].

How does DDP work?

In short, DDP wraps your model and figures out what tensors need to be allreduced and when. It leverages the situations where, during a sequential backward pass, the earlist tensors to be used going backwards (which are the latest in the model!) are ready for an allreduce operation much sooner than the latest tensors (which are the first in the model!). So, during the backwards pass the allreduce of tensors that are no longer needed in the graph will happen while the backwards pass is still computing.

Additionally, DDP performs tensor fusion to create larger buffers for tensors. Horovod also has the option, though DDP uses it by default.

DDP support

DDP is only available in newer versions of python, and on ThetaGPU works most reliably using Nvidia's docker or singularity containers. The examples here will use these containers.

For collective communication, DDP can use `NCCL` on GPUs, and `gloo` on CPUs.

Setting up DDP

To start DDP, you need to call pytorch's `init_process_group` function from the `distributed` package. This has a few options, but the easist is typically to use the environment variables. You also need to assign each script you have a rank and world size - fortunately, DDP is compatible with MPI so this is no trouble using `mpi4py`.

To set up the environment, do something like this:

```
import os
import sys
```

```
import tempfile
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
import torch.multiprocessing as mp

from torch.nn.parallel import DistributedDataParallel as DDP

# Get MPI:
from mpi4py import MPI

def setup(backend: Optional[str] = 'nccl') -> None:
    os.environ['MASTER_ADDR'] = master_addr
    os.environ['MASTER_PORT'] = master_port
    # initialize the process group
    # Use openmpi environment variables to read the local rank:
    local_rank = os.environ['OMPI_COMM_WORLD_LOCAL_RANK']
    # Use MPI to get the world size and the global rank:
    size = MPI.COMM_WORLD.Get_size()
    rank = MPI.COMM_WORLD.Get_rank()



    # Pytorch will look for these:
    os.environ["RANK"] = str(rank)
    os.environ["WORLD_SIZE"] = str(size)
    os.environ['CUDA_VISIBLE_DEVICES'] = str(local_rank)

    # Get the hostname of the master node, and broadcast it to all other
nodes
    # It will want the master address too, which we'll broadcast:
    if rank == 0:
        master_addr = socket.gethostname()
    else:
        master_addr = None

    master_addr = MPI.COMM_WORLD.bcast(master_addr, root=0)
    # Set the master address on all nodes:
    os.environ["MASTER_ADDR"] = master_addr
```

```
    # Port can be any open port
    os.environ["MASTER_PORT"] = str(2345)
    dist.init_process_group(
        backend,
        rank=rank,
        world_size=world_size,
        init_method='env://'
    )


def cleanup():
    dist.destroy_process_group()
```

After this, you can use the `init_method='env://` argument in init_process.

Other init methods?

Unlike `hvd.init()`, which use MPI and does everything under the hood, DDP is more steps and more options.

Other choices are to pass a common file all processes can write to.

Wrapping your model in DDP:

It's easy:

```
model = DDP(model)
```

You simply replace your model with `DDP(model)` and DDP will synchronize the weights and, during training, all reduce gradients before applying updates. DDP handles all of it for you.

DDP Performance

DDP Generally scales better than horovod for pytorch at large node counts because it can begin all reduce before the backwards pass has finished. On Summit, with 1,536 V100 GPUs, one ALCF model had a scaling efficiency if 97% with DDP compared to ~60% with horovod.

When to use DDP?

It's only available with pytorch. It's only worthwhile with distributed training, and if your model is small than you won't see great scaling efficiency with either DDP or horovod.

Examples

We provide an example illustrating a simple use case of DDP for distributed training on the MNIST dataset.

This example can be found in `./DDP/main.py`

Running in a container

Unfortunately, running in a container to use DDP requires two scripts, not one, to successfully launch the program. The reason for this is that our container does not have `mpi4py` and so we launch singularity with MPI from the main cobalt script, and each singularity instance launches a shell script that itself launches python. To be more clear:

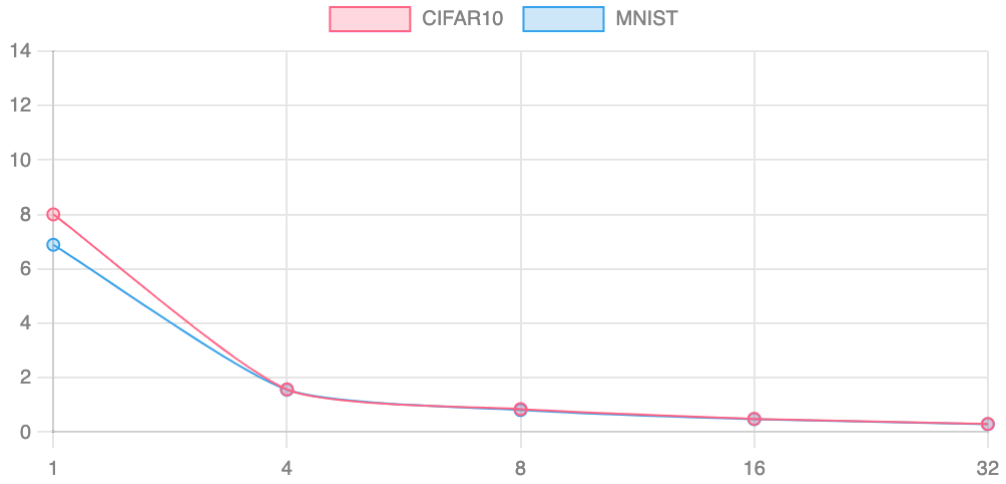
- Cobalt script which runs on one node, and uses `mpirun` to launch `n` singularity instances, typically 8 per node.
  - Each singularity instance launches the same shell script
  - Each shell script sets up a virtual environment, and then executes the main python script.

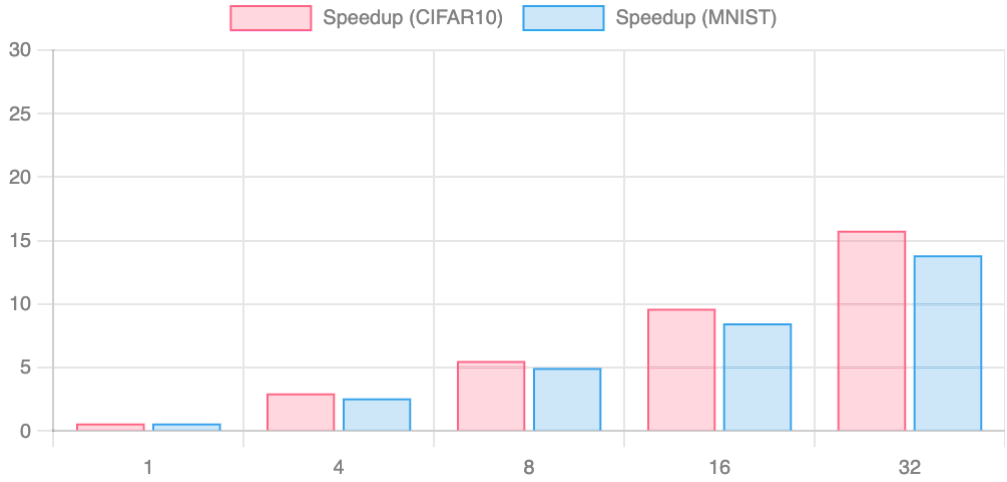
Take a look in the `submissions` folder for more details about this.

Example Performance

Here I show the results I got measuring the time-per-epoch averaged over the last 5 epochs of a training run. I scaled out over a single node, and out onto 4 nodes x 8 GPUs

GPUs	Cifar10 Time/epoch [s]	Speedup	MNIST Time/epoch [s]	Speedup
1	13.6	1.0	11.7	1.0
4	2.66	5.113	2.64	4.4318
8	1.43	9.510	1.37	8.5401
16	0.82	16.585	0.80	14.625
32	0.5	27.2	0.49	23.878





Example

Modified from: Distributed Data Parallel — PyTorch 1.11.0 documentation

> **Warning**
>
> The implementation of `torch.nn.parallel.DistributedDataParallel` is under relatively active development and should be expected to change over time.

References

1. DDP notes offer a starter example and some brief descriptions of its design and implementation. If this is your first time using DDP, start from this document.
2. Getting Started with Distributed Data Parallel explains some common problems with DDP training, including unbalanced workload, checkpointing, and multi-device models. Note that, DDP can be easily combined with single-machine multi-device model parallelism which is described in the Single-Machine Model Parallel Best Practices tutorial.
3. The Launching and configuring distributed data parallel applications document shows how to use the DDP launching script.
4. The Shard Optimizer States With ZeroRedundancyOptimizer recipe demonstrates how ZeroRedundancyOptimizer helps to reduce optimizer memory footprint.
5. The Distributed Training with Uneven Inputs Using the Join Context Manager tutorial walks through using the generic join context for distributed training with uneven inputs.

FSDP (Fully Sharded Data Parallel)

tutorialsFSDP_tutorial.rst at master · pytorchtutorials

---

1. Getting Started with Distributed Data Parallel — PyTorch Tutorials 1.11.0+cu102 documentation ↩