EURO

Multi-GPU and Multi-Node with PyTorch DDP

Georg Zitzlsberger, IT4Innovations, 13-09-2023

Agenda

- Parallelism
- Distributed Training
- Native Data Parallel Training
- PyTorch DDP
- Hands-On: PyTorch DDP

Parallelism

Parallelism can be found in:

- Low level operations in the network
- Parallelized networks
- Distributed training

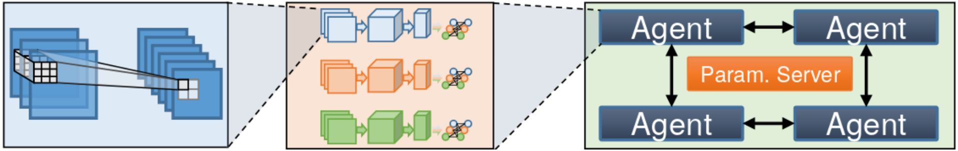
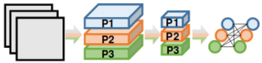
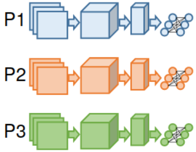


Image: Ben-Nun, et al.

Difference Data vs. Model Parallelism



- Network layers assigned to different workers
- Every worker trains with the same data
- Activations are exchanged (requires large I/O bandwidth)
- **Enables bigger models**



- All workers see the same network
- Every worker trians with different data
- Gradints (weights) are exchanged (averaging to common model)
- Side effect: "sharp" minima
- **Enables faster training**

Images: Ben-Nun, et al.

Distributed Training

What if one node (GPU) is not enough?

- Model Consistency
- Parameter Distribution and Communication:
  Centralization and Compression (e.g. FP16)
- Training Distribution:
  Model Consolidation and Optimization Algorithms

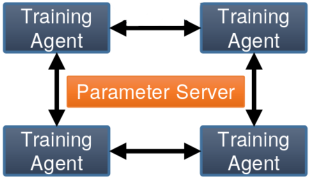
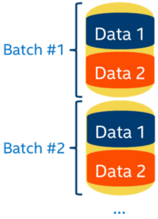
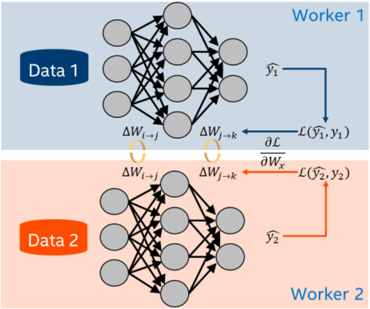


Image: Ben-Nun, et al.

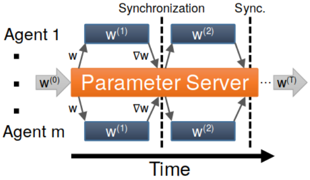
Distributed Training: Data Parallelism in Detail



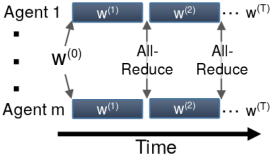


- Batch size limits parallelism
- Scaling batch size requires scaling of learning rate (linearly)

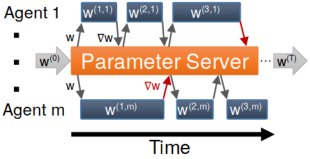
Distributed Training: Model Consistency



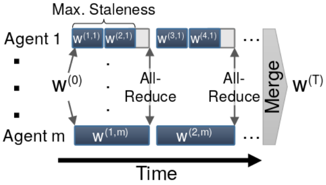
(a) Synchronous, Parameter Server



(b) Synchronous, Decentralized



(c) Asynchronous, Parameter Server



(d) Stale-Synchronous, Decentralized

Image: Ben-Nun, et al.

Distributed Training: Libraries

Different backends:

- Message Passing Interface (MPI)
- NVIDIA Collective Communications Library (NCCL)
- Intel oneAPI Collective Communications Library (oneCCL) ▸ Intel oneCCL
  ~~Intel Machine Learning Scaling Library~~ ▸ Intel MLSL

Strategies vary among frameworks:

- ▸ Horovod for Tensorflow/Keras, PyTorch and MXNet (NCCL + MPI, or Gloo)
- Tensorflow has different "strategies" (e.g. `MultiWorkerMirroredStrategy`)
- PyTorch supports MPI, NCCL and Gloo

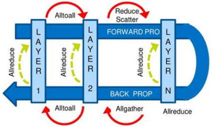


Image: Intel (MLSL)

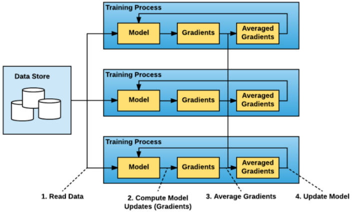


Image: Uber

NVIDIA Collective Communications Library

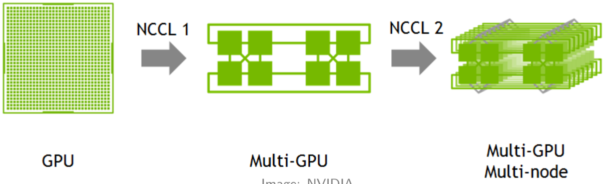


Image: NVIDIA

NVIDIA Collective Communications Library (NCCL):

- Similarity to MPI collectives
- Pre NCCL 2.4: ring communication
- NCCL 2.4: double binary tree communication with improved latency
- Supported by major Deep Learning frameworks, e.g. Tensorflow, PyTorch
- Supported by distributed training framework Horovod

Native Data Parallel Training

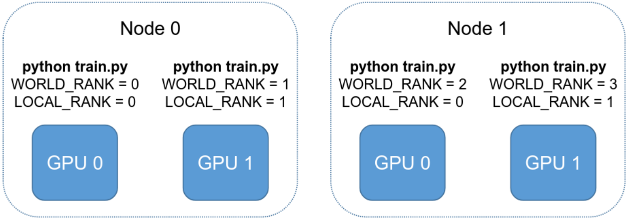
One of the first ways was Uber's Horovod:

- Aimed at and demonstrated for large scale
- Uses MPI based collective communication (synchronous & decentralized)
- Only small code modifications needed
- Supports Tensorflow (1.x & 2.x) + Keras, PyTorch, and MXNet

Native support of (sync.) data parallel training is available meanwhile:

- Tensorflow:
  `tf.distribute.Strategy` with different strategies (MirroredStrategy, TPUStrategy, MultiWorkerMirroredStrategy, CentralStorageStrategy, ParameterServerStrategy)
  ▸ Documentation
- PyTorch:
  `torch.distributed` with three backends (GLOO, MPI, NCCL) ▸ Documentation
  Use ▸ `torch.nn.parallel.DistributedDataParallel` (DDP) with **NCCL** backend for multi-node/-GPU.

PyTorch DDP



1. Initialize:

```
import os
import torch
import torch.distributed as dist

LOCAL_RANK = int(os.environ['OMPI_COMM_WORLD_LOCAL_RANK'])
WORLD_SIZE = int(os.environ['OMPI_COMM_WORLD_SIZE'])
WORLD_RANK = int(os.environ['OMPI_COMM_WORLD_RANK'])

os.environ["MASTER_ADDR"] = "acn1" # Master node (rank 0 process)
os.environ["MASTER_PORT"] = "6006" # Some free port on master node

dist.init_process_group(backend="nccl", rank=WORLD_RANK, world_size=WORLD_SIZE)
```

PyTorch DDP - cont'd

2. Assign to GPU(s)

```
device = torch.device("cuda:" + str(LOCAL_RANK))
model = my_model.to(device)
```

3. Make model aware of DDP

```
ddp_model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[LOCAL_RANK], output_device=LOCAL_RANK)
```

4. Data partitioning

```
train_sampler = torch.utils.data.distributed.DistributedSampler(train_set, num_replicas=WORLD_SIZE, rank=WORLD_RANK)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, sampler=train_sampler)
```

5. I/O on master only

```
if WORLD_RANK == 0: # for shared file system
    mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, transform=None)
...
if WORLD_RANK == 0:
    print("Epoch␣=␣{:2d}:␣Validation␣Loss␣=␣{:5.3f},␣Validation␣Accuracy␣=␣{:5.3f}".format(epoch+1, v_loss, val_accuracy))
...
if WORLD_RANK == 0:
    torch.save(model.state_dict(), 'model.pt')
```

PyTorch DDP - cont'd

Run in an MPI environment:

- OpenMPI:

```
$ mpirun -np 4 --map-by ppr:2:node --host node1,node2 python train.py
```

- Intel MPI:

```
$ mpirun -n 4 -np 2 -hosts node1,node2 python train.py
```

Note:
This runs four processes, two on each node, each executing the **same** `train.py` script!

Difference PyTorch and Tensorflow/Horovod

- Tensorflow/Horovod:
  - Data parallelization is done in optimizer
  - Decomposition of data is done with `tf.data.Dataset.shard`
- PyTorch:
  - Data parallelization is done in model:

```
import os
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
...
num_gpus = int(os.environ['OMPI_COMM_WORLD_SIZE'])
rank = int(os.environ['OMPI_COMM_WORLD_RANK'])

dist.init_process_group("nccl", rank=rank, world_size=num_gpus)

model = Model().to(rank) # Move to GPU
ddp_model = DDP(model, device_ids=[rank])
```

  - Data decomposition with `torch.utils.data.distributed.DistributedSampler`, e.g.:

```
from torch.utils.data import DataLoader
from torch.utils.data.distributed import DistributedSampler
...
sampler = DistributedSampler(dataset, num_replicas=num_gpus, rank=rank)
loader = DataLoader(dataset, sampler=sampler)
```

Hands-On Time

Thank you!

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101101903. The JU receives support from the Digital Europe Programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Türkiye, Republic of North Macedonia, Iceland, Montenegro, Serbia.