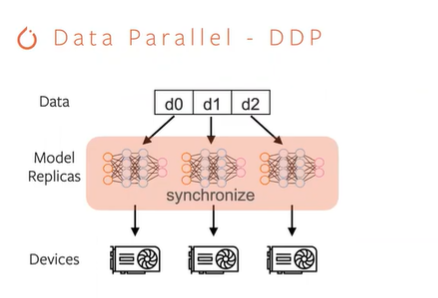
Pytorch Distributed

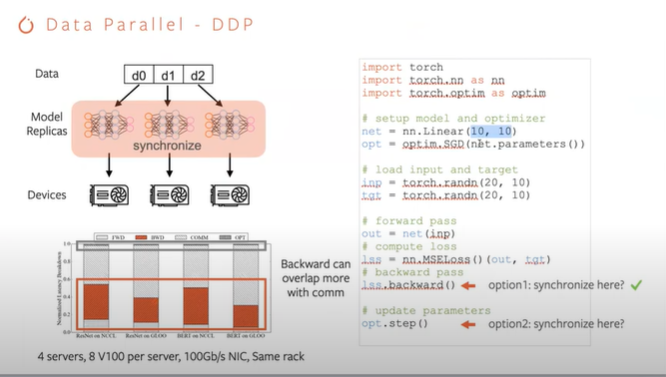
Distributed Data Parallel – DPP



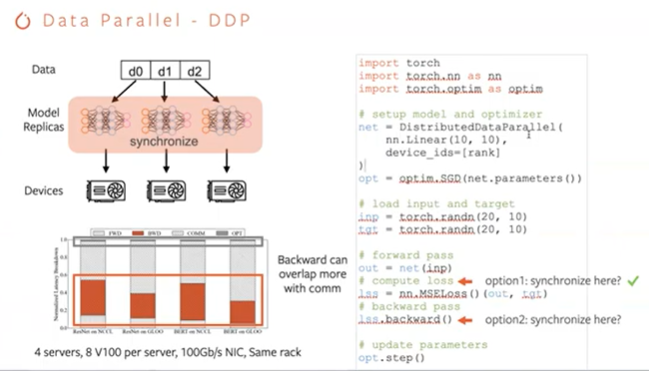
How synchronize model replica?

Une image contenant texte

Description générée automatiquement



Backward pass is more expensive than forward



Pytorch distributed automatically or do we have control as a user ?

DDP contains an argument called bucket size – so it’s configurable

Not possible for some uses case that the bucket 1 is not the same size as the bucket 2 ?

Can we a bunch of different bucket size ?

Not possible with DDP API

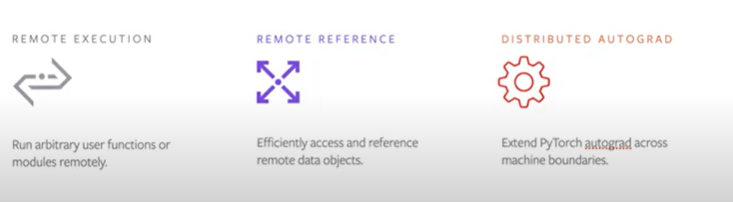
Bucketization effect or Synchronization problem ?

Drawbacks of synchronized training: Using more GPUs -> structure become more a problem

Struggler problem when they are using infinite training

Asynchronous training -> industry doesn’t have a good solution to always guarantee to give the same model accuracy level -> That’s why they still use synchronize training

## Model Parallel – RPC

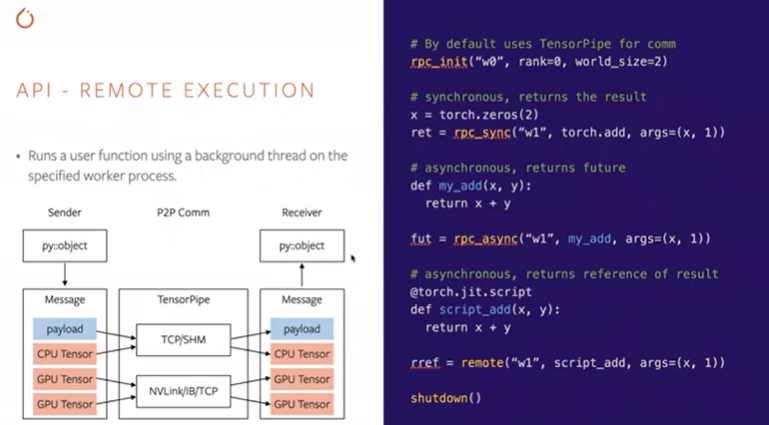


RPC try to solve problem with 3 features

**Remote Execution :** run arbitrary user functions or modules remotely

**Remote Reference:** access, reference & share a remote data object without fetching back the actual data in that object

**Distributed Autograd:** autograd engine is a per process concept – stitch together the local autograd & coordinate multiple autograd engines to collectively produce the gradient for the parameters during the backward pass



Rpc\_init(name current process, rank = ID, world\_size = nb of processes in this RPC game)

* Create agent running in background - When agent is ready – can receive & process requests from other peers
* After RPC unit returns, it means all peers in this RPC band is ready – start sending requests to any peer in this game

We don’t have like a server client

**3 ways to do communication in RPC**

* rpc\_sync(name of destination process, function that you want to run on destination, list of argument you want to provide it) – going to block until the response is fetched to the caller
* rpc\_async() – only difference with sync is that API returns a future immediately & can read on the future to get the result – it will guarantee to be flashed to the caller in some future time
* remote() – returns immediately, difference with async -> remote returns a remote reference

Why use Pytorch RPC & not use Local pytorch connect to GRPC & the script to the communication ?

We can do that but Pytorch RPC understands tensors

Pytorch local training with GRBC -> data needs to be serialized into one message before they are sent out

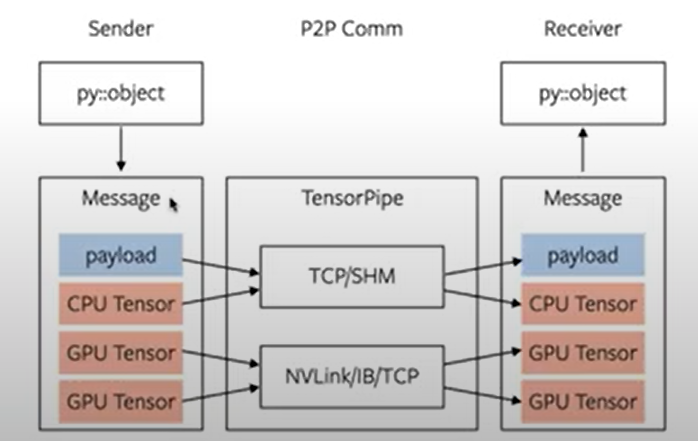
When use RPC system ?

During forward pass to send activation

During backward pass to send the gradients

The arguments can be very large activation tensors, gradients tensors

If we have to serialize & digitalize those large tensors then the serialization can be a bottleneck in the system



Pytorch RPC will pass a python object to RPC system

RPC system going to serialize that object in the message however you are going to extract out all the tensors & keeping those tensors intact

Message format of the RC system is that there will be a vector of tensors

All tensors are not touched the storage are keep intact – everything (argument, meta information, function name,…) will be packed into a binary payload

Why tensor pipe ? Can MPI be the backend for RPC system ?

Yes it can be but the difference Tensor Pipe & MPI is that tensor pipe will automatically try to pick the best channel for us

When you call the RPC image they are argument to specify what are the available hardware

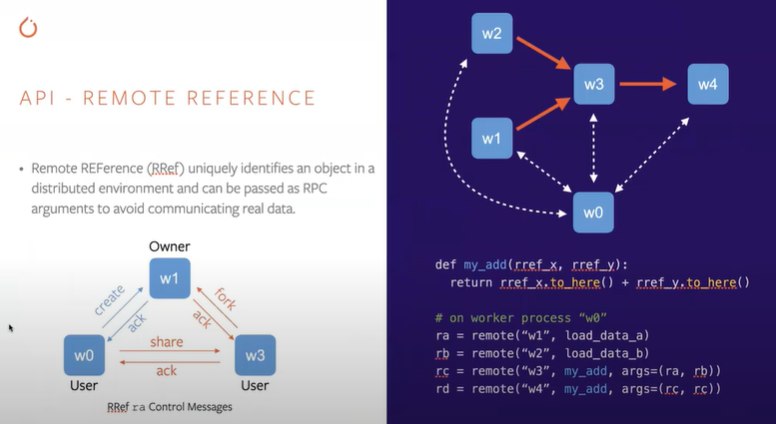
Tensor Pipe will pick the best communication meter

Situation when data movement between CPU & GPU became a bottleneck instead of just moving data from one server to another ? (copy data from one node to another then you have to copy data from gpu to a cpu memory, then copy cpu data from cpu memory to another node of cpu memory the move to the GPU)

GPU trainers & CPU parameters servers – doing gpu to gpu communication & then on a parameter server side

Moving tensor from GPU to CPU on the trainer & then do the CPU to TCP communication

CPU offloading – requires frequent GPU to CPU communication



During training, common that we have 1 master node or 1 driver node that drives the entire execution of training loop

In this example :

w0 as the driver

Other peers are the ones that does the real work

Define a function “my\_add” takes the 2 reference remote references & fetch data to itself & add them together

On worker process “w0” 4 lines are caughts on w0

We first cut load data A on w1, then leave data there only hold on to the reference of that data

Then do similar thing on w2

W0 has the reference align with reference to those data

Then w0 calls another remote to w3 & ask w3 to add those data together

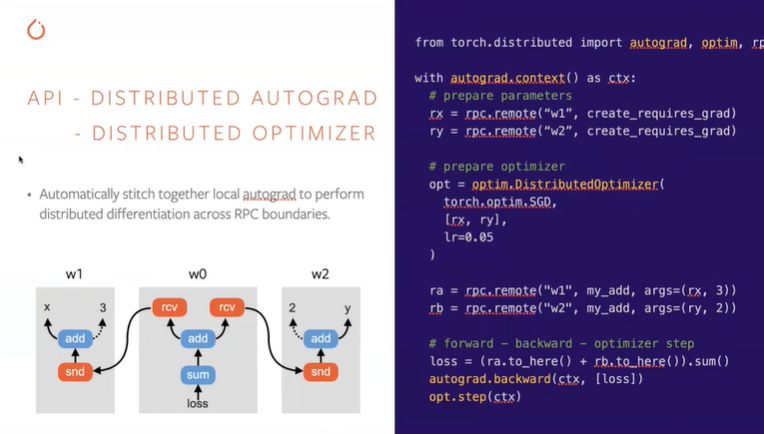
All the code are running on w0

W0 control all the workers -> very flexible for Federate Learning

Basically, you can define the reference of whatever factions you want & can pass them around the flow to the workers & we materialize at specifics words that we want

W0 is like the manager to control the entire network to access different tasks

We can define the whole operation that we want to compute as references & them we materialize them on the specific workers that we need



When we call RPC function we insert a send function, send autograd on the caller & then insert a corresponding receive autograd function

During backward pass on the callee & local autograd engine on the calle will eventually reach the receive function – will be responsible for sending a gradient over to its across one instance function

First we create a context for autograd

Then we prepare the parameters & distributed optimizers

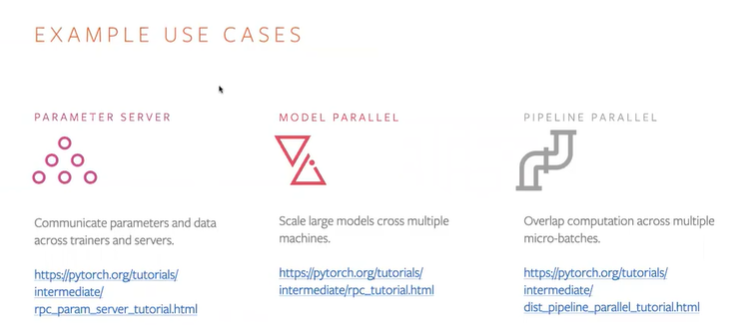
Provide a thin wrapper a distributed option optimizer

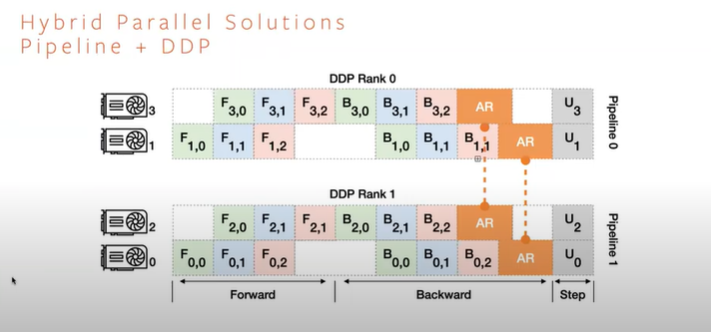
Going to take remote references

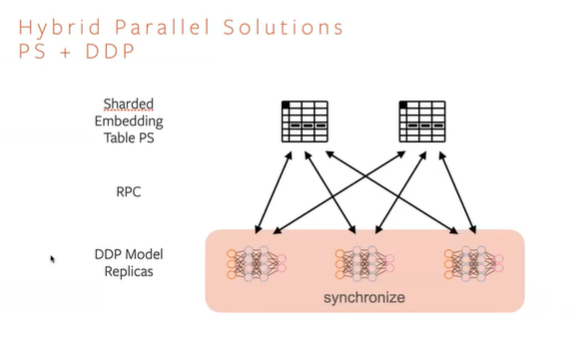
* optim.DistributedOptimizer -> will reach out to all the distinct owners of those remote references & then create a local optimizer on those owners
* It will create a SGD on W1 for x & W2 for y

Ra & rb result of those 2 additions

Then we call to.here() on ra & rb going to incur RPC communication & then you are gonna insert a stand on a caller then I receive on the callee







RPC Backend : TensorPipe

<https://pytorch.org/docs/master/rpc.html#tensorpipe-backend>

TensorPipe :P2P comm library & designed to automatically figure out comm between RPC workers (TCP, nvlink, ib)

Natively Asynchronous

Mainly Smart payload delivery layer (there are unoptimized RPC libraries – GRPC, Dubbo, Swift)

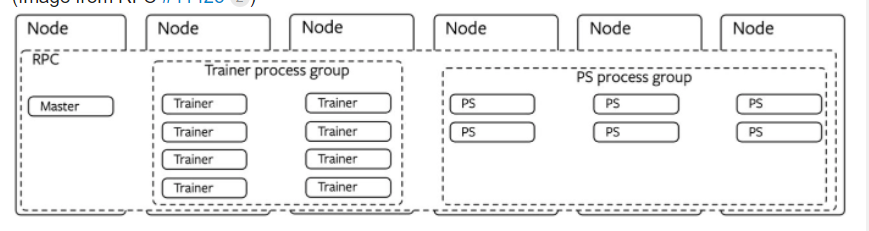
**Heterogeneous Torch RPC Apps**

Master Node (running a single master process)

2 Trainer Nodes (each trainer workers)

3 parameter server nodes (each 2 parameter server workers)

RFC : automatic role-based launcher implementation



https://discuss.pytorch.org/t/distributed-model-parallel-using-distributed-rpc/87875/20

<https://github.com/pytorch/pytorch/issues/41425>

Documentation :

https://www.youtube.com/watch?v=3XUG7cjte2U&ab\_channel=ChaoyangHe