Benchmark combining Distributed Data Parallel and Distributed RPC

This Benchmark is used to measure distributed training iteration time. It combines Distributed Data Parallelism with Distributed Model Parallelism leveraging PyTorch DDP and the Distributed RPC Framework. The number of trainer nodes and parameter servers are configurable. The default is 8 trainers, 1 master node and 8 parameter servers.

Background

There are different training paradigms where combining these two techniques might be useful. For example: 1) If we have a model with a sparse part (large embedding table) and a dense part (FC layers), we might want to set the embedding table on a parameter server and replicate the FC layer across multiple trainers using DistributedDataParallel. The Distributed RPC framework comes handy to perform embedding lookups on the parameter servers. 2) Enable hybrid parallelism as described in the PipeDream paper. We can use the Distributed RPC framework to pipeline stages of the model across multiple workers and replicate each stage (if needed) using DistributedDataParallel.

Training Process

This benchmark focuses on the first paradime above. The training process is executed as follows:

1) The master creates embedding tables on each of the 8 Parameter Servers and holds an RRef to it.
2) The master, then kicks off the training loop on the 8 trainers and passes the embedding table RRef to the trainers.
3) The trainers create a `HybridModel` which performs embedding lookups in all 8 Parameter Servers using the embedding table RRef provided by the master and then executes the FC layer which is wrapped and replicated via DDP (DistributedDataParallel).
4) The trainer executes the forward pass of the model and uses the loss to execute the backward pass using Distributed Autograd.
5) As part of the backward pass, the gradients for the FC layer are computed first and synced to all trainers via allreduce in DDP.
6) Next, Distributed Autograd propagates the gradients to the parameter servers, where the gradients for the embedding table are updated.
7) Finally, the Distributed Optimizer is used to update all parameters.

Example Benchmark output:

———- Info ———

- PyTorch version: 1.7.0

- CUDA version: 9.2.0

——— nvidia-smi topo -m ———

GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	CPU	Affinity
GPU0	X	NV2	NV1	NV2	NV1	NODE	NODE	NODE	0-19,40-59
GPU1	NV2	X	NV2	NV1	NODE	NV1	NODE	NODE	0-19,40-59
GPU2	NV1	NV2	X	NV1	NODE	NODE	NV2	NODE	0-19,40-59
GPU3	NV2	NV1	NV1	X	NODE	NODE	NODE	NV2	0-19,40-59
GPU4	NV1	NODE	NODE	NODE	X	NV2	NV1	NV2	0-19,40-59
GPU5	NODE	NV1	NODE	NODE	NV2	X	NV2	NV1	0-19,40-59
GPU6	NODE	NODE	NV2	NODE	NV1	NV2	X	NV1	0-19,40-59
GPU7	NODE	NODE	NODE	NV2	NV2	NV1	NV1	X	0-19,40-59

Legend:

X = Self SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI) NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU) PXB = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge) PIX = Connection traversing a single PCIe switch NV# = Connection traversing a bonded set of # NVLinks

——— PyTorch Distributed Benchmark (DDP and RPC) ———

		sec/epoch	epoch/sec		sec/epoch	epoch/sec		sec/epoch	epoch/sec		sec/
Trainer0:	p50:	0.376s	185/s	p75:	0.384s	182/s	p90:	0.390s	179/s	p95:	0.3
Trainer1:	p50:	0.377s	204/s	p75:	0.384s	200/s	p90:	0.389s	197/s	p95:	0.3
Trainer2:	p50:	0.377s	175/s	p75:	0.384s	172/s	p90:	0.390s	169/s	p95:	0.3
Trainer3:	p50:	0.377s	161/s	p75:	0.384s	158/s	p90:	0.390s	156/s	p95:	0.3
Trainer4:	p50:	0.377s	172/s	p75:	0.383s	169/s	p90:	0.389s	166/s	p95:	0.3
Trainer5:	p50:	0.377s	180/s	p75:	0.383s	177/s	p90:	0.389s	174/s	p95:	0.3
Trainer6:	p50:	0.377s	204/s	p75:	0.384s	200/s	p90:	0.390s	197/s	p95:	0.3
Trainer7:	p50:	0.377s	185/s	p75:	0.384s	182/s	p90:	0.389s	179/s	p95:	0.3
All:	p50:	0.377s	1470/s	p75:	0.384s	1443/s	p90:	0.390s	1421/s	p95:	0.3