Distributed Deep Learning

Huihuo Zheng

Data Science Team
Argonne Leadership Computing Facility
Aug 11, 2023

huihuo.zheng@anl.gov
Outline

• The need for distributed training
• State-of-the-art parallelization schemes
• Data parallel training in details
• I/O and data management in distributed training
• Hands on
The need for distributed training on HPC

“Since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.5 month doubling time (by comparison, Moore’s Law had an 18 month doubling period).”

https://openai.com/blog/ai-and-compute/

Large language model: # parameters grows by about 10x every year
Distributed deep learning for ResNet-50

<table>
<thead>
<tr>
<th>Training time and Top-1 validation accuracy with ResNet-50 on ImageNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Size</td>
</tr>
<tr>
<td>He et al. [1]</td>
</tr>
<tr>
<td>Goyal et al. [2]</td>
</tr>
<tr>
<td>Smith et al. [3]</td>
</tr>
<tr>
<td>Akiba et al. [4]</td>
</tr>
<tr>
<td>Jia et al. [5]</td>
</tr>
<tr>
<td>Ying et al. [6]</td>
</tr>
<tr>
<td>Mikami et al. [7]</td>
</tr>
<tr>
<td>Yamazaki et al</td>
</tr>
</tbody>
</table>

Quoted from Masafumi Yamazaki, arXiv:1903.12650
Training Large Natural Language Model is expensive

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Number of parameters (billion)</th>
<th>Model-parallel size</th>
<th>Batch size</th>
<th>Number of GPUs</th>
<th>Microbatch size</th>
<th>Achieved teraFLOP/s per GPU</th>
<th>Training time for 300B tokens (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeRO-3 without Model Parallelism</td>
<td>174.6</td>
<td>1</td>
<td>1536</td>
<td>384</td>
<td>4</td>
<td>144</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>768</td>
<td>2</td>
<td>88</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1536</td>
<td>1</td>
<td>44</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>529.6</td>
<td>1</td>
<td>2560*</td>
<td>640</td>
<td>4</td>
<td>138</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1120</td>
<td>2</td>
<td>98</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2240</td>
<td>1</td>
<td>48</td>
<td>140</td>
</tr>
<tr>
<td>PTD Parallelism</td>
<td>174.6</td>
<td>96</td>
<td>1536</td>
<td>384</td>
<td>1</td>
<td>153</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>768</td>
<td>1</td>
<td>149</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1536</td>
<td>1</td>
<td>141</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>529.6</td>
<td>280</td>
<td>2240</td>
<td>560</td>
<td>1</td>
<td>171</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1120</td>
<td>1</td>
<td>167</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2240</td>
<td>1</td>
<td>159</td>
<td>42</td>
</tr>
</tbody>
</table>

The need for distributed training

- Increase of model complexity leads to dramatic increase of the amount of computation;
- Increase of the size of dataset makes sequentially scanning the whole dataset increasingly impossible;
- Coupling of deep learning to traditional HPC simulations might require distributed training and inference.

Examples of scientific large scale deep learning
- Thorsten Kurth, Exascale Deep Learning for Climate Analytics, arXiv:1810.01993 (Gordon Bell Prize)
- W. Dong et al, Scaling Distributed Training of Flood-Filling Networks on HPC Infrastructure for Brain Mapping, arXiv:1905.06236
Parallelization schemes – Model Parallelism (MP)

```
import torch
import torch.nn as nn
import torch.optim as optim

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

    def forward(self, x):
        x = self.relu(self.net1(x.to('cuda:0')))
        return self.net2(x.to('cuda:1'))

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

optimizer.zero_grad()
outputs = model(torch.randn(20, 10))
labels = torch.randn(20, 5).to('cuda:1')
loss_fn(outputs, labels).backward()
optimizer.step()
```

PyTorch multiple GPU model parallelism within a node

https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html
Parallelization schemes – Pipeline parallelism (PP)

- Partition model layers into multiple groups (stages) and place them on a set of inter-connected devices.
- Each input batch is further divided into multiple micro-batches, which are scheduled to run over multiple devices in a pipelined manner.

**Pipeline libraries:**
- Pipe-torch:
  DOI: 10.1109/CBD.2019.00020
- PipeDream: arXiv:1806.03377
- PyTorch Distributed RPC Frameworks: [https://pytorch.org/tutorials/intermediate/dist_pipeline_parallel_tutorial.html](https://pytorch.org/tutorials/intermediate/dist_pipeline_parallel_tutorial.html)
- DeepSpeed: [https://github.com/microsoft/DeepSpeed](https://github.com/microsoft/DeepSpeed)
Parallelization schemes – Data Parallelism (DP)

**Data parallelism**

Worker 1 → Worker 4 → Worker N

- Model is replicated on each worker
- Each worker processes a subset of the minibatch
- Sync up the weights before updating the model

Popular frameworks support DP

- DDP
- HOROVOD
- DeepSpeed

https://leimao.github.io/blog/PyTorch-Distributed-Training/
https://github.com/horovod/horovod
https://github.com/microsoft/DeepSpeed
Data parallel training in details
Linear rate scaling rule

When the minibatch size is multiplied by $k$, multiply the learning rate by $k$.

Mini-batch stochastic Gradient Descent

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B} \nabla l(x, w_t)$$

Learning rate, $lr$  Mini-batch

Typical practice / suggestion:
- keep local batch size per worker, i.e., increase the global batch size linearly
- Increase the learning rate proportionally

Actuality: need to adjust the batch size and learning rate at different scale

$$lr_{scale} = lr \times nprocs$$
Potential issues for large batch size and learning rate

Minibatch Size Effect on Accuracy and Performance

Tal Ben-Nun and Torsten Hoefler, arXiv:1802.09941

Validation error for different mini-batch size for Resnet50


- Stability of optimization – can be solved by learning rate warming up
- Generation gap – systematic issue; has to reduce the learning rate

Generation gap: Keskar et al, arXiv:1609.04836
Data parallel training with Horovod

https://eng.uber.com/horovod/
Data parallel training with Horovod

Steps to parallelize a series code:

• Import Horovod modules and initialize horovod
• Scale the learning rate by number of workers
• Wrap optimizer in hvd.DistributedOptimizer
• Broadcast the weights from worker 0 to all the workers
• Worker 0 saves the check point files
• Dataset sharding: make sure different workers load different samples.

Instruction on how to change the code is here
Tensorflow: 04_keras_cnn_verbose_hvd.py
Pytorch: 04_pytorch_cnn_hvd.py

https://eng.uber.com/horovod/
Scaling TensorFlow using Data parallelism on Theta @ ALCF: fixing local batch size = 512

![Graphs showing performance of AlexNet, ResNet-50, and Inception V3 with varying number of KNL nodes.](image)

- **AlexNet**
  - Ideal performance
  - Horovod performance

- **ResNet-50**
  - Ideal performance
  - Horovod performance

- **Inception V3**
  - Ideal performance
  - Horovod performance
Different frameworks show similar scaling efficiency

Horovod, DDP, and DeepSpeed show similar performance for three PyTorch models. Evaluation were done on Polaris @ ALCF

Zhenhao Z. @IIT

https://github.com/argonne-lcf/dl_scaling.git
Data Management and I/O for AI


DLIO Benchmark: https://github.com/argonne-lcf/dlio_benchmark.git

Deep Learning I/O characteristics

Typical process of AI training. The dataset is loaded from the storage to the host RAM and then feed into the accelerators for training.

Characteristics of I/O for AI applications

- Read intensive
- Metadata intensive
- Small and sparse I/O operations
- Random access
- Complex data format (json, text, key-value store)
- Utilizing storage hierarchy
- Multithreading background I/O
I/O bottleneck for UNet3D workload on fast accelerator

Timeline tracing for training the UNet3D workload on a single GPU on JLSE with GPFS file system. This shows that I/O become a bottleneck for faster accelerator

UNet3D Model: https://github.com/mlcommons/training/tree/master/image_segmentation/pytorch
Scaling bottleneck from IO for UNet3D workload (simulated using DLIO Benchmark)

UNet3D weak scaling

AU = \frac{\text{compute\_time}}{\text{total\_time}}

I/O throughput

Throughput = \frac{\text{data\_size}}{\text{time}}

Accelerator utilization (AU) and I/O throughput at different scale on Polaris for UNet3D model, with Lustre file system and NVMe -> staging helps

https://github.com/argonne-lcf/dlio_benchmark.git
Ideal scaling for less I/O intensive workload – Bert (simulated using DLIO Benchmark)

Accelerator utilization (AU) and I/O throughput at different scale on Polaris for Bert model, with the Lustre file system

https://github.com/argonne-lcf/dlio_benchmark.git
Tips for I/O and data management

• Preprocess the raw data (resize, interpolation, etc) into binary format before the training;
• Store the dataset in a reasonable way (file per sample, single shared file, or multiple samples per file)
• Optimal setting (Lustre stripe count, size)
• Remember to shard the dataset;
• Prefetch and caching the data (from disk; from host to device; staging to NVMe, SSDs);
• Use more I/O workers to load data concurrently (e.g., adjust num_workers in TorchDataLoader)

Streaming I/O using Data Loader
• TensorFlow Data Pipeline
• PyTorch Data Loader
• Nvidia Dali Data Loader
Main take aways

• Distributed training can be done through model parallelism and Data parallelism.

• Data parallelism frameworks: Horovod, DDP (PyTorch only), DeepSpeed (PyTorch only) → similar performance.

• Efficient data management and I/O is crucial for data intensive training
Hands on

$ git clone git@github.com:argonne-lcf/ATPESC_MachineLearning.git
$ cd ATPESC_MachineLearning/04_distributedLearning
$ qsub submissions/qsub_thetagpu.sc or qsub submissions/qsub_polaris.sc
Thank you!

huihuo.zheng@anl.gov