HPC for Training Large Scale Distributed Deep Networks

Mohammadreza Bayatpour
bayatpour.1@osu.edu
High Performance DL Course
Fall 2018

HiDL Course Fall 2018
Paper #1

• Deep Learning at 15PF: Supervised and Semi-Supervised Classification for Scientific Data

• **Purpose:**

• *Scaling the training of a single model to ~9600 Xeon-Phi nodes while obtaining near peak performance*
Deep Learning For Science

- Deep Learning (DL) has enabled fundamental breakthroughs
  - Computer vision
  - Speech recognition
  - Control system problems, etc.
Deep Learning For Science

- Contemporary scientific datasets
  1. Are TBs-PBs in size
     - Training might take days to converge on $O(10)$ GB datasets
  2. Contain dozens of channels/variables
     - In contrast to the small number of channels in images or audio data

*Scientists need to be able to leverage parallel computational resources to train DNNs faster*
Deep Learning on Single Node

• Well optimized libraries exist for dense linear algebra
  • BLAS
  • LaPACK
  • cuDNN for GPU
  • Intel MKL for CPU
• Hardware efficiency depends on
  • Input data sizes
  • Model parameters
• Some kernels can reach 75-80% of peak flops whiles others are around 20-30%
  • Results from DeepBench (Baidu)
Deep Learning on Synchronous-parallel Architectures

• Using synchronization barriers
  • To force computational nodes to perform every update step in lock-step

• Using data parallelism
  • Different nodes split a big mini-batch of samples
  • Each processing a chunk of the data
Limiting Factor of Synchronous Scaling

• Batch size
  • Most systems use some variant of SGD with batch sizes that range from 64 to 1024
  • Large batch sizes have been shown to cause slowdown in convergence

• Stragglers
  • Because of synchronization barriers, the duration of the iteration depends on the slowest node
  • Leading to load imbalance
Multi-node Scaling with Synchronous Approach

• Utilizing the Intel Machine Learning Scalability Library (MLSL)
  • Handles all communication required to perform training in a synchronous setting
  • Enables different forms of parallelism
    • Both data and model parallelism
  • MLSL also introduces performance improvements over vanilla MPI implementations using endpoints - proxy threads/processes
Deep Learning on Asynchronous Architectures

• Each node works on its own iteration (mini-batch) and produces independent updates to the model
• Updates are sent to a central parameter store, the parameter server (PS)
Deep Learning on Asynchronous Architectures (cont’d)

• The PS applies the updates to the model in the order they are received, and sends back the updated model to the worker

• Asynchronous methods are known to give significant computational benefits in large-scale systems
  • They do not suffer from straggler effects and are not limited by the total batch size in the same way that synchronous systems are
Performance tradeoff between Sync and Async Architectures

• Main side-effect of asynchrony is the use of out-of-date gradients
• Asynchronous systems may need more iterations to solution, due to staleness
  • Staleness: number of updates that other workers perform between the time a worker reads the model and the time it sends its own update to the PS
  • Async designs have worse statistical efficiency
• Synchronous systems typically take longer per iteration due to the straggler effect
  • Sync designs have worse hardware efficiency
Hybrid Architecture

• The trade-off between statistical efficiency vs. hardware efficiency suggests a third kind of architecture: a *hybrid* system
• Worker nodes coalesce into separate *compute groups*
• Each compute group follows a synchronous architecture
• The workers split a mini-batch among themselves and produce a single update to the model
• There is no synchronization across compute groups
• A parameter server (PS) holds the model and each compute group communicates its updates to the PS asynchronously.
• The size of each group is tunable
Multi-node Scaling with Hybrid Approach

- Nodes are organized into compute groups
- Parallelization is synchronous within (using all-reduce), but asynchronous across groups via a set of parameter servers
- All-reduce operations are used to get the aggregate model update from all workers in the group
- Then a single node per group, called the root node is responsible for communicating the update to the parameter servers, receiving the new model, and broadcasting it back to the group
Optimizations to the Hybrid Approach

- Extended MLSL to enable hybrid implementation
  - To facilitate node placement into disjoint communication groups and dedicating nodes as parameter servers
Optimizations to the Hybrid Approach (cont’d)

- Dedicated parameter servers for each layer
  - Parameter server needs to be able to handle the volume of network traffic and computation for the updates originating from multiple compute groups and for very large models
  - Reduce the chances of PS saturation
Overview of the Proposed Designs

SYNCHRONOUS

ASYNC. PS

ASYNCHRONOUS

Compute group 1

ASYNC. PS

HYBRID

Compute group G
Scientific Application #1: HEP

• HEP: High-Energy Physics (HEP)
  • Is used to find rare signals of new particles produced at accelerators such as the Large Hadron Collider (LHC) at CERN

• Data from the surface of the cylindrical detector can be represented as a sparse 2D image, with data from different layers of instrumentation as channels in that image
Scientific Application #1: HEP (cont’d)

• Simulations of the underlying physics processes and the detector response
  • These can be used for training networks
• Three channels
  • “Electromagnetic”, and “Hadronic calorimeters”, and the number of “Tracks” as three channels
Scientific Application #2: Climate

• Climate: A Climate Simulator
  • Provides understanding the future impact of various carbon emission scenarios and intervention strategies

• Modern Climate simulation codes produce massive datasets

• Purpose is to find extreme weather events in such large datasets

• Images have 16 or more ’channels
  • Consequently, we cannot leverage pre-trained weights from contemporary networks such as VGG or AlexNet
Experimental Setup

• Cori Phase II system at NERSC

• A Cray XC40 supercomputer comprised of 9,688 self-hosted Intel Xeon Phi TM 7250 (Knight’s Landing, KNL) compute nodes
  • Each KNL processor includes 68 cores running at 1.4GHz and capable of hosting 4 Hyper Threads for a total of 272 threads per node.

• The peak performance for single precision can be computed as: (9688 KNLs) x (68 Cores) x (1.4 GHz Clock Speed) x (64 FLOPs / Cycle) = 59 PetaFLOP/s
• The overall flop rate:
  • HEP network stands at 1.90 TFLOP/s
  • Climate network stands at 2.09 TFLOP/s
• Most of the runtime is spent in convolutional layers
Multi-Node Experiments

• **Strong scaling**
  • Keeping the overall batch size per update step fixed while varying the number of nodes

• **Weak scaling**
  • Keeping a constant batch size per node across all the configuration

• Testing three configurations:
  • 1 synchronous group
  • 2 hybrid groups
  • 4 hybrid groups
Multi-Node Experiments Specifications

• Showing scalability from 1 to 1024 nodes

• In strong scaling tests
  • Batch size of 2048 per update
  • For the synchronous configuration
    • All nodes split the batch of 2048 images
  • For hybrid configurations
    • Each compute group independently updates the model
    • Each compute group is assigned a complete batch

• In weak scaling tests
  • All configurations have the batch size of 8 per node
• Synchronous algorithm does not scale past 256 nodes
• The scalability improves moderately for Hybrid designs
  • With 2 hybrid groups performance saturates at 280x beyond 512 nodes
  • With 4 hybrid groups performance saturates at 580x scaling at 1024 nodes
Weak Scaling

- **HEP**
  - Performance scales sub-linearly for all configurations
  - About 575-750x speed-up on 1024 nodes and 1150-1250x speed-up on 2048 nodes for asynchronous
  - Synchronous speed-up on 2048 nodes stands at about 1500x
- **Climate**
  - Near-linear scaling (1750x for synchronous and about 1850x for hybrid configurations)
  - Significant variability in runtime across iterations for HEP at scale, leading to sublinear scaling
Paper #2

• Large Scale Distributed Deep Networks

• **Purpose:**

• Developing a software framework called DistBelief that can utilize computing clusters with thousands of machines to train large models
Need for Distributed Training

• Increasing the scale of deep learning drastically improve ultimate classification accuracy
  • Scaling with respect to
  • The number of training examples
  • The number of model parameters

• Use of GPUs is a significant advance in recent years
Need for Distributed Training (cont’d)

• A known limitation of the GPU approach is that the training speed-up is small when the model does not fit in GPU memory
  • To use a GPU effectively, researchers often reduce the size of the data or parameters

• While data and parameter reduction work well for small problems, they are less attractive for problems with a large number of examples and dimensions
Proposed *DistBelief* Framework

- DistBelief can utilize computing clusters with thousands of machines to train large models
- Enables model parallelism
  - Within a machine (via multithreading)
  - Across machines (via MPI)
- Enables data parallelism
  - Multiple replicas of a model are used to optimize a single objective
Requirements for Training Large Models in DistBelief

• Need to distribute training across multiple model instances/replicas

• Simultaneously solve a single optimization problem
  • By employing a set of DistBelief model instances, or replicas

• Take advantage of two novel methods for large-scale distributed training
  • Downpour SGD
  • Sandblaster L-BFGS
Background:
SGD: Stochastic Gradient Descent

• Calculates the error and updates the model for each example in the training dataset

• Pros
  • Faster learning in some problems
  • Immediately gives an insight into the performance of the model and the rate of improvement
Background:
SGD: Stochastic Gradient Descent (cont’d)

• Cons
  • More computationally expensive
  • The frequent updates can result in a noisy gradient signal
    • May lead to a higher variance over training epochs
    • Also make it hard for the algorithm to settle on an error minimum for the model

• Also referred to online training
Background: Batch Gradient Descent

• Calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated

• Pros
  • More computationally efficient
  • More stable error gradient
  • Easier to code parallel processing based implementations
Background:
Batch Gradient Descent (cont’d)

• Cons
  • May lead premature convergence
  • Additional complexity of accumulating prediction errors
  • Requires the entire training dataset in memory
  • May be very slow for large datasets
Background:
Mini-Batch Gradient Descent

• Splits the training dataset into small batches that are used to calculate model error and update model coefficients
• Seeks to find a balance between Stochastic and Batch gradient descent
• More robust convergence compared to Batch
• More computationally efficient SGD
• Efficiency of not having all training data in memory
Downpour SGD

- Divide the training data into a number of subsets
- Run a copy of the model on each of the subsets
- Models communicate updates through a centralized parameter server (PS)
Downpour SGD (cont’d)

- Asynchronous aspects
  - Model replicas run independently
  - Parameter server shards also run independently
Sandblaster L-BFGS

- A batch method
- BFGS is an iterative method for solving unconstrained nonlinear optimization problems
- Key idea
  - Distributed parameter storage and manipulation
- L-BFGS resides in coordinator process
  - Does not have direct access to the model parameters
Sandblaster L-BFGS (cont’d)

• Coordinator issues commands drawn from a small set of operations
• Sandblaster workers only fetch parameters at the beginning of each batch
• This allows running large models (billions of parameters)
  • Without incurring the overhead of sending all the parameters and gradients to a single central server
Load-Balancing Issue

• Typically, in L-BFGS data is distributed to many machines
  • Each machine computes the gradient on a specific subset of data examples
  • Then gradients are sent back to a central server
• Such methods wait for the slowest machine
  • Not scalable to large shared cluster
Proposed Load-Balancing

- The coordinator assigns each of the N model replicas a small portion of work
  - Much smaller than 1/Nth of the total size of a batch
- Assigns replicas new portions whenever they are free
- Faster model replicas do more work than slower replicas!
Comparison of Two Proposed Methods

- Downpour SGD, an online method, and Sandblaster L-BFGS, a batch method
- Both methods leverage the concept of a centralized sharded parameter server
  - Model replicas share their parameters
- Both methods tolerate variance in the processing speed of different model replicas and failure
- Both approaches simultaneously process distinct training examples in model replicas, and periodically combine their results
Experimental Specification

• Training models for two different deep learning problems:
  • Object recognition in images
  • Acoustic processing for speech recognition
• Considering two baseline optimization procedures by training a DistBelief model (on 8 partitions) using
  • Conventional (single replica) SGD
  • Training the identical model on a GPU using CUDA
Optimization Methods Comparison

- For any fixed budget of machines or cores
  - Downpour SGD with Adagrad takes less time to reach the accuracy target
Optimization Methods Comparison

- For any fixed budget of machines or cores
  - Downpour SGD with Adagrad takes less time to reach the accuracy target
- The Sandblaster L-BFGS system does show promise in terms of its scaling with additional cores,
  - Suggesting that it may ultimately produce the fastest training times
Thanks!

Mohammadreza Bayatpour
bayatpour.1@osu.edu