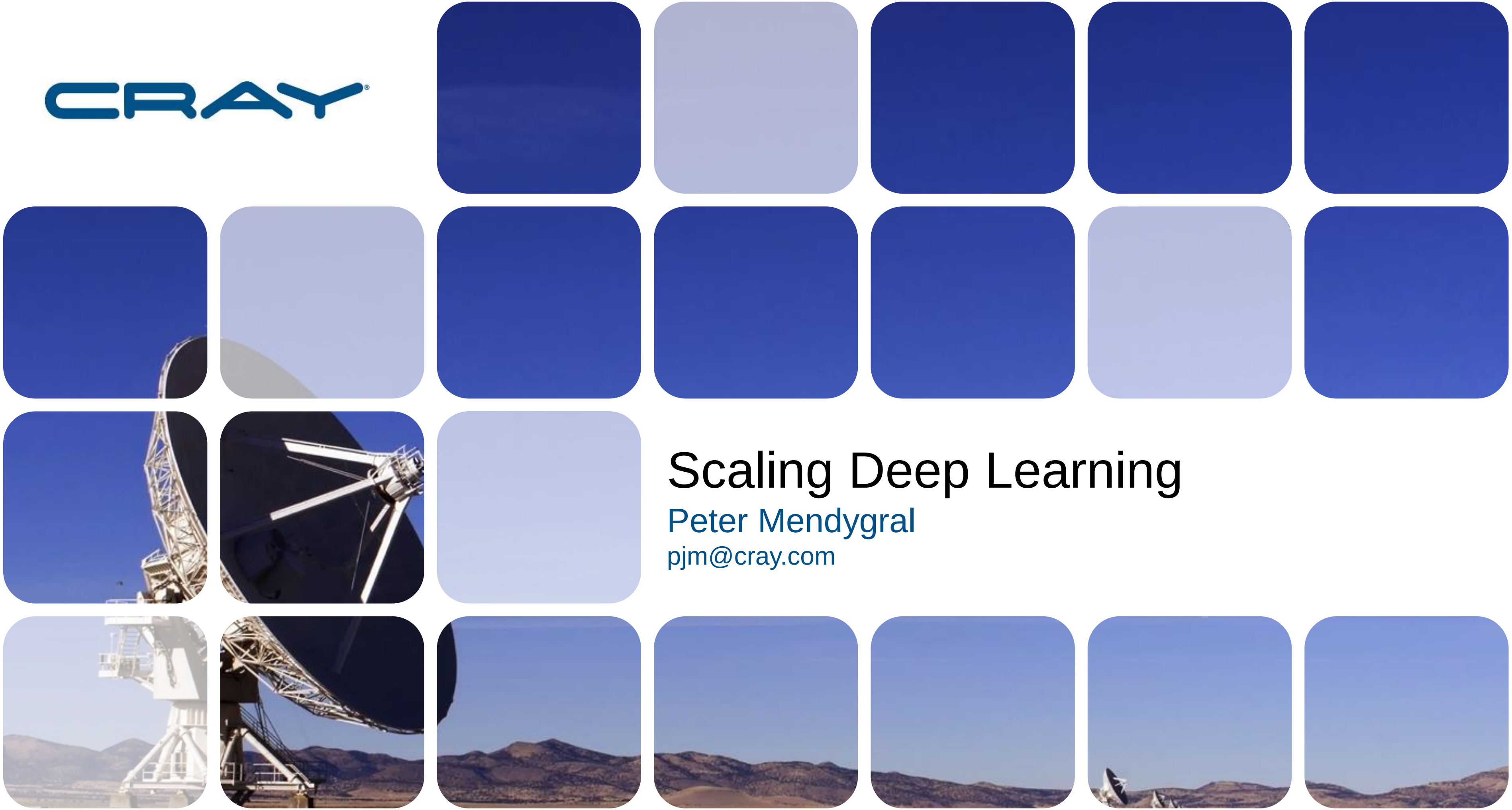


Scaling Deep Learning

Peter Mendygral
pjm@cray.com

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM, REVEAL. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

Motivation

- **A trained neural network can be a powerful tool for**
  - Pattern recognition
  - Classification
  - Clustering
  - Others…

- **Scaling Deep Learning (DL) training is also a tool for**
  - Models that take a very long time to train (and have a very large training dataset)
  - Increasing the frequency at which models can be retrained with new or improved data

- **This talk reviews scaling DL training and topics that can be important to successfully applying it**

Agenda

- HPC Attributes of Deep Learning
- TensorFlow on Theta
- Parallelization Methods for TensorFlow
- Convergence Considerations at Scale
- CPE ML Plugin Example

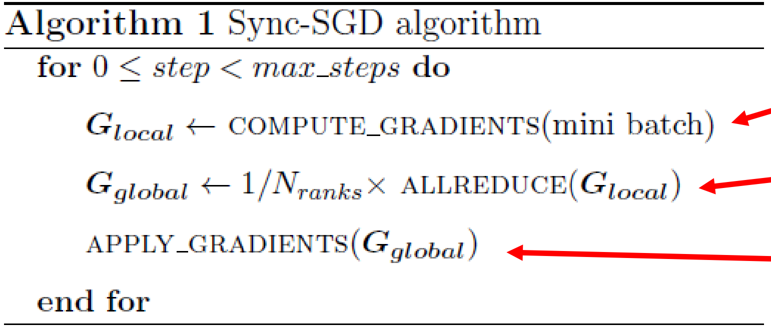
HPC Attributes of Deep Learning

HPC Attributes

- **DL training is a classic high-performance computing problem which demands:**
  - Large compute capacity in terms of FLOPs, memory capacity and bandwidth
  - A performant interconnect for fast communication of gradients and model parameters
  - Parallel I/O and storage with sufficient bandwidth to keep the compute fed at scale

Data Parallelism - Collective-based Synchronous SGD

- Data parallel training divides a global mini-batch of examples across processes
- Each process computes gradients from their local mini-batch
- Average gradients across processes
- All processes update their local model with averaged gradients (all processes have the same model)



Algorithm 1 Sync-SGD algorithm

```
for $0 \leq step < max\_steps$ do
    $G_{local} \leftarrow$ COMPUTE_GRADIENTS(mini batch)
    $G_{global} \leftarrow 1/N_{ranks} \times$ ALLREDUCE($G_{local}$)
    APPLY_GRADIENTS($G_{global}$)
end for
```

Compute intensive

Communication intensive

Typically not much compute

- Not shown is the I/O activity of reading training samples (and possible augmentation)

Why do we want to scale?

- **Deep Network Training**
  - We can strong scale training time-to-accuracy provided
    - Number of workers (e.g., # nodes) << number of training examples
    - Learning rate for particular batch size / scale is known
- **Hyper-Parameter Optimization**
  - For problems and datasets where baseline accuracy is not known
    - learning rate schedule
    - momentum
    - batch size
  - Evolve topologies if good architecture is unknown (common with novel datasets / mappings)
    - Layer types, width, number filters
    - Activation functions, drop-out rates

TensorFlow on Theta

TensorFlow

- **Developed by Google**
- **Most popular DL framework**
- **Large open source community**
- **APIs for**
  - Python
  - C++
  - Go
  - Java
- **Optimized for CPU and GPU architectures**

TensorFlow on Theta

- Python 2.7 and 3.6 builds of TensorFlow prepared for this workshop

- TensorFlow 1.5 with MKL-DNN optimizations for KNL

- Python 2.7 setup

```
module load cray-python/2.7.13.1
export PYTHONUSERBASE=/lus/theta-fs0/projects/SDL_Workshop/mendygra/pylibs
python –c “import tensorflow as tf”
```

- Python 3.6 setup

```
module load cray-python/3.6.1.1
export PYTHONUSERBASE=/lus/theta-fs0/projects/SDL_Workshop/mendygra/pylibs
python –c “import tensorflow as tf”
```

Performance Tuning Tips for KNL

- **Recommended MKL settings**
  - OMP_NUM_THREADS=62
  - KMP_BLOCKTIME=0 (30 sometimes good too)
  - KMP_AFFINITY=“granularity=fine,compact,1,0”
  - TensorFlow thread settings
    - num_inter_threads=3
    - num_intra_threads=$OMP_NUM_THREADS
- **Use NCHW data format (NHWC is TensorFlow default)**
- **Use the Dataset API to pipeline reading and preparing of input samples**
- **I/O bandwidth requirements for Dataset API (using dedicated preprocessing threads) can be estimated with**
  - `B/s/node = (#processes/node) x (local mini-batch size) x (B/sample) / (batch time [s])`
- **Use lustre striping on sample data directory, for example:**
  - `lfs setstripe -c 16 [samples directory]`
  - `cp [dataset files] [samples directory]`

Parallelization Methods for DL

Parallelization Techniques

- **Data Parallelism**
  - As described earlier, divides global mini-batch among processes
  - Two methods for this:
    - Synchronous: single model (possibly replicated across all processes) updated with globally averaged gradients every iteration
    - Asynchronous: processes provide gradients every iteration but are allowed to fall out of sync from one another. Processes each have their own model that may or may not be the same as any other process
- **Model Parallelism**
  - Single model with layers decomposed across processes
  - Activations communicated between processes
- **This talk will focus on synchronous data parallel approach**

Distributed TensorFlow

- **TensorFlow has a native method for parallelism across nodes**
  - ClusterSpec API
  - Uses gRPC layer in TensorFlow based on sockets

- **Can be difficult to use and optimize**

- **User must specify**
  - hostnames and ports for all worker processes
  - hostnames and ports for all parameter server processes (see next slide)
  - # of workers
  - # of parameter server processes
  - Chief process of workers

Distributed TensorFlow

- **Number of parameter servers (PS) processes to use is not clear**
  - Too few results in many-to-few comm pattern (very bad) and stalls delivering updated parameters
  - Too many results in many-to-many comm patter (also bad)

- **Users typically have to pick a scale and experiment for best performance**

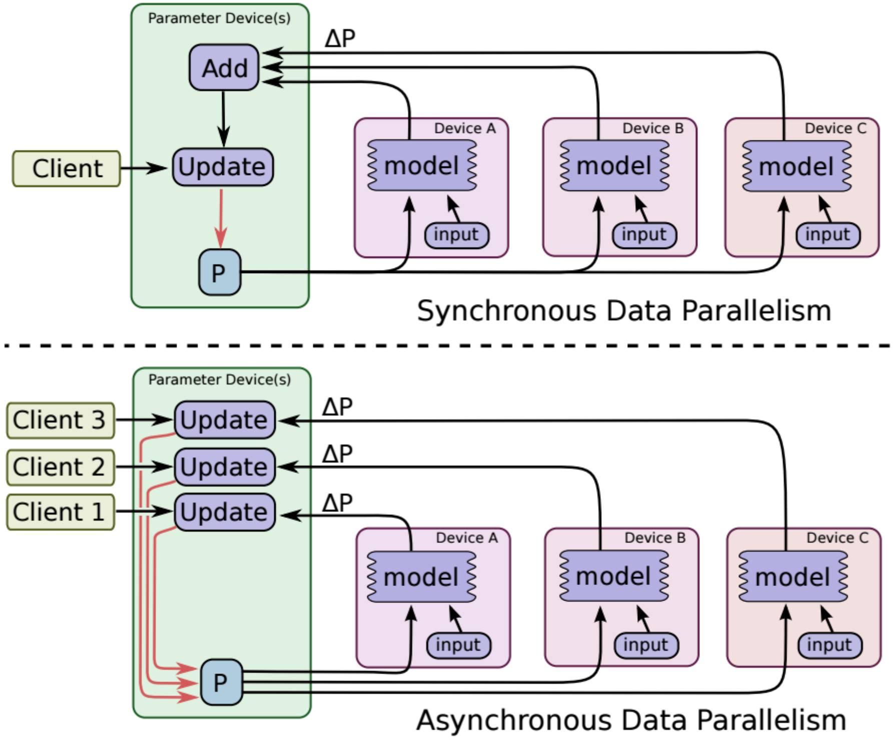
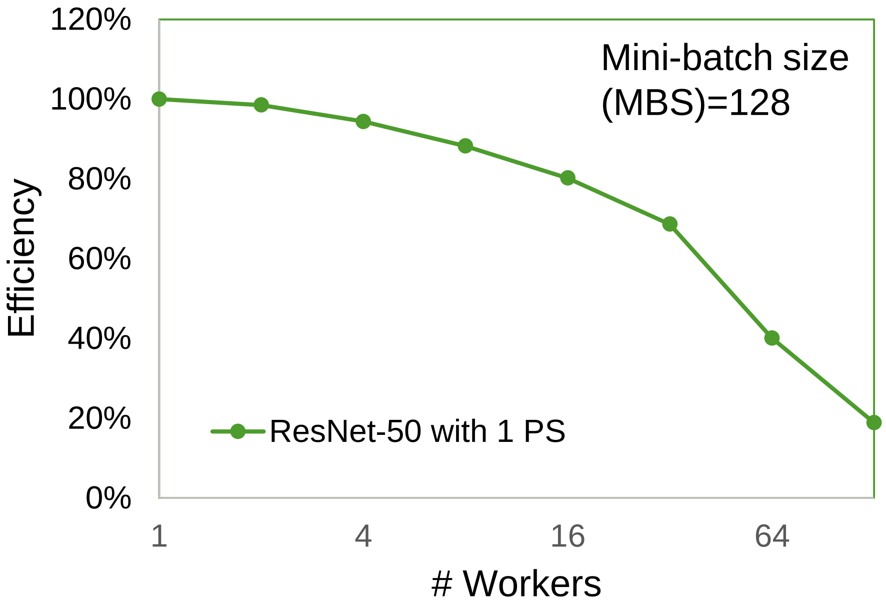
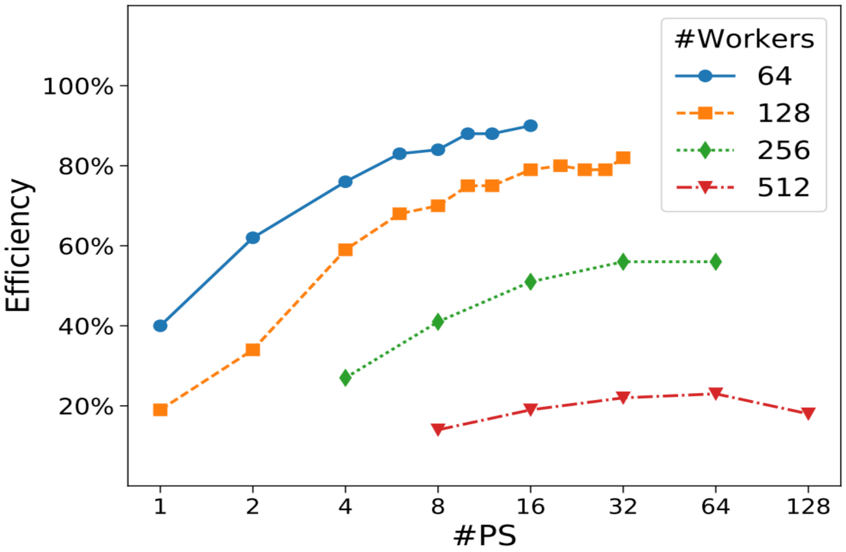


Figure 7: Synchronous and asynchronous data parallel training

Distributed TensorFlow Scaling on Cray XC40 - KNL



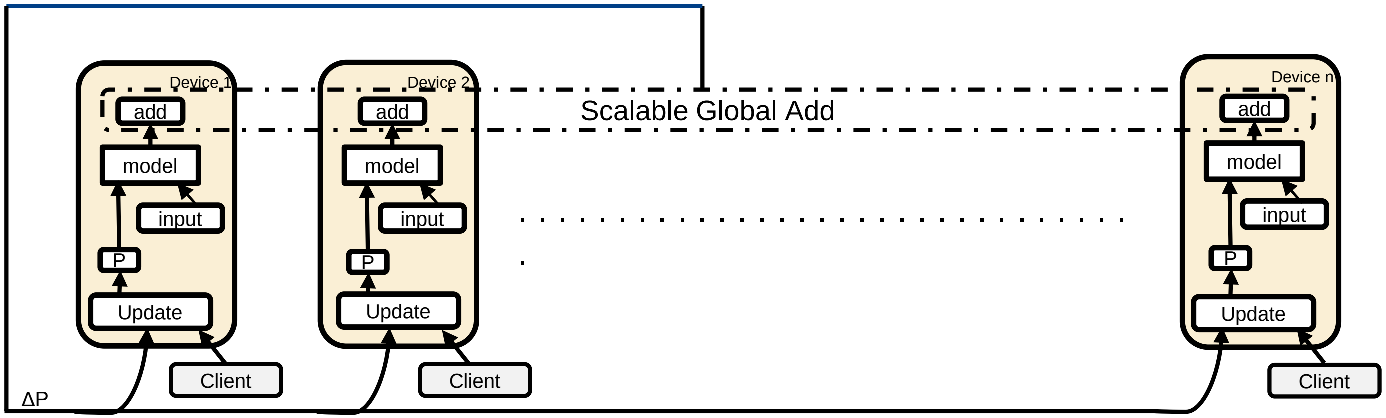


From Mathuriya et al. @ NIPS 2017

MPI-based Data Parallel TensorFlow

- **The performance and usability issues with distributed TensorFlow can be addressed by adopting an MPI communication model**
- **TensorFlow does have an MPI option, but it only replaces point to point operations in gRPC with MPI**
  - Collective algorithm optimization in MPI not used
- **Other frameworks, such as Caffe and CNTK, include MPI collectives**
- **An MPI collective based approach would eliminate the need for PS processes and likely be optimized without intervention from the user**

Scalable Synchronous Data Parallelism



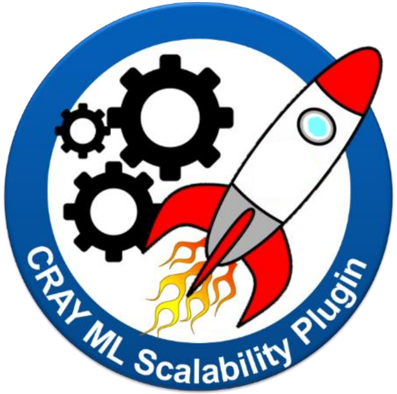
- Note there are no PS processes in this model
- Resources dedicated to gradient calculation

Uber Horovod

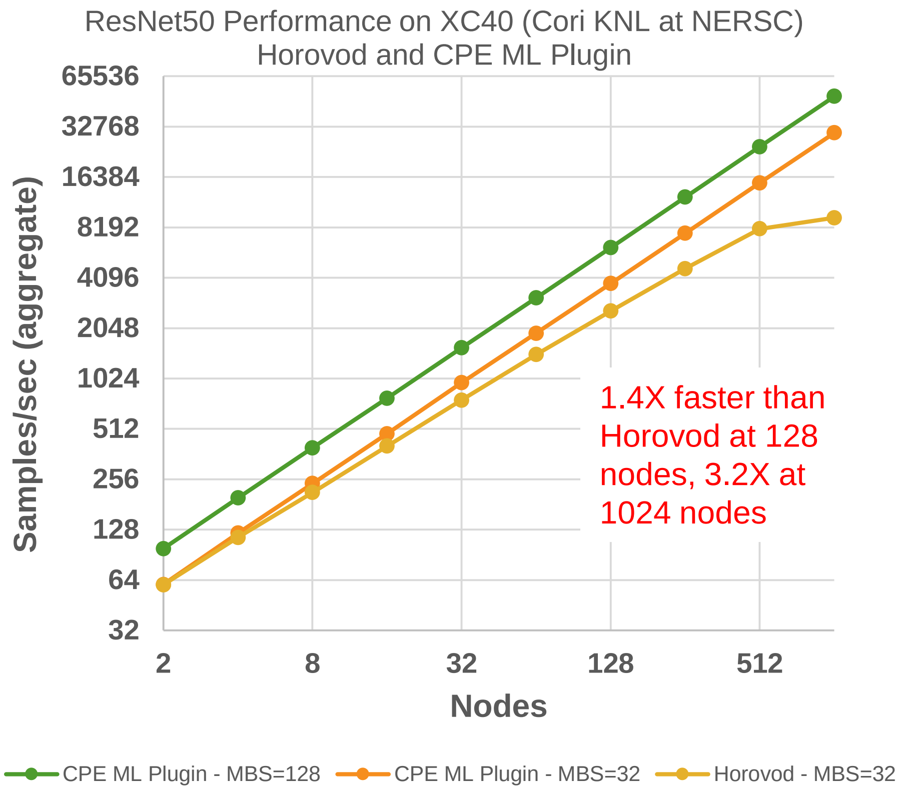
- **Uber open source addon for TensorFlow *only* that replaces native optimizer class with a new class**
  - Horovod adds an allreduce between gradient computation and model update in this class
- **New Python class includes NCCL and MPI collective reductions for gradient aggregation**
- **https://github.com/uber/horovod**
- **No modifications to TensorFlow source required**
  - User modifies Python training script instead

Cray Programming Environment Machine Learning Plugin (CPE ML Plugin)

- **DL communication plugin with Python and C APIs**
- **Optimized for TensorFlow but also portable to other frameworks**
  - Callable from C/C++ source
  - Called from Python if data stored in NumPy arrays or Tensors
- **Like Horovod does not require modification to TensorFlow source**
  - User modifies training script
- **Uses custom allreduce specifically optimized for DL workloads**
  - Optimized for Cray Aries interconnect and IB for Cray clusters
- **Tunable through API and environment variables**
- **Supports multiple gradient aggregations at once with thread teams**
  - Useful for Generative Adversarial Networks (GAN), for example

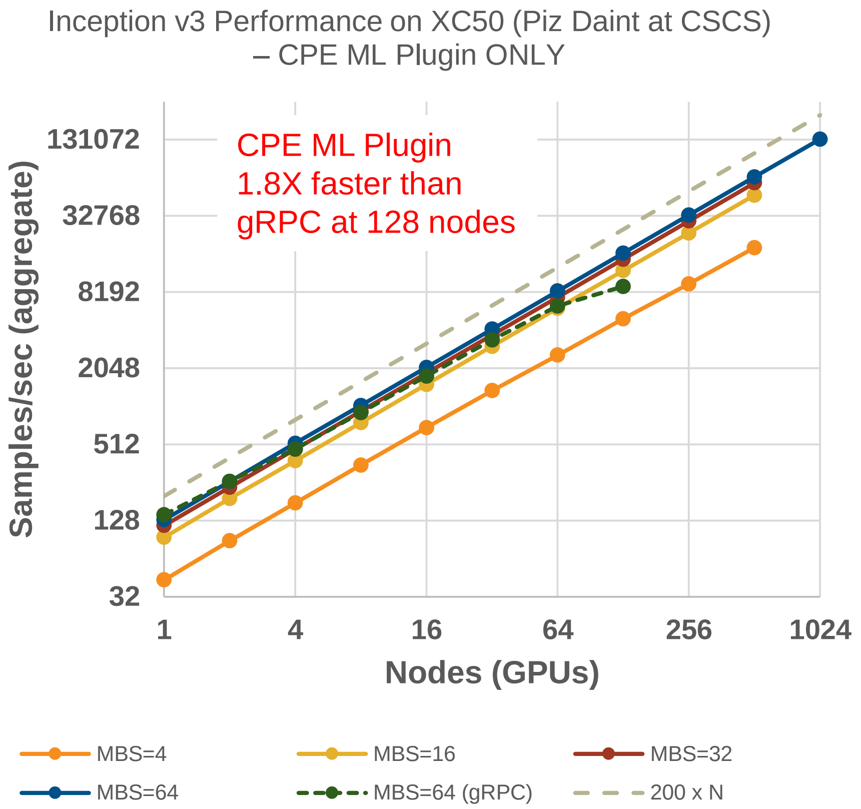


Horovod / CPE ML Plugin – Throughput Scaling



ResNet50 Performance on XC40 (Cori KNL at NERSC)
Horovod and CPE ML Plugin

1.4X faster than Horovod at 128 nodes, 3.2X at 1024 nodes



Inception v3 Performance on XC50 (Piz Daint at CSCS)
– CPE ML Plugin ONLY

CPE ML Plugin
1.8X faster than
gRPC at 128 nodes

Convergence Considerations at Scale

Problems in Scaling DL Training

- **Increasing workers increases the global batch size**
  - This reduces the number of updates to the model (iterations) per epoch (full pass through dataset)
  - Can require more iterations to converge to same validation accuracy for models trained at smaller batch sizes

- **Large-batch (LB) training can have different convergence properties than Small-batch (SB) training**
  - LB training can lead to models which fail to generalize to validation datasets
  - LB training error can look similar to SB training error, but validation error fails to improve

Problems in Scaling DL Training

- **Possible reasons for the observed failure to generalize using large batch methods [2]:**
  - The model overfits
  - Optimization is attracted to saddle-points
  - Loss of the explorative properties gained with small batches

Observations on Scaled Learning Rates

- **Step 1) Start with common initial learning rate for selected optimizer (from Keras documentation)**
  - Adam -> 0.001
  - RMSProp -> 0.001
  - SGD -> 0.01
  - Adagrad -> 0.01
  - Adadelta -> 1.0
- **Step 2) Multiply learning rate by N or Sqrt(N)**
  - N is the number of parallel processes
  - Discussed in further detail on next slide
- **Step 3) Decay learning rate during training (e.g., exponential decay)**
  - Setup a learning rate schedule using your initial learning rate as the starting state
  - Learning rate typically lowered periodically or continuously
  - Helps improve final accuracy
  - Likely very important to reduce learning rate over time when initial learning rate scaled large
- **Step 4) Run it**
  - Train and observe loss or training accuracy, check validation accuracy
  - Adjust initial learning rate up if learning too slowly or down if model is not learning
  - Repeat steps as needed to improve convergence and accuracy

Learning Rate Scaling Rules

- **Sqrt Scaling Rule:**
  - When the local minibatch size is multiplied by $N_{workers}$, multiply the learning rate by $\sqrt{N_{workers}}$.

$$\eta_{init} = \eta_{init} * \sqrt{N_{workers}}$$

  - Error on the mean only improves as sqrt(N_workers)

- **Linear Scaling Rule:**
  - When the minibatch size is multiplied by N, multiply the learning rate by N.

$$\eta_{init} = \eta_{init} * N_{workers}$$

  - Naïve rule for scaling learning rate in distributed training but it works for some problems
  - More attractive (when it works) because it shouldn't require many additional iterations to reach same accuracy

Rules of Thumb

- **3 Remarks from Facebook Paper:**
  - “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour” [1]

- **Momentum: If using momentum with learning rate absorbed into the update tensor, apply momentum correction described in the paper**
  - Tensorflow:
    - RMSProp momentum needs corrections
    - GradientDescentOptimizer can ignore correction
    - Anything using momentum, inspect further

- **Batch-normalization**
  - Consider your local batch-size to be a hyper-parameter of the batch-normalization, so it affects the loss computation locally, and the node efficiency

- **Gradient Aggregation: Normalize the per-worker loss by global minibatch size, not local**
  - Already handled by parallel methods like gRPC, Horovod and CPE ML Plugin

Other Rules of Thumb

- **Optimizers can behave in unexpected ways as scale increases**
  - At small scale (e.g., < 100 workers) errors from assumptions in common optimizers also probably small
    - Learning rate scaling rules and schedule may be sufficient
  - At larger scales optimizers can break down and require corrections
  - Improved optimizers are likely required for very large global batch sizes

Warm-Up Iterations

- **Linearly scaled learning rate causes most problems early in training [3]**
  - Design a warm-up set of iterations to reduce these errors
  - Once training settled on good path, transition to larger learning rate
- **Can also use different optimizers for each phase [5]**
- **Allows you to use momentum and weight decay**

RMSProp -> momentum SGD given in [5]

$$m_t = \mu_2 m_{t-1} + (1 - \mu_2) g_t^2,$$

$$\Delta_t = \mu_1 \Delta_{t-1} - \left(\alpha_{\mathrm{SGD}} + \frac{\alpha_{\mathrm{RMSprop}}}{\sqrt{m_t + \varepsilon}} \right) g_t, \text{ and}$$

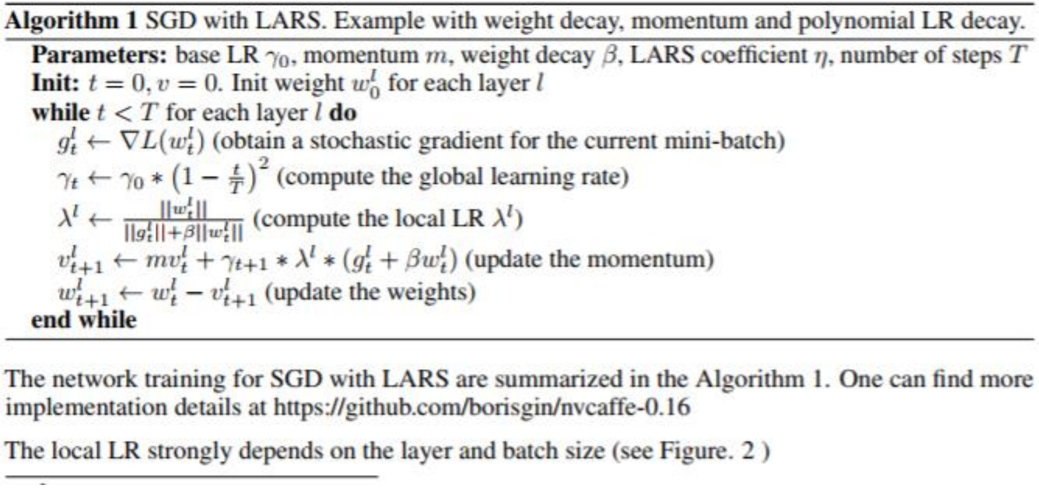
$$\theta_t = \theta_{t-1} + \eta \Delta_t .$$

$$\alpha_{\mathrm{SGD}} = \begin{cases} \frac{1}{2}\exp(2(\mathrm{epoch} - \beta_{\mathrm{center}})/\beta_{\mathrm{period}}) & (\mathrm{epoch} < \beta_{\mathrm{center}}) \\ \frac{1}{2} + 2(\mathrm{epoch} - \beta_{\mathrm{center}})/\beta_{\mathrm{period}} & (\mathrm{epoch} < \beta_{\mathrm{center}} + \frac{1}{2}\beta_{\mathrm{period}}) \\ 1 & (\mathrm{otherwise}) \end{cases} .$$

This 2 is a mistake, should be 1

Layer-wise Adaptive Rate Scaling (LARS)

- **Layer-wise learning rate**
- **Allows you to use momentum and weight decay**
- **Demonstrated no loss of accuracy on ResNet50 with global batch size of 32K**
  - Could be 1024 nodes each with local batch size of 32
- **See reference [1]**



Algorithm 1 SGD with LARS. Example with weight decay, momentum and polynomial LR decay.

```
**Parameters:** base LR $\gamma_0$, momentum $m$, weight decay $\beta$, LARS coefficient $\eta$, number of steps $T$
**Init:** $t = 0, v = 0$. Init weight $w_0^l$ for each layer $l$
**while** $t < T$ for each layer $l$ **do**
    $g_t^l \leftarrow \nabla L(w_t^l)$ (obtain a stochastic gradient for the current mini-batch)
    $\gamma_t \leftarrow \gamma_0 * \left(1 - \frac{t}{T}\right)^2$ (compute the global learning rate)
    $\lambda^l \leftarrow \frac{||w_t^l||}{||g_t^l|| + \beta ||w_t^l||}$ (compute the local LR $\lambda^l$)
    $v_{t+1}^l \leftarrow m v_t^l + \gamma_{t+1} * \lambda^l * (g_t^l + \beta w_t^l)$ (update the momentum)
    $w_{t+1}^l \leftarrow w_t^l - v_{t+1}^l$ (update the weights)
**end while**
```

The network training for SGD with LARS are summarized in the Algorithm 1. One can find more implementation details at https://github.com/borisgin/nvcaffe-0.16

The local LR strongly depends on the layer and batch size (see Figure. 2)

Useful References

[1] LARGE BATCH TRAINING OF CONVOLUTIONAL NETWORKS -- https://arxiv.org/pdf/1708.03888.pdf

[2] ON LARGE-BATCH TRAINING FOR DEEP LEARNING: GENERALIZATION GAP AND SHARP MINIMA -- https://openreview.net/pdf?id=H1oyRlYgg

[3] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour -- https://research.fb.com/wp-content/uploads/2017/06/imagenet1kin1h5.pdf

[4] Train longer, generalize better: closing the generalization gap in large batch training of neural networks -- https://arxiv.org/pdf/1705.08741.pdf

[5] Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes -- https://arxiv.org/pdf/1711.04325.pdf

CPE ML Plugin Example

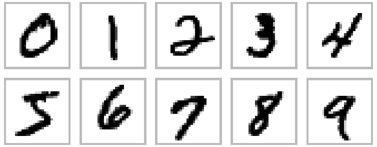
Training Script Modifications

- **Both Horovod and CPE ML Plugin require some modifications to a serial training script**

- **For the CPE ML Plugin the changes are**
  - Importing the Python module
  - Initialize the module
    - Possibly configure the thread team(s) for specific uses
  - Broadcast initial model parameters
  - Incorporate gradient aggregation between gradient computation and model update
  - Finalize the Python module

MNIST Example

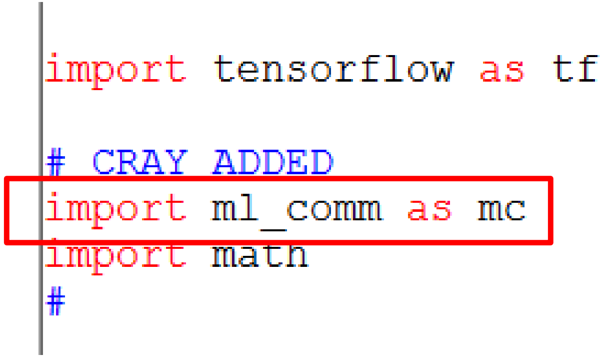
- Dataset of handwritten digits from 0-9
- Simple CNN can be used to identify handwritten digits



- This example is adapted from the TensorFlow official MNIST example
- https://github.com/tensorflow/models/tree/master/official/mnist
- Modified script included with CPE ML Plugin
  - `module load craype-ml-plugin-py2/1.1.0`
  - `less $CRAYPE_ML_PLUGIN_BASEDIR/examples/tf_mnist/mnist.py`

CPE ML Plugin - Import

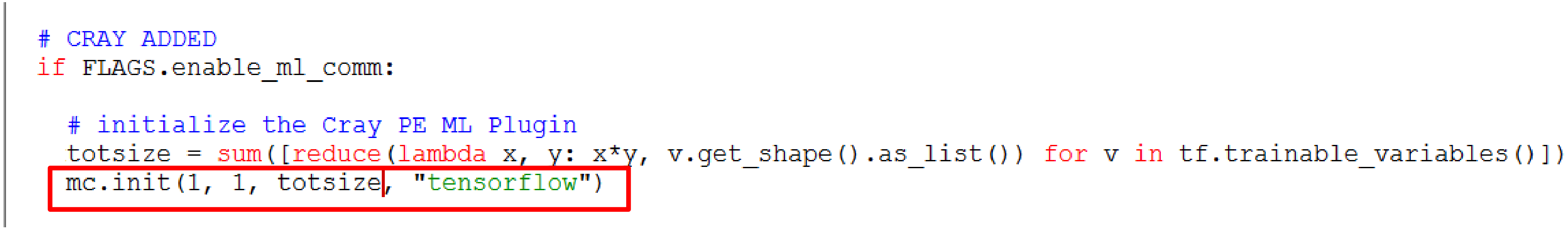
- Access the Python API by importing the module



```
import tensorflow as tf

# CRAY ADDED
import ml_comm as mc
import math
#
```

CPE ML Plugin - Initialization

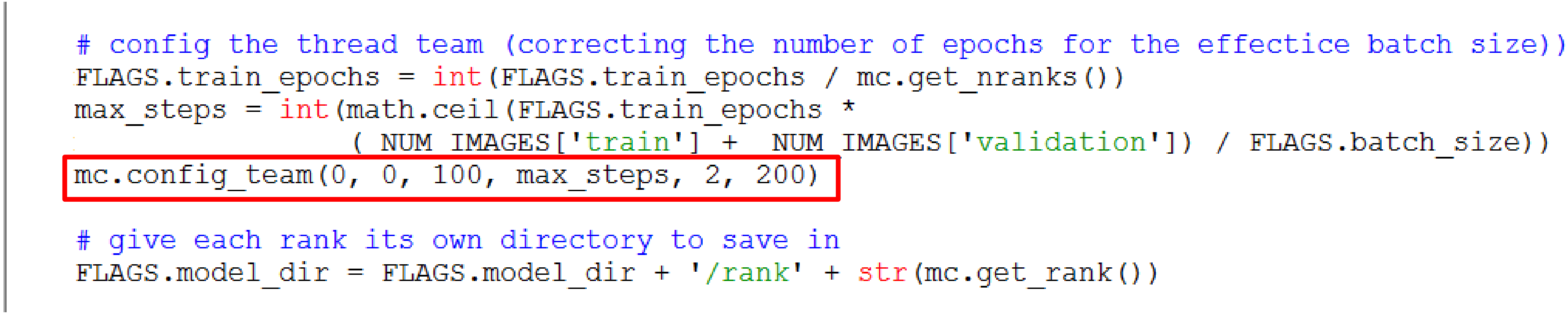
- **Compute the number of trainable variables in the model**
  - Required for the CPE ML Plugin to pre-allocate needed communication buffers
  - Example sets up a single thread team with one thread



```
# CRAY ADDED
if FLAGS.enable_ml_comm:

  # initialize the Cray PE ML Plugin
  totsize = sum([reduce(lambda x, y: x*y, v.get_shape().as_list()) for v in tf.trainable_variables()])
  mc.init(1, 1, totsize, "tensorflow")
```

CPE ML Plugin – Team Configuration

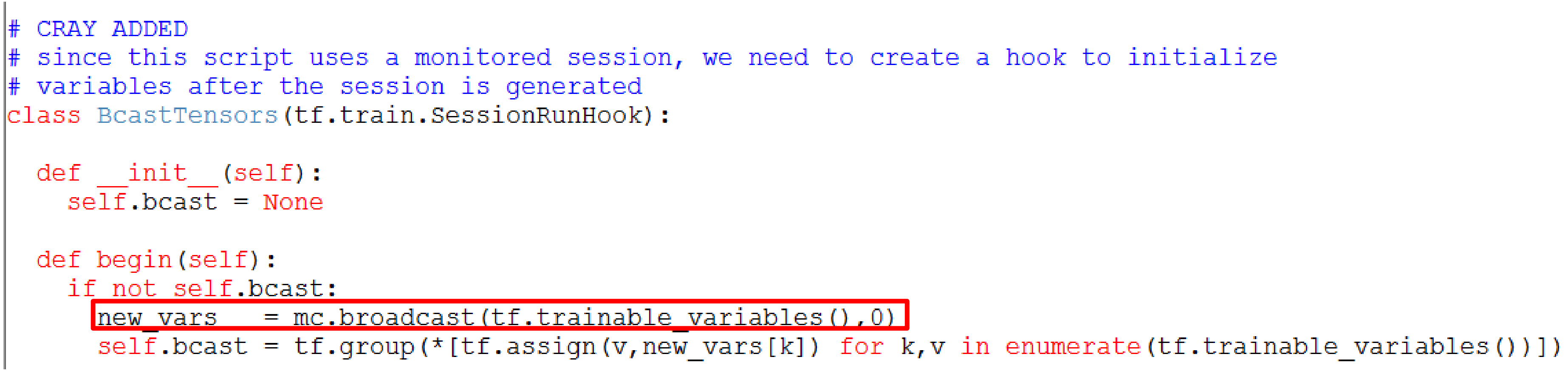
- **Set the maximum number of steps (mini batches) to train for**
  - Verbose output every 200 steps
- **Also set output path to rank-specific location**



```
# config the thread team (correcting the number of epochs for the effectice batch size))
FLAGS.train_epochs = int(FLAGS.train_epochs / mc.get_nranks())
max_steps = int(math.ceil(FLAGS.train_epochs *
                 ( NUM IMAGES['train'] +  NUM IMAGES['validation']) / FLAGS.batch_size))
mc.config_team(0, 0, 100, max_steps, 2, 200)

# give each rank its own directory to save in
FLAGS.model_dir = FLAGS.model_dir + '/rank' + str(mc.get_rank())
```

CPE ML Plugin – Broadcast Initial Model

- Broadcast initial model parameter values from rank 0 to all other ranks
- Then assign broadcasted values locally



```
# CRAY ADDED
# since this script uses a monitored session, we need to create a hook to initialize
# variables after the session is generated
class BcastTensors(tf.train.SessionRunHook):

  def __init__(self):
    self.bcast = None

  def begin(self):
    if not self.bcast:
      new_vars   = mc.broadcast(tf.trainable_variables(),0)
      self.bcast = tf.group(*[tf.assign(v,new_vars[k]) for k,v in enumerate(tf.trainable_variables())])
```

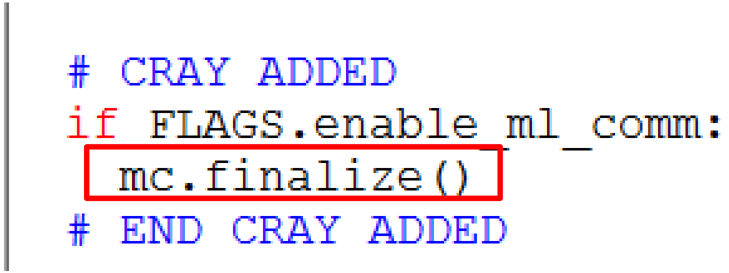
CPE ML Plugin – Gradient Aggregation

- Perform gradient averaging across all ranks between local gradient calculation and model update

```
# CRAY ADDED
if FLAGS.enable_ml_comm:

  # we need to split out the minimize call below so we can modify gradients
  grads_and_vars = optimizer.compute_gradients(loss)

  grads     = mc.gradients([gv[0] for gv in grads_and_vars], 0)
  gs_and_vs = [(g,v) for (_,v), g in zip(grads_and_vars, grads)]

  train_op = optimizer.apply_gradients(gs_and_vs,
                                       global_step=tf.train.get_or_create_global_step())
# END CRAY ADDED
```

CPE ML Plugin – Finalize

- After all training steps are complete clean up data structures and MPI



```
# CRAY ADDED
if FLAGS.enable_ml_comm:
  mc.finalize()
# END CRAY ADDED
```

CPE ML Plugin – Execution Example

- **Once the script is modified job launch is just like a typical MPI job**
  - Example assumes user has TensorFlow installed in PYTHONPATH or PYTHONUSERBASE

```
module load cray-python
module load craype-ml-plugin-py2/1.1.0
export OMP_NUM_THREADS=62

aprun -n4 -N1 -cc none -b python \
$CRAYPE_ML_PLUGIN_BASEDIR/examples/tf_mnist/mnist.py \
--enable_ml_comm \
--data_dir=/lus/theta-fs0/projects/SDL_Workshop/mendygra/mnist_data \
--model_dir=[train dir]
```

CPE ML Plugin – Example Batch Scripts

- **Sample batch scripts for tf_cnn_benchmarks**
  - Adapted from
    - https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks
  - A suite of CNNs are included

- **Batch scripts are available for you to try in:**
  - Python 2: /lus/theta-fs0/projects/SDL_Workshop/mendygra/cpe_plugin_py2.batch
  - Python 3: /lus/theta-fs0/projects/SDL_Workshop/mendygra/cpe_plugin_py3.batch

```
mkdir /lus/theta-fs0/projects/SDL_Workshop/[username]
cd /lus/theta-fs0/projects/SDL_Workshop/[username]
cp /lus/theta-fs0/projects/SDL_Workshop/mendygra/cpe_plugin_py2.batch .
qsub cpe_plugin_py2.batch
```

- **Please refer to CPE ML Plugin manpage for more details on usage**
  - `man intro_ml_plugin`

CRAY®

COMPUTE | STORE | ANALYZE

Questions?

Thank You!