TensorFlow at Scale

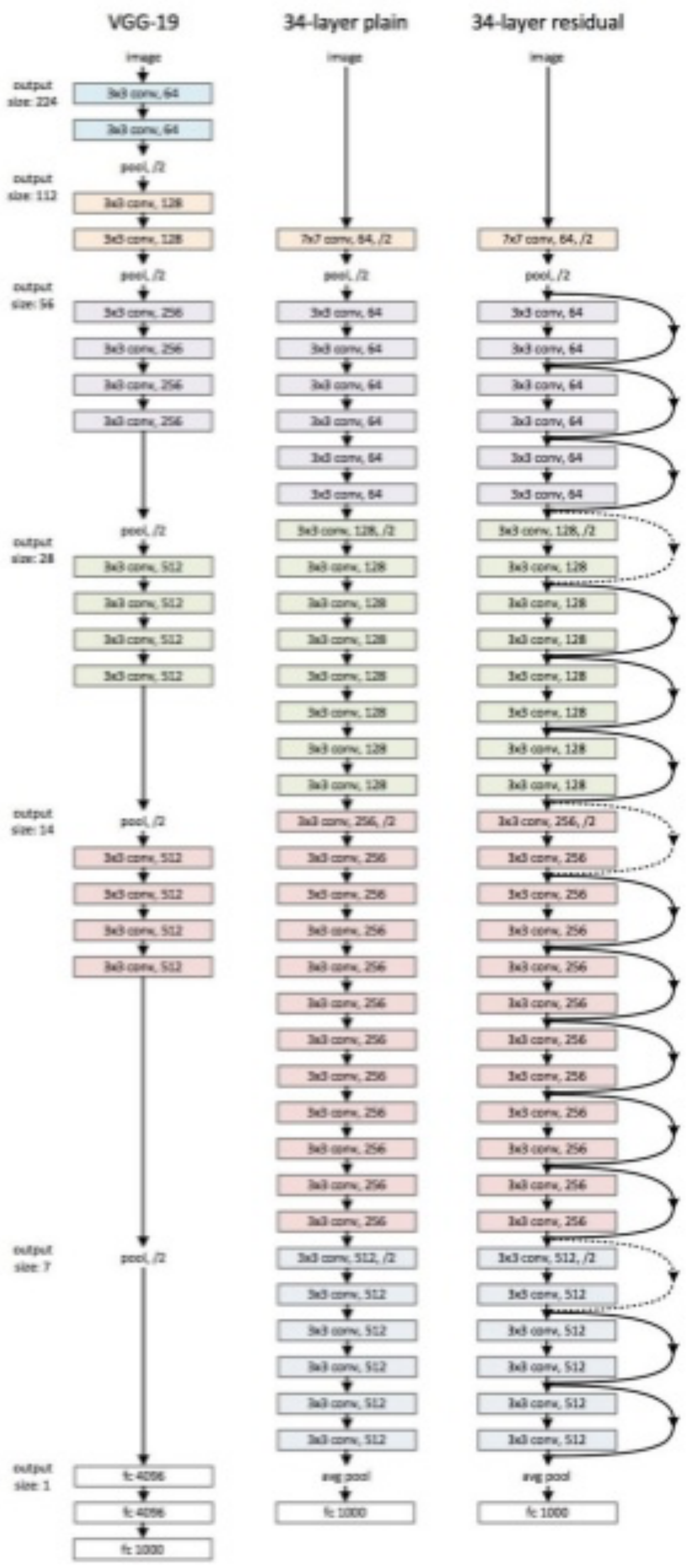
MPI, RDMA and All That

Thorsten Kurth, Mikhail Smorkalov, Peter Mendygral, Srinivas Sridharan, Amrita Mathuriya

CUG18
Stockholm, Sweden

Motivation for Scalable Deep Learning

- rapid prototyping/model evaluation
- problem scale
  - volume of scientific datasets can be large
  - scientific datasets can be complex (multivariate, high-dimensional)
- machine learning models become bigger (model parallelism)

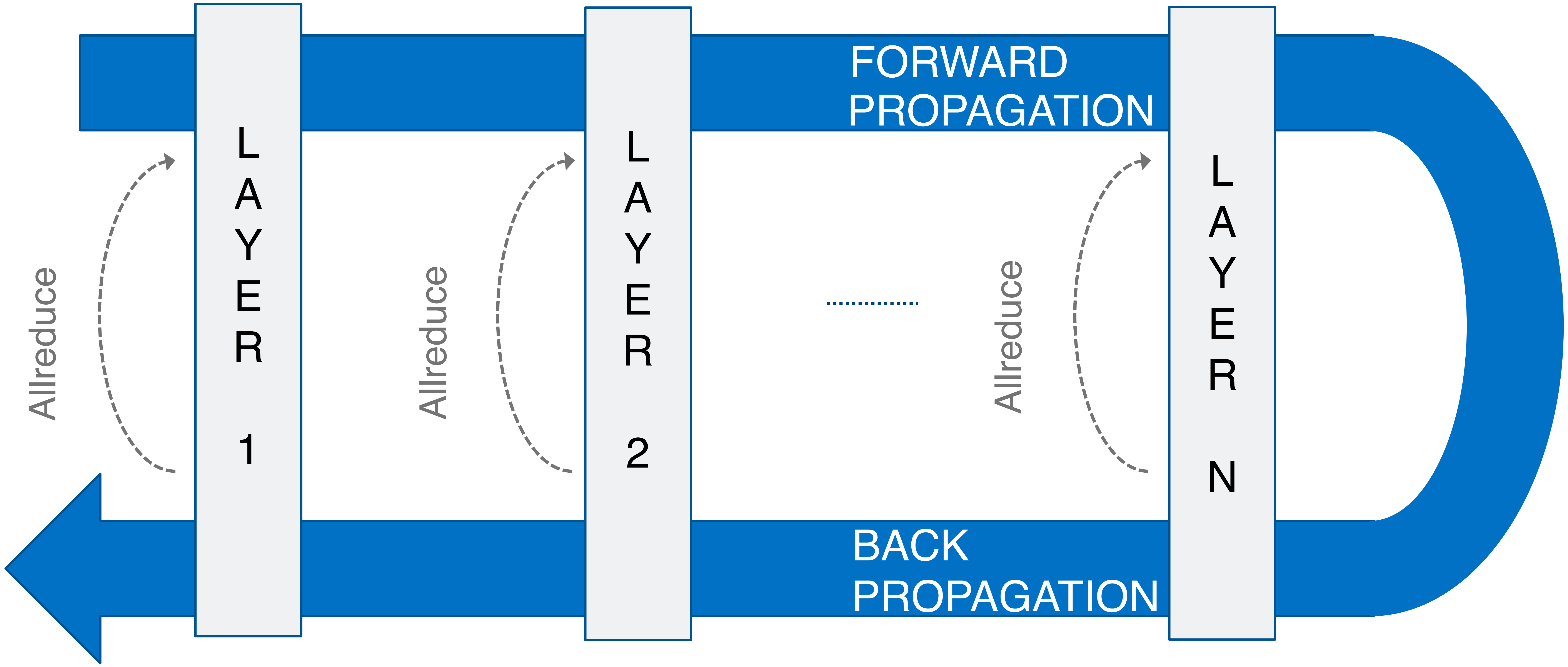


ResNet

http://tjmachinelearning.com/lectures/deep/deepnn.html

Data Parallel Training

- applies to Stochastic Gradient Descent-type algorithms
  - each node takes part of the data and computes model updates independently without communication
  - these updates are then collectively summed and applied to the local model



From Pradeep Dubey, "Scaling to Meet the Growing Needs of Artificial Intelligence (AI), IDF 2016
https://software.intel.com/en-us/articles/scaling-to-meet-the-growing-needs-of-ai

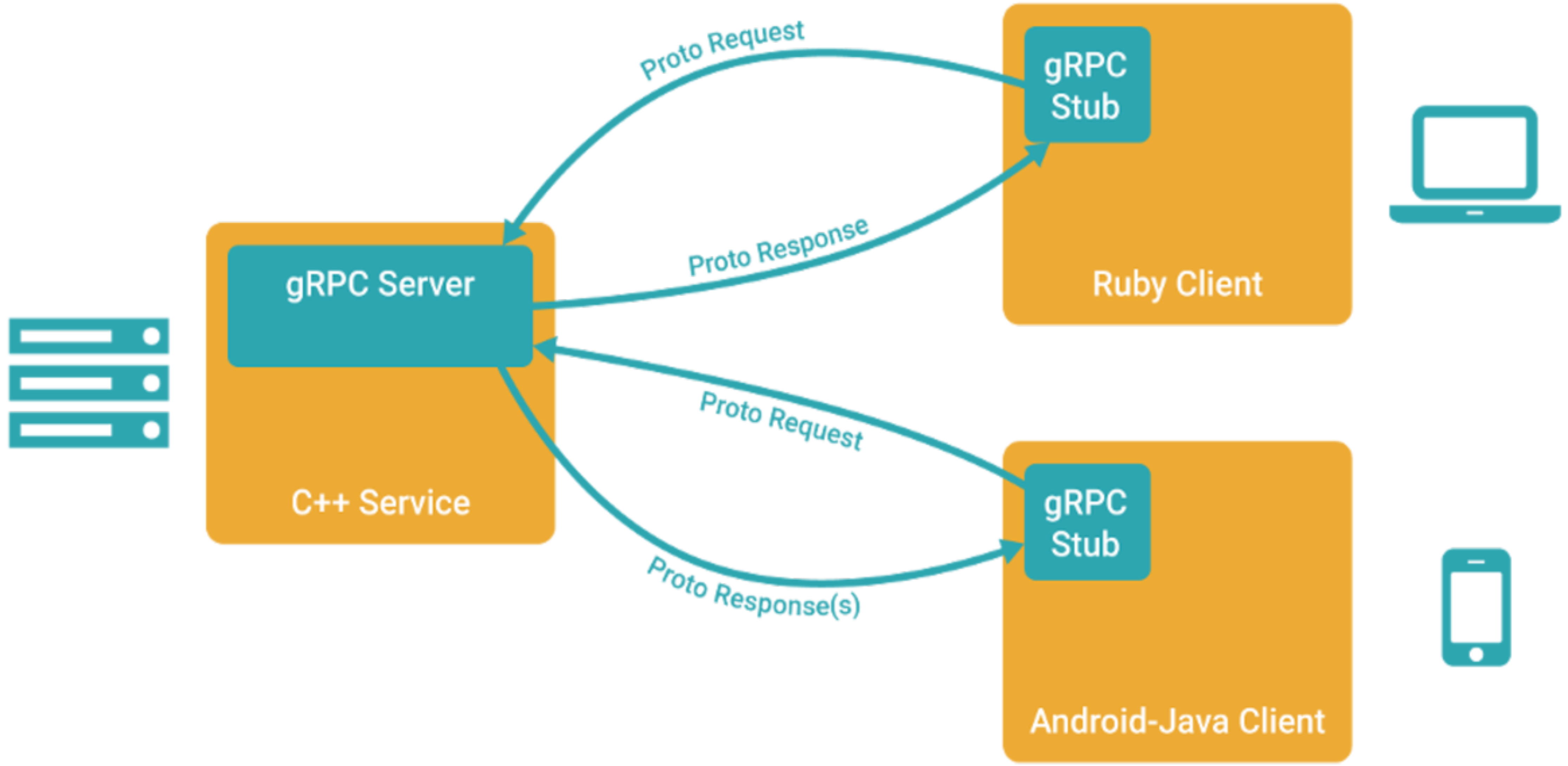
TensorFlow

- high-productivity deep learning framework
- uses Python functions with optimized backends (MKL-DNN, cuDNN)
- user defines graph and executes it in a `tf.Session`
- enables users to write efficient dl code for cutting edge hardware without knowledge about performance-oriented programming

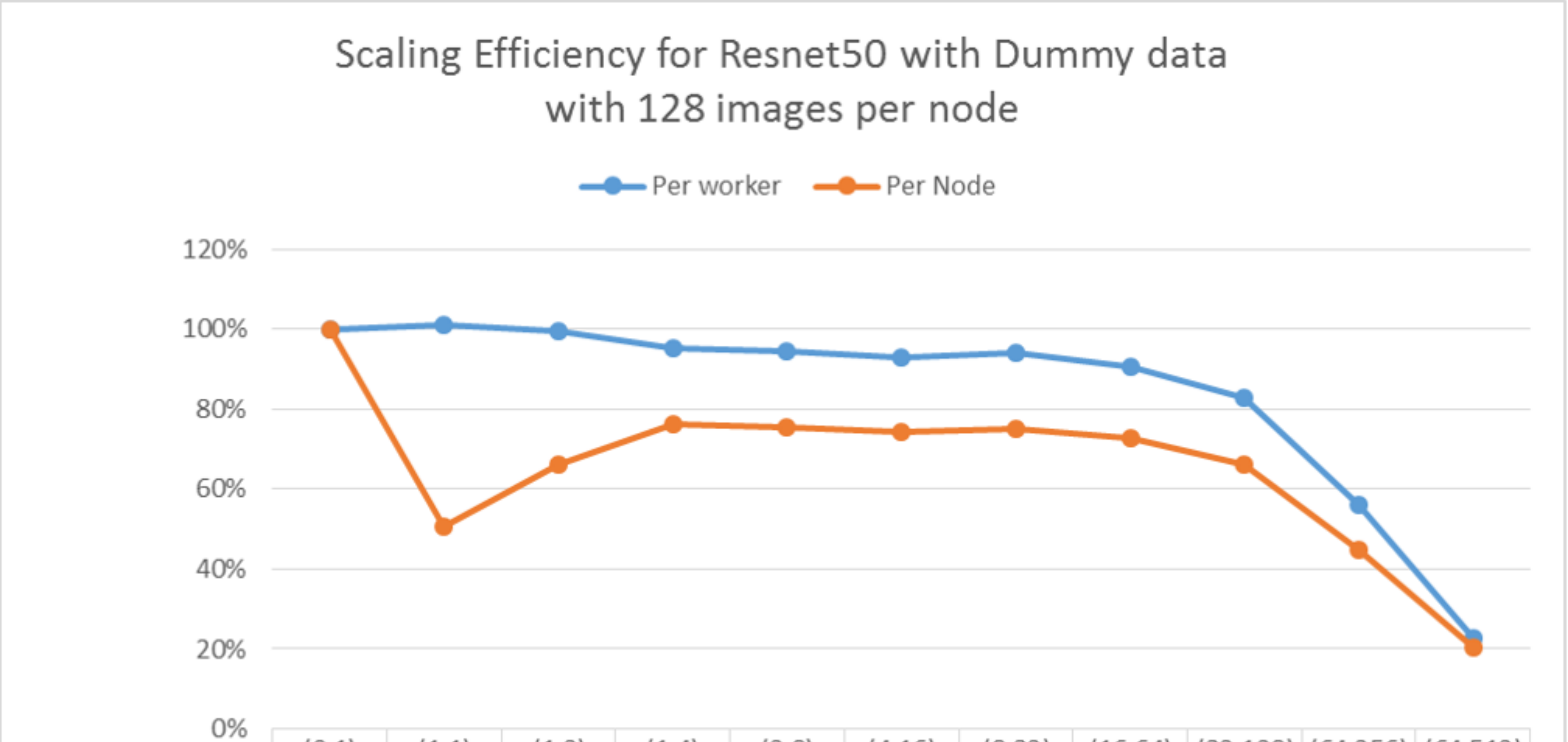
```python
import tensorflow as tf
import numpy

# Parameters
learning_rate = 0.01
training_epochs = 1000
display_step = 50

# Training Data
train_X = numpy.asarray([3.3,4.4,5.5,6.71,6.93,4.168,9.779,6.182,7.59,2.167,
                         7.042,10.791,5.313,7.997,5.654,9.27,3.1])
train_Y = numpy.asarray([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.596,2.53,1.221,
                         2.827,3.465,1.65,2.904,2.42,2.94,1.3])
n_samples = train_X.shape[0]

# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)
# Gradient descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})
```

Distributed Training in TensorFlow

- TensorFlow natively supports Google RPC
  - pros: asynchronous, multi-platform, resilient, integrated
  - cons: not made for HPC, slow, server-client model, initialization on HPC systems painful
- **solution**: use custom TensorFlow ops hook to hook-in *your own* framework



https://www.theregister.co.uk/2015/10/27/another_go_at_remote_objects_google_grpc_hits_beta/



	(0,1)	(1,1)	(1,2)	(1,4)	(2,8)	(4,16)	(8,32)	(16,64)	(32,128)	(64,256)	(64,512)
Per worker	100%	101%	99%	95%	95%	93%	94%	91%	83%	56%	23%
Per Node	100.00%	50.47%	66.29%	76.23%	75.63%	74.43%	75.18%	72.62%	66.29%	44.90%	20.26%

A. Mathuriya, et.al.: *Scaling GRPC Tensorflow on 512 nodes of Cori Supercomputer*

Horovod(-MPI)

- plugin developed by Uber
- works with TensorFlow and Keras (higher level TF abstraction)
- couples communication background thread asynchronously into executed TensorFlow graph
- communication performed using MPI intrinsics (Send/Recv/Bcast/Allreduce)
- works on all platforms with MPI
- can be mixed and matched with mpi4py

```python
import tensorflow as tf
import numpy
import horovod.tensorflow as hvd

hvd.init()

# Parameters
..

# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)
# Gradient descent
global_step = tf.train.get_or_create_global_step()
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
optimizer = hvd.DistributedOptimizer(optimizer)
optimizer = optimizer.minimize(cost, global_step=global_step)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
bcast = hvd.broadcast_global_variables(0)

# Start training
with tf.train.MonitoredTrainingSession() as sess:
    sess.run(init)
    sess.run(bcast)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})
```

Horovod(-MPI)

- plugin developed by Uber
- works with TensorFlow and Keras (higher level TF abstraction)
- couples communication background thread asynchronously into executed TensorFlow graph
- communication performed using MPI intrinsics (Send/Recv/Bcast/Allreduce)
- works on all platforms with MPI
- can be mixed and matched with mpi4py

```python
import tensorflow as tf
import numpy
import horovod.tensorflow as hvd

hvd.init()

# Parameters
..

# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)
# Gradient descent
global_step = tf.train.get_or_create_global_step()
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
optimizer = hvd.DistributedOptimizer(optimizer)
optimizer = optimizer.minimize(cost, global_step=global_step)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
bcast = hvd.broadcast_global_variables(0)

# Start training
with tf.train.MonitoredTrainingSession() as sess:
    sess.run(init)
    sess.run(bcast)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})
```

limited changes to source code

Other Horovod Variants

- Horovod-MLSL by Intel
  - uses MLSL instead of MPI
  - wrappers are source code compatible with Horovod-MPI
  - can employ more than one BG process to progress communication
  - on machines without `MPI_Comm_spawn()` support,
    MLSL servers need to be launched manually
- Horovod-NCCL
  - uses nvidia NCCL 2 for GPU2GPU communication and efficient collectives
  - aims at improving performance for HPC systems with fat (multi-GPU) nodes

CPE ML Plugin

- plugin similar to Horovod
- different syntax
- more LOC need to be changed
- support for other frameworks than TensorFlow
- support for sophisticated features such as pipelining, multi-threaded communication, solver cool-down
- only available on Cray hardware

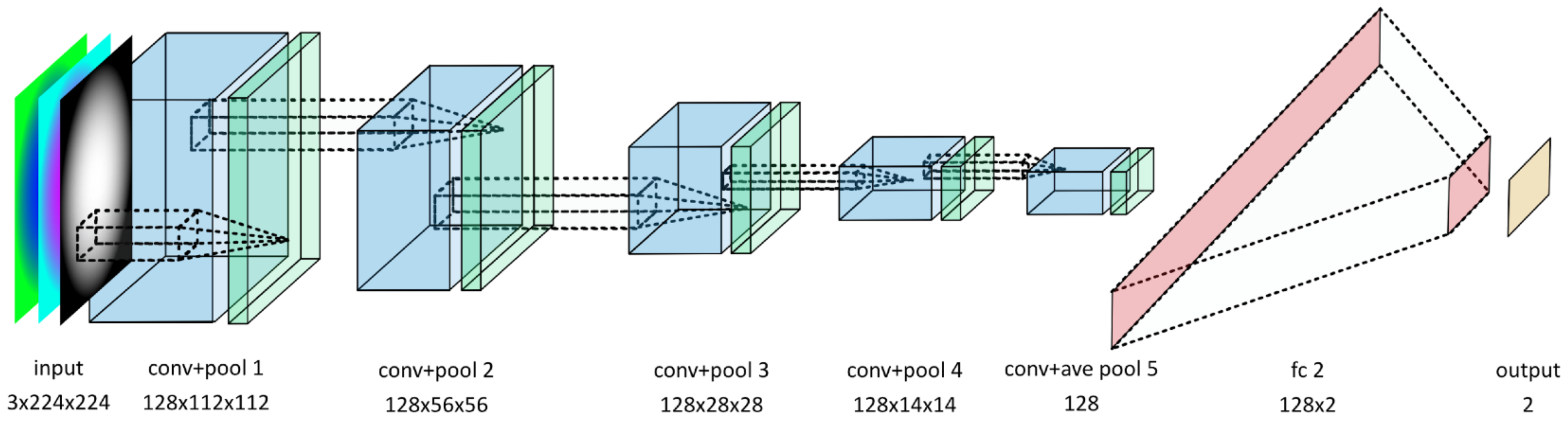
```
import tensorflow as tf
import numpy
import ml_comm as mc

mc.init(1, 1, 5*1024*1024, "tensorflow")

# Parameters
..

# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)
# Gradient descent
global_step = tf.train.get_or_create_global_step()
optimizer = tf.train.GradientDescentOptimizer(learning_rate)

# Split global and local reduction
grads_and_vars = optimizer.compute_gradients(cost)

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)]

optimizer = optimizer.apply_gradients(gs_and_vs, global_step=global_step)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
bcast = ..

# Start training
..
```

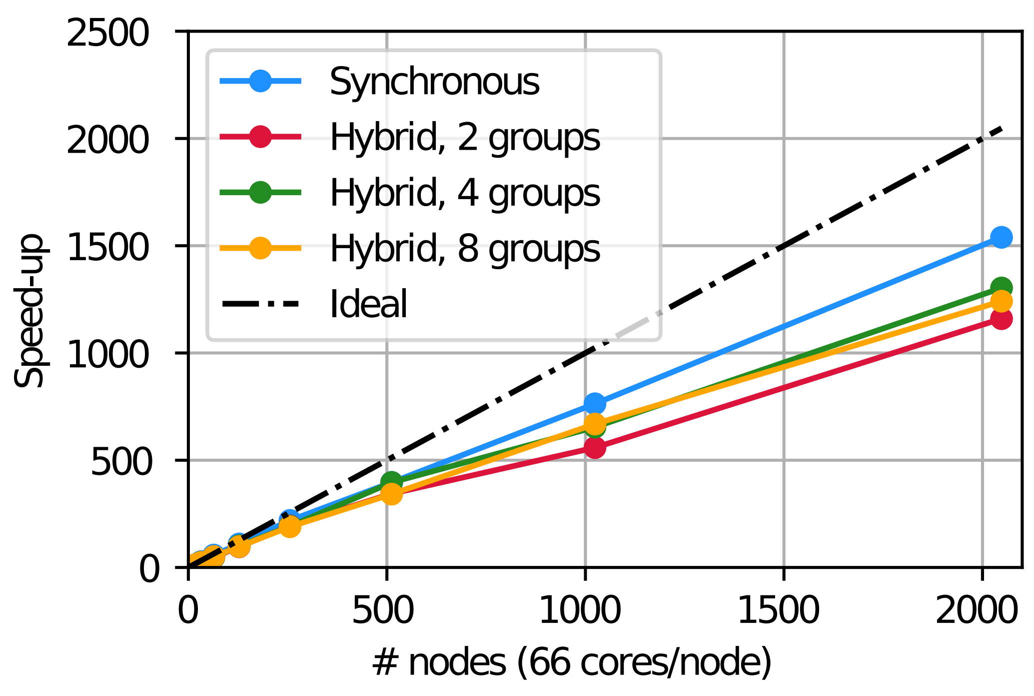
HPC Systems

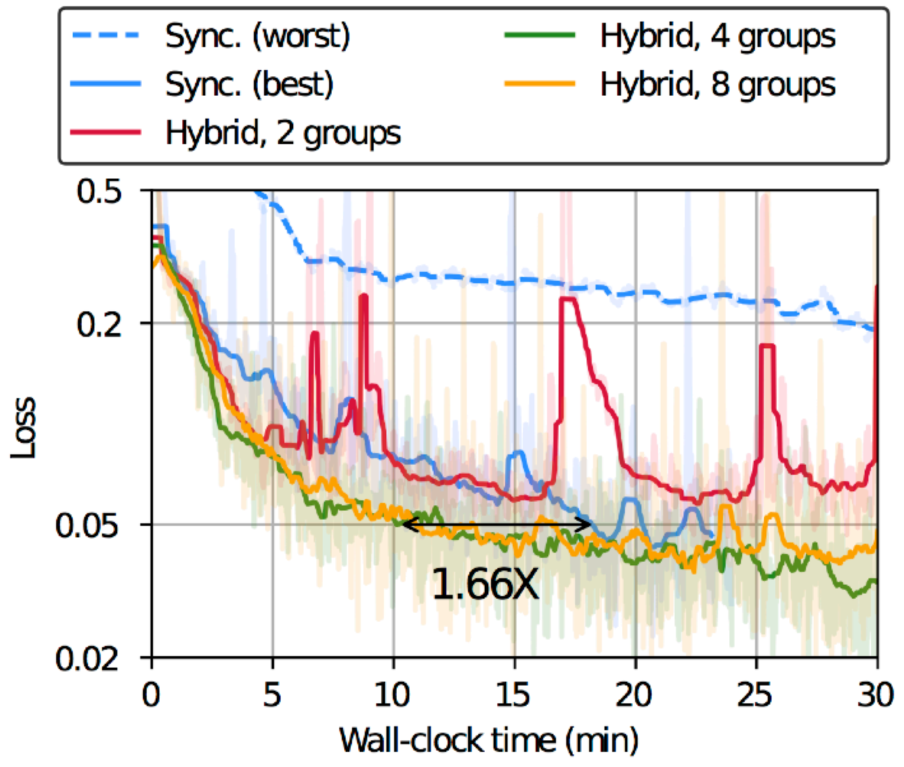
- Cori-KNL (NERSC, XC40)
  - 9688 Intel Xeon Phi 7250 (KNL)
  - 68 cores@1.4Ghz, AVX512
  - 96GB DDR and 16GB on-package MCDRAM
- Piz Daint (CSCS, XC50)
  - 5320 CPU+GPU
  - Xeon E5-2695v3 and Tesla P100
  - 64 GB DDR, 16GB HBM2

Deep Learning Models

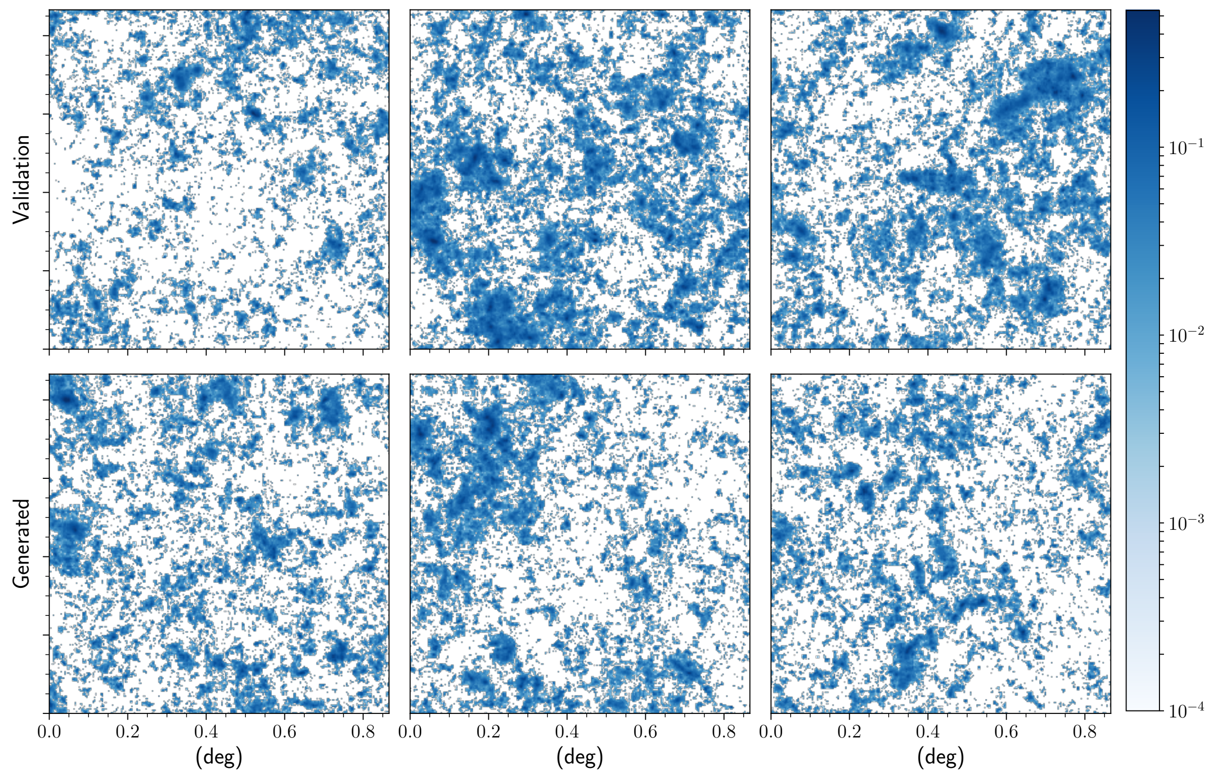
- HEP-CNN
  - binary collider event-classification
  - 7-layer CNN
  - lightweight: sparse or small layers only
- CosmoGAN
  - generates cosmology mass-maps
  - 2x5-layer DC-GAN
  - underlying architecture for other scientific GAN use-cases at NERSC

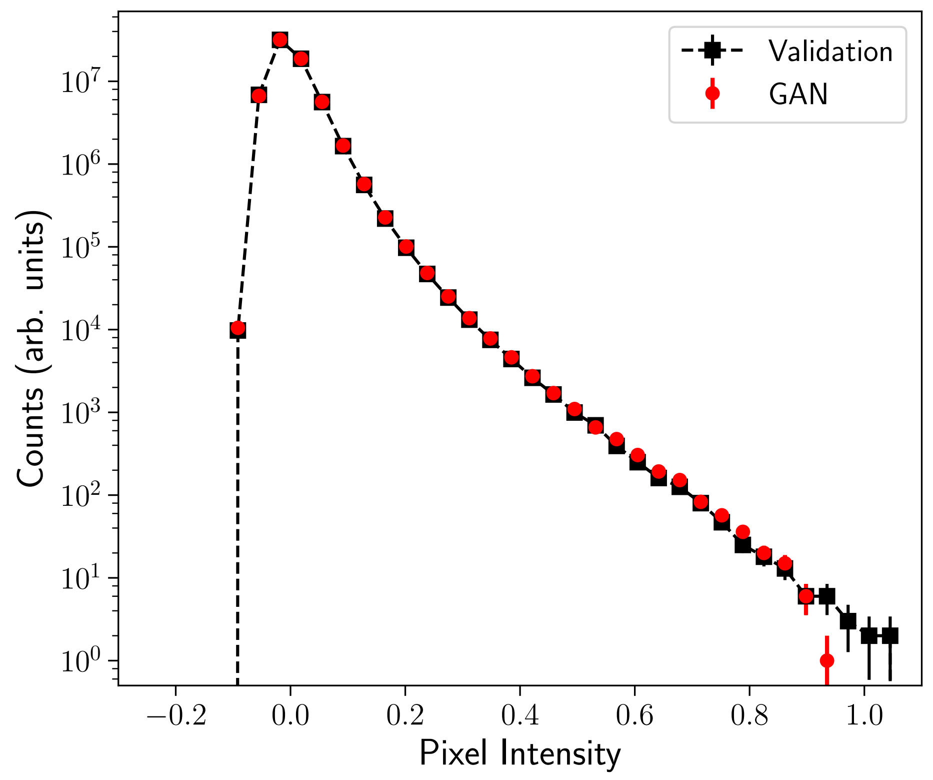






T. Kurth et al.: *Deep Learning at 15 PF* (2017)

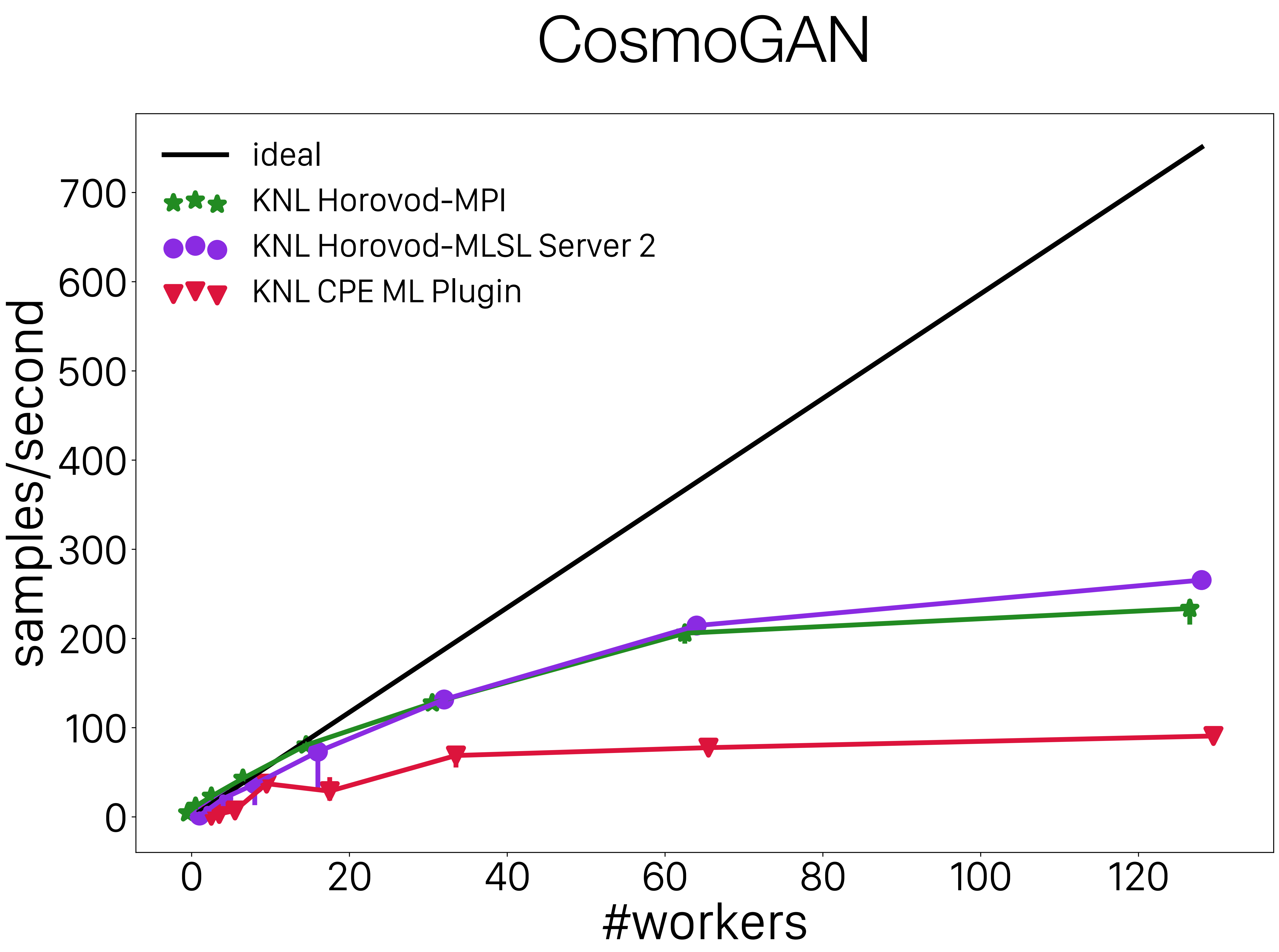




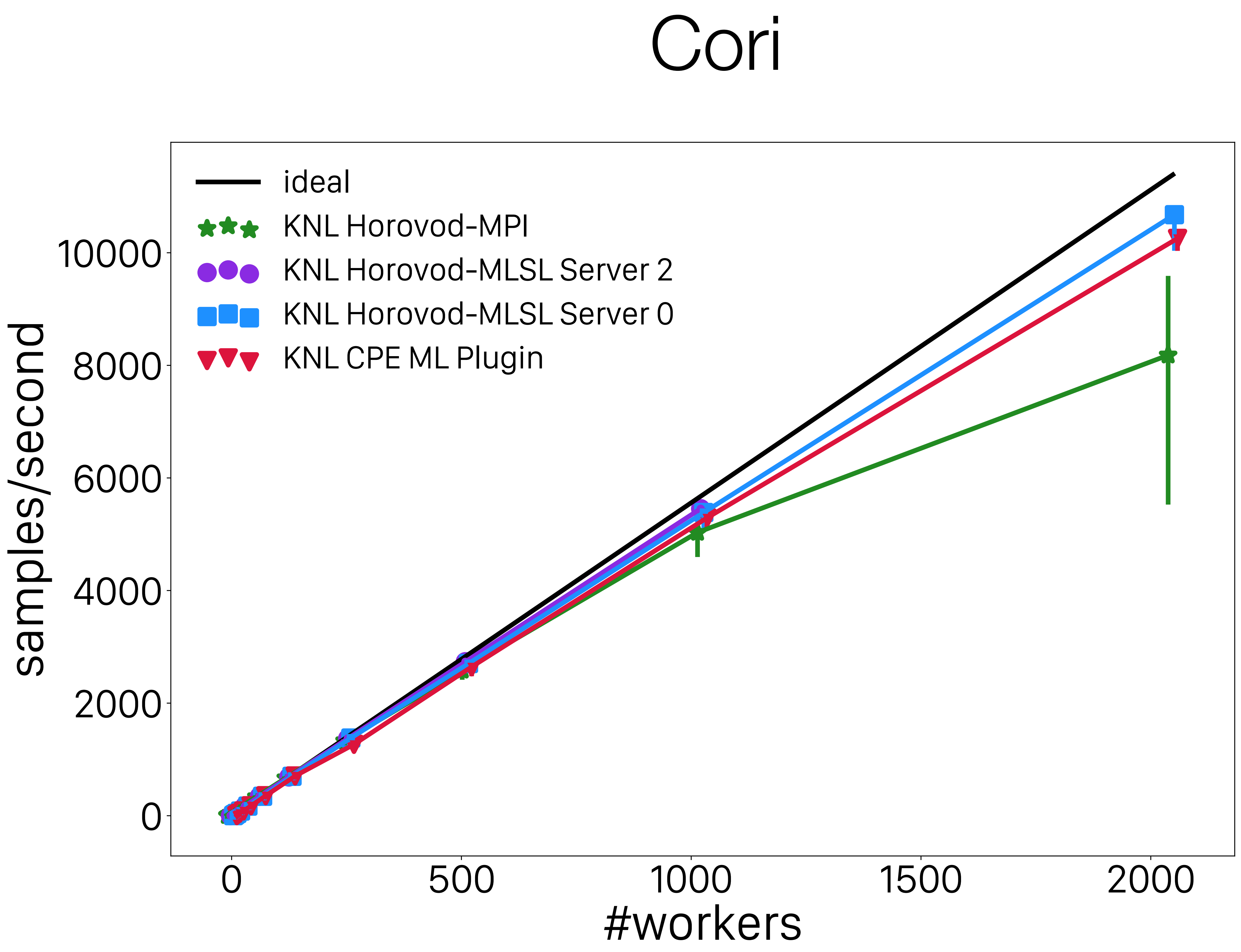
M. Mustafa et al.: *Creating Virtual Universes Using Generative Adversarial Networks* (2017)

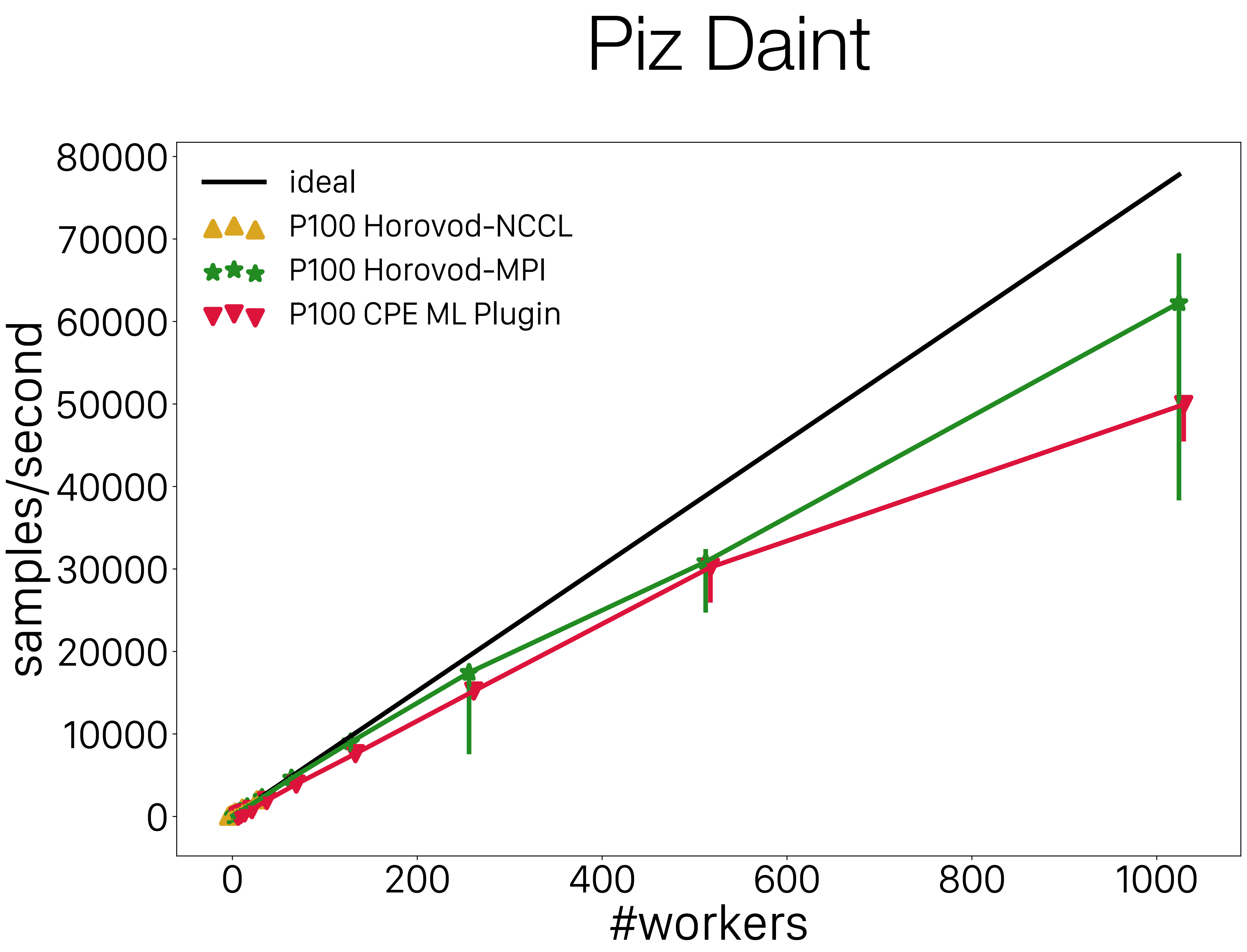
Strong Scaling on Cori

- scale from 1 node w/ batch-size 64 to 64 nodes w/ batch-size 1
- efficient performance scaling stops around 8 or 16 nodes for all frameworks
- can be applied for small-scale parallelization w/o as it requires no additional HPO
- not the favorable mode to scale deep learning training



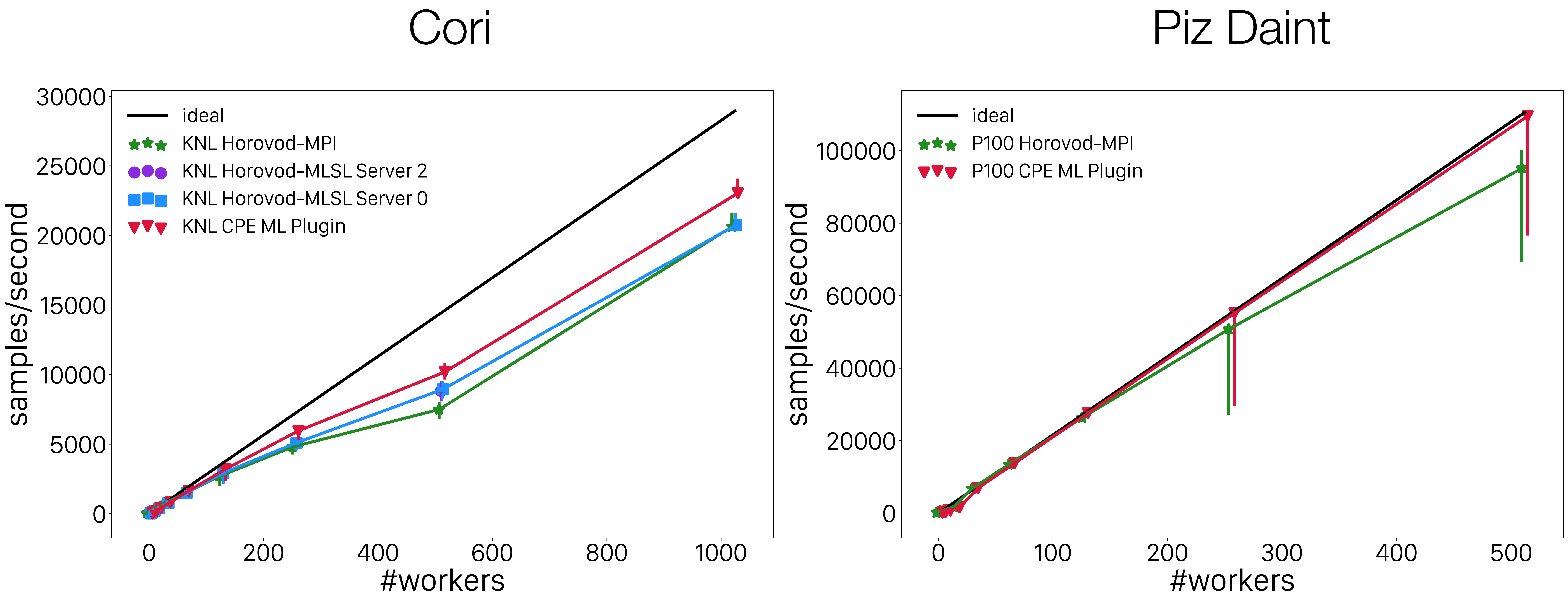
Weak Scaling CosmoGAN





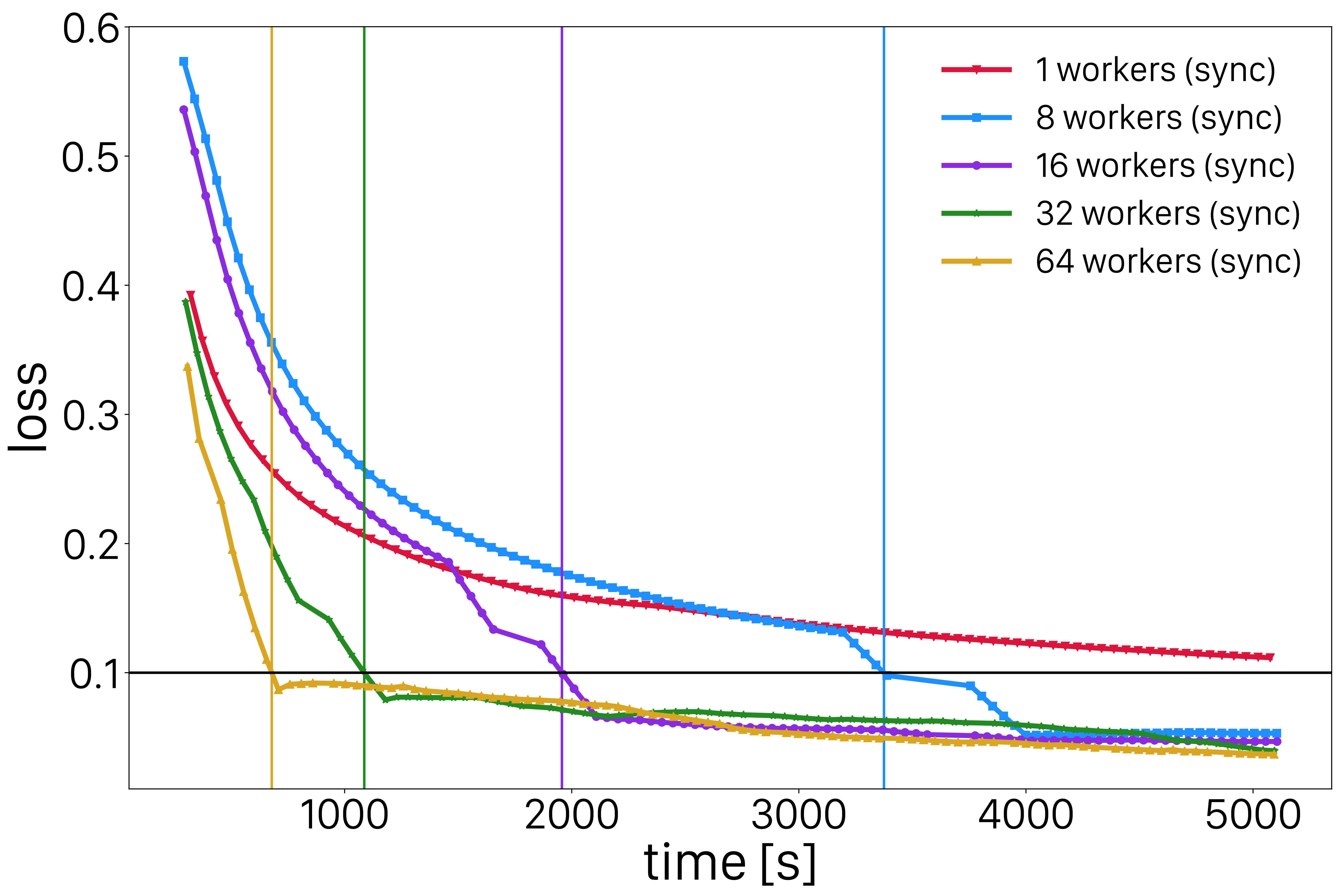
- good scaling efficiency of all frameworks
- Horovod-MPI: high variability but outperforms CPE ML on Piz Daint (better overlap, more threads might help)

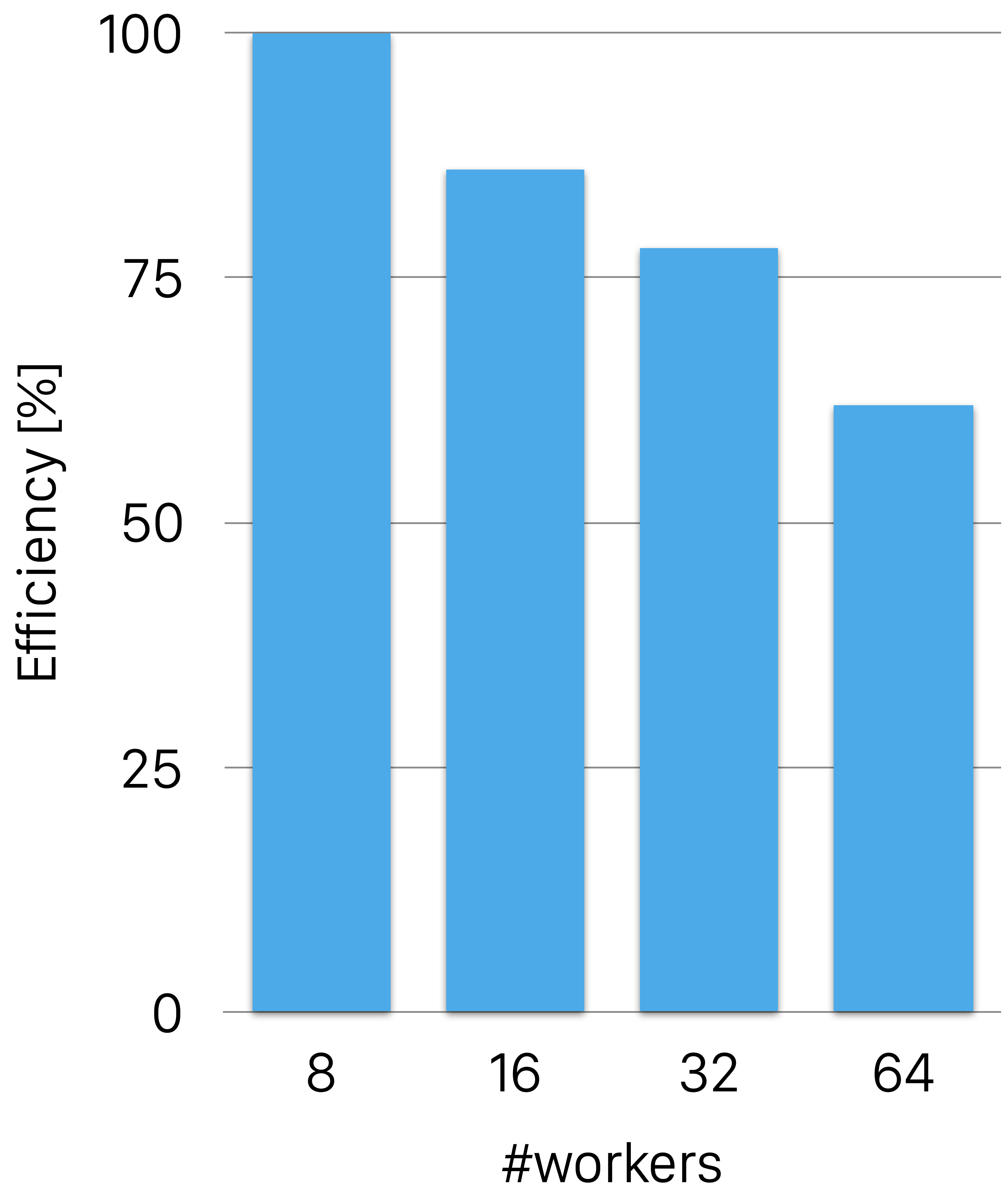
Weak Scaling HEP-CNN



- network latency and jitter sensitive, ~40 ms (FW+BW)/sample (KNL)
- on Cori similar efficiency for all frameworks, on Piz Daint CPE significantly better than Horovod-MPI

Convergence (Time to Solution)





- result with plain ADAM optimizer

Conclusion

- scalable deep learning solutions can be implemented in high productivity frameworks such as TensorFlow
- all tested frameworks integrate beautifully into TensorFlow and outperform GRPC
- performance: Horovod-MLSL, CPE ML Plugin
- portability: Horovod-MPI, Horovod-MLSL
- fat nodes? try Horovod-NCCL but has some limitations still
- missing in all frameworks: hooks for model parallelism