High-performance Data Analytics
Basic concepts of distributed deep learning

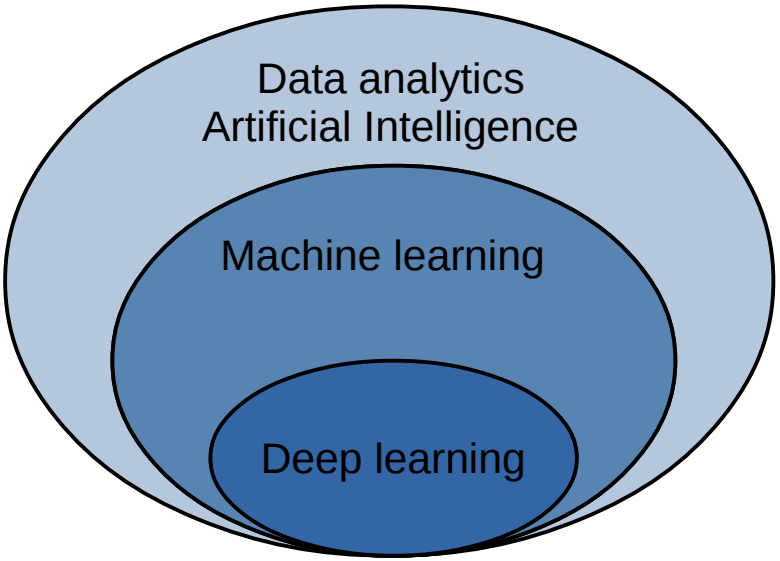


Image adapted from: arXiv:1903.11314

Markus Rampp (markus.rampp@mpcdf.mpg.de)

Andreas Marek (andreas.marek@mpcdf.mpg.de)

Max Planck Computing and Data Facility (MPCDF)

BiGmax Summer School, Platja d'Aro/Spain, Sep 9-13, 2019

BiG max MAX PLANCK RESEARCH NETWORK on big-data-driven materials science

Acknowledgments:

- IPAM @UCLA: Long Program "Science at Extreme Scales: Where Big Data Meets Large-Scale Computing", 2018
- BiGmax
- L. Stanisic, N. Fabas, G. DiBernardo, J. Kennedy (MPCDF)

Distributed Deep Learning: Why bother ?

- we use high-level frameworks like TensorFlow/Keras, PyTorch, … anyway ?
  → welcome to the jungle!

- applications in basic physics? is there large-scale data?
  →

Distributed Deep Learning: Why bother ?

- we use high-level frameworks like TensorFlow/Keras, PyTorch, … anyway ?
  → welcome to the jungle!

- applications in basic physics? is there large-scale data?
  →

Aims and claims of this *introductory* lecture:

→ sketch fundamentals of parallelizing artificial neural network (ANN) computations
→ understand challenges and limitations
→ make the connection to high-performance computing (HPC)
→ provide orientation in the (rapidly evolving) jungle of methodologies and software
→ starting point for mastering non-standard applications

→ this lecture is *not:*
- an introduction to deep learning: familiarity with the basics of ANN is assumed
- a TensorFlow tutorial
- specific to materials science
- presenting novel concepts or ideas

ANN basics

- **“architecture”** of an ANN (MLP)

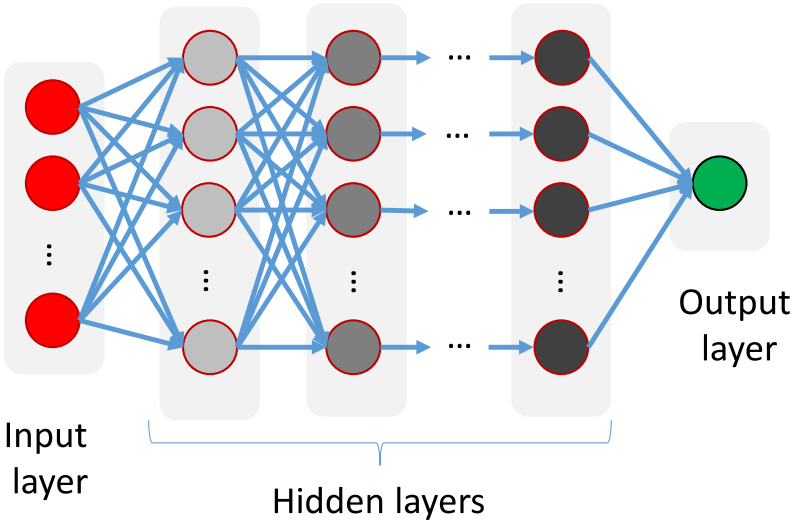


image: arXiv:1903.11314

- **“training”**: optimization via stochastic gradient descent (SGD), taking (small, |*B*|=1) batches of data (*B*) to iteratively update the weights *w* in order to minimize the prediction error (“loss” function)

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

→ time consuming, requires HPC => exploit parallelism

- **“inference”**: use the “trained model” $\{w_{t=final}\}$ as interpolator for new (yet unseen) data

Types of parallelism in ANN

Data parallelism:

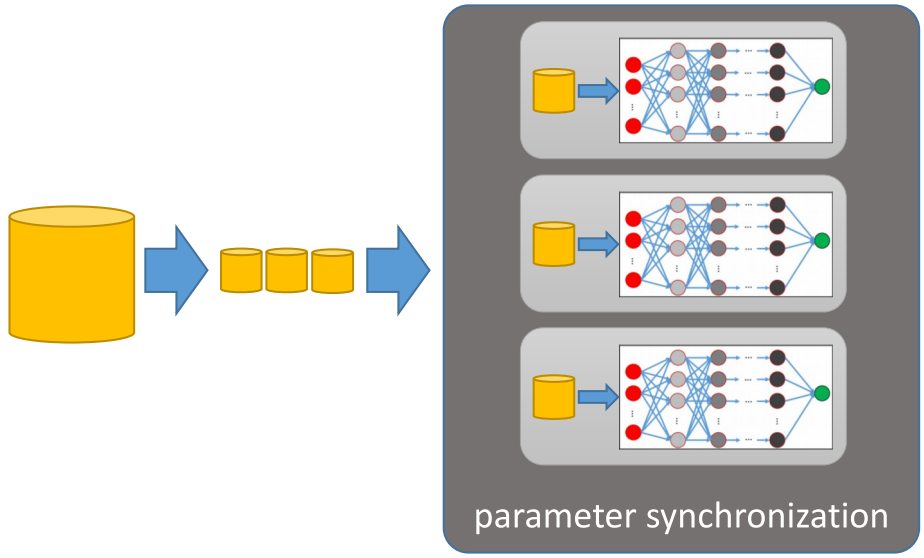


image: arXiv:1903.11314

- model (all ANN parameters) is ***replicated*** across all “workers” (PEs: CPUs, GPUs)
- training data is divided across workers
  => speedup with increasing number of workers expected
  => synchronization mechanism required
- limitations: entire model has to fit into memory
  enough training data to keep multiple workers busy
- conceptually straightforward (corresponds to a *domain-cloning* concept in HPC)
- most popular in prototypical ANN application domains (Facebook et al.) where huge amounts of training data are available

Types of parallelism in ANN

Model parallelism:

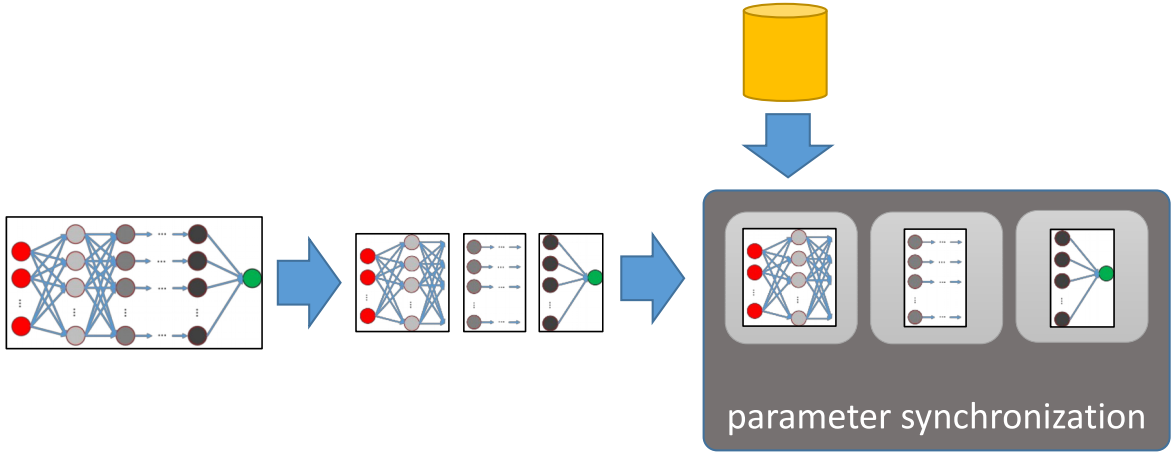


image: arXiv:1903.11314

- model (all parameters) is ***divided***
  across all workers (CPUs, GPUs, nodes, …)
  => speedup with increasing number of workers expected (*training only*)
  => memory requirements per worker/node are relaxed
  => synchronization mechanism required

- limitations: how to achieve speedup in inference stage ?

- conceptually more challenging (corresponds to a *domain-decomposition* concept in HPC)

- not yet commonly supported/applied, but necessary for to fit huge models in memory of commodity HPC clusters

Types of parallelism in ANN

+ Hybrid parallelism: combination of model and data parallelism
+ ...

+ Hyperparameter optimization:

- run many independent trainings of the same network to tune network hyperparameters (mini-batch size, number of epochs, learning rate, ...)
- conceptually trivial (*embarrasingly parallel,* formally 100% parallel efficiency)
- to be practically efficient requires good optimization strategies and workflow management

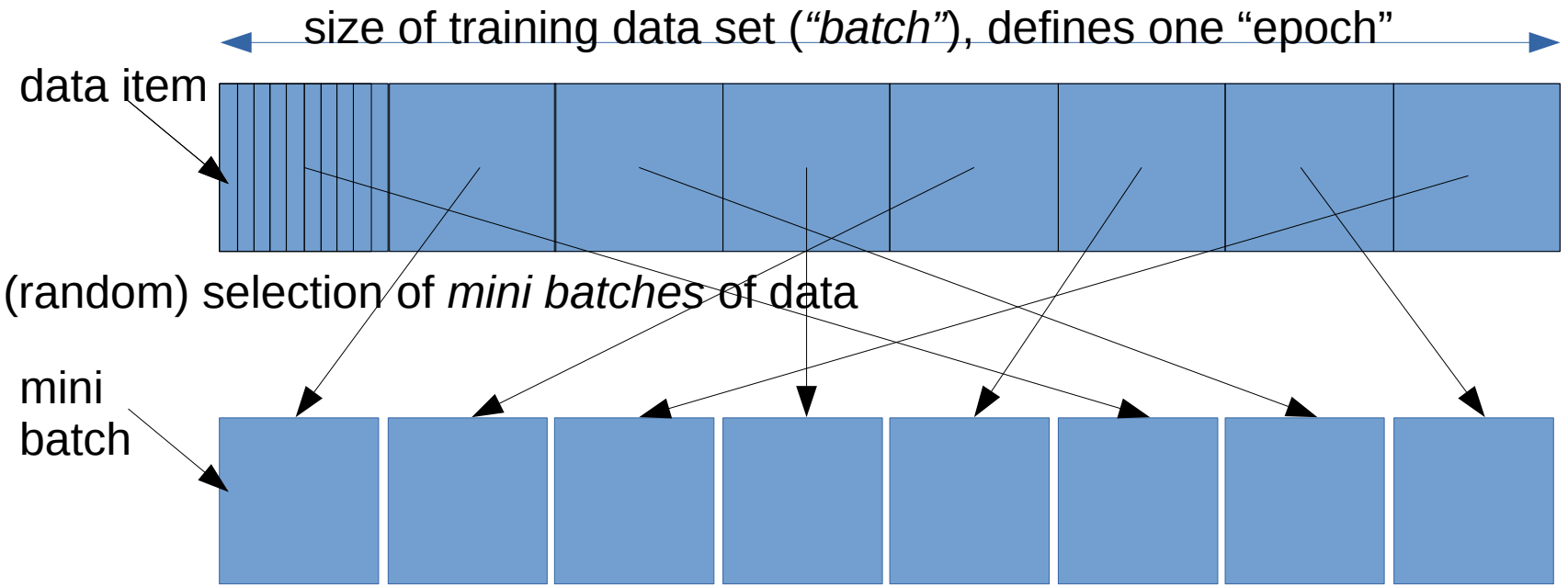
→ software tool Hyperopt: Distributed Asynchronous Hyper-parameter Optimization (https://github.com/hyperopt/hyperopt)

→ implemented on MPCDF HPC systems (slurm integration, mongoDB)

Data-parallelism in ANN training:

- “strong” scaling vs. “weak” scaling
- A basic example with Tensorflow/Keras/Horovod

ANN training: terminology

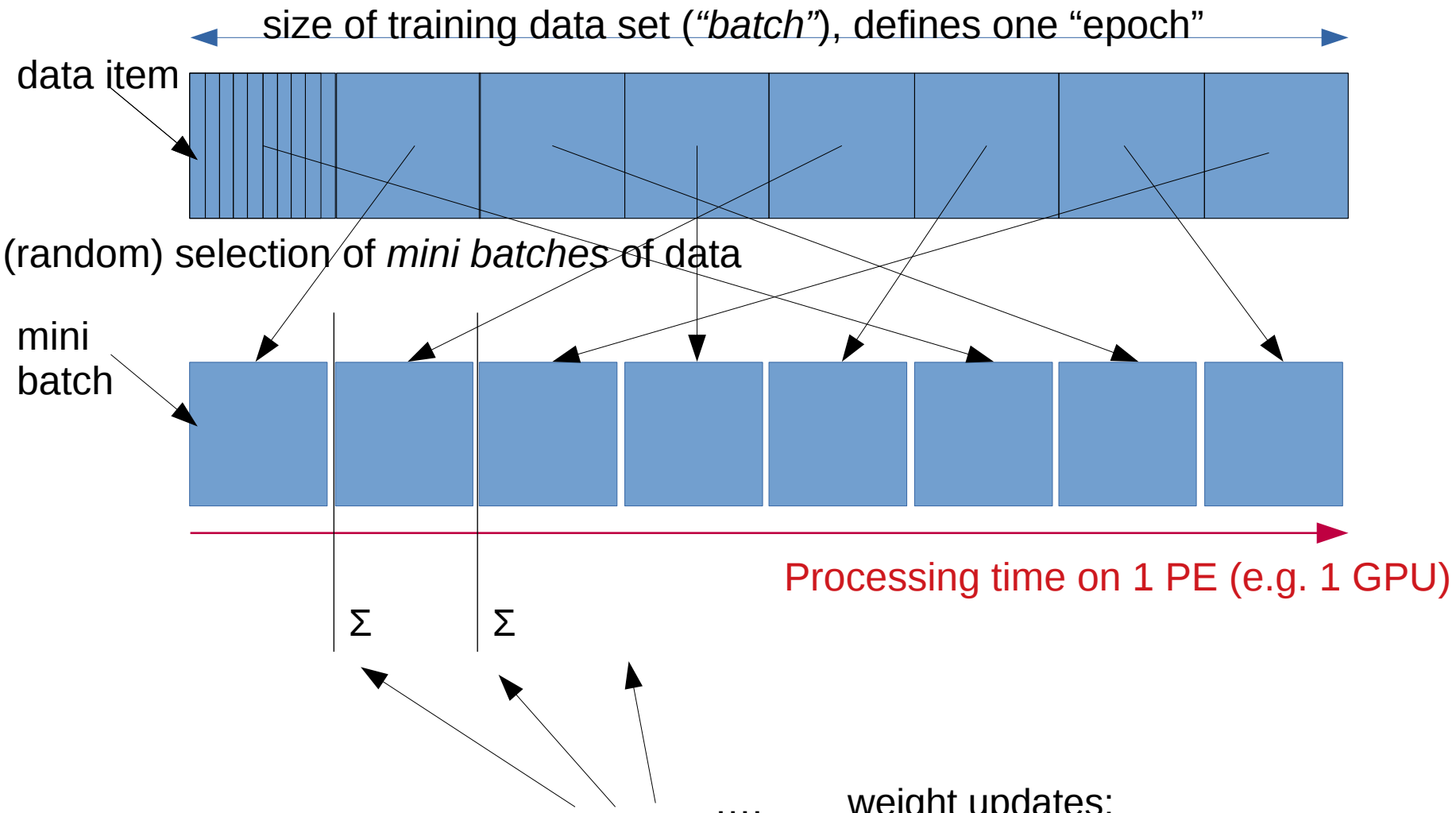


Terminology:
Batch: amount of data items processed for each model update

Batch Gradient Descent:	batch size = size of training data set
Stochastic Gradient Descent:	batch size = 1 (data item)
Mini-Batch Gradient Descent:	1 < batch size < size of training set typically: 128, 256, …

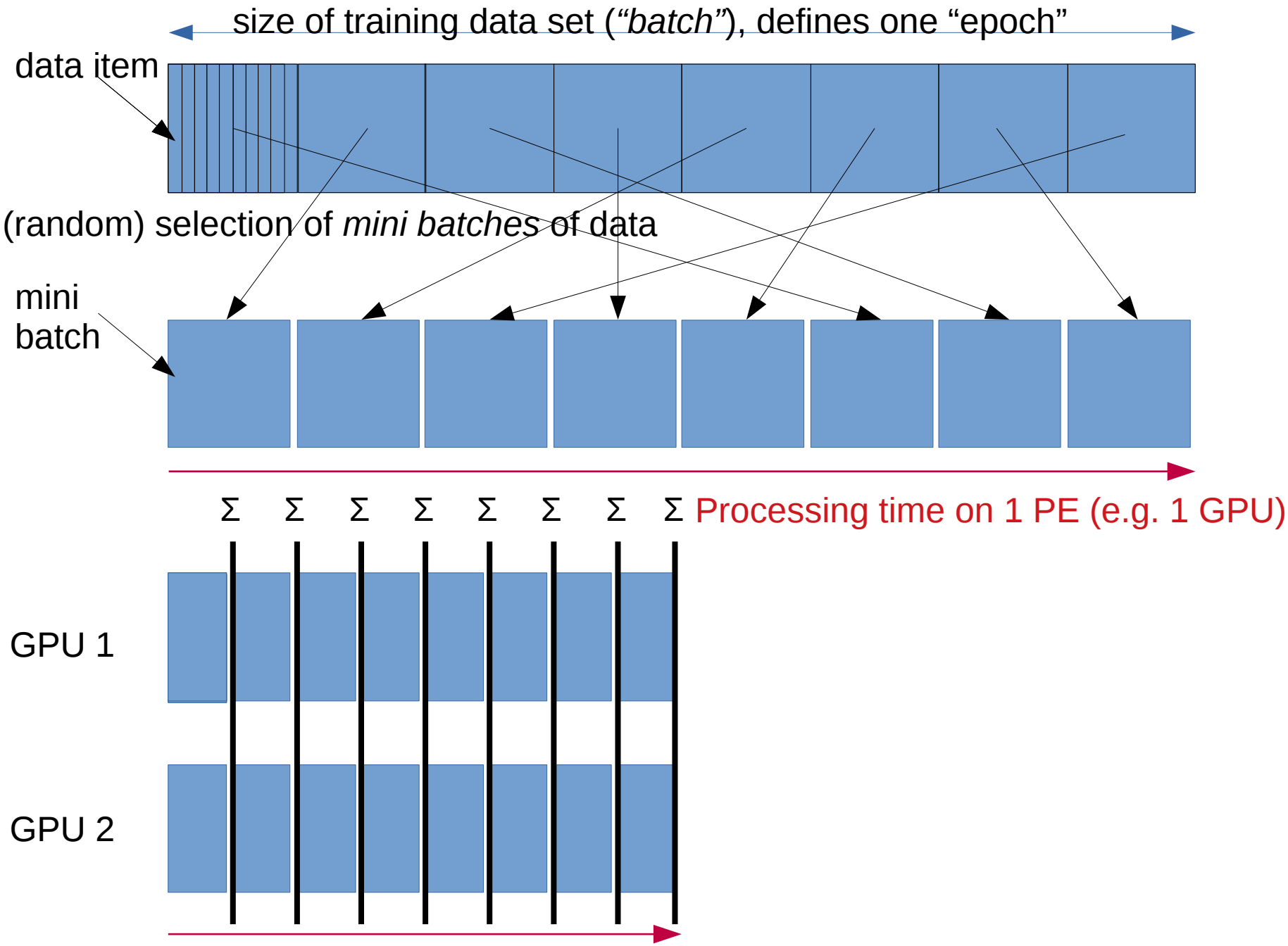
→ size of mini batch determines convergence properties and model performance (“generalizability”)

ANN training: data parallelism

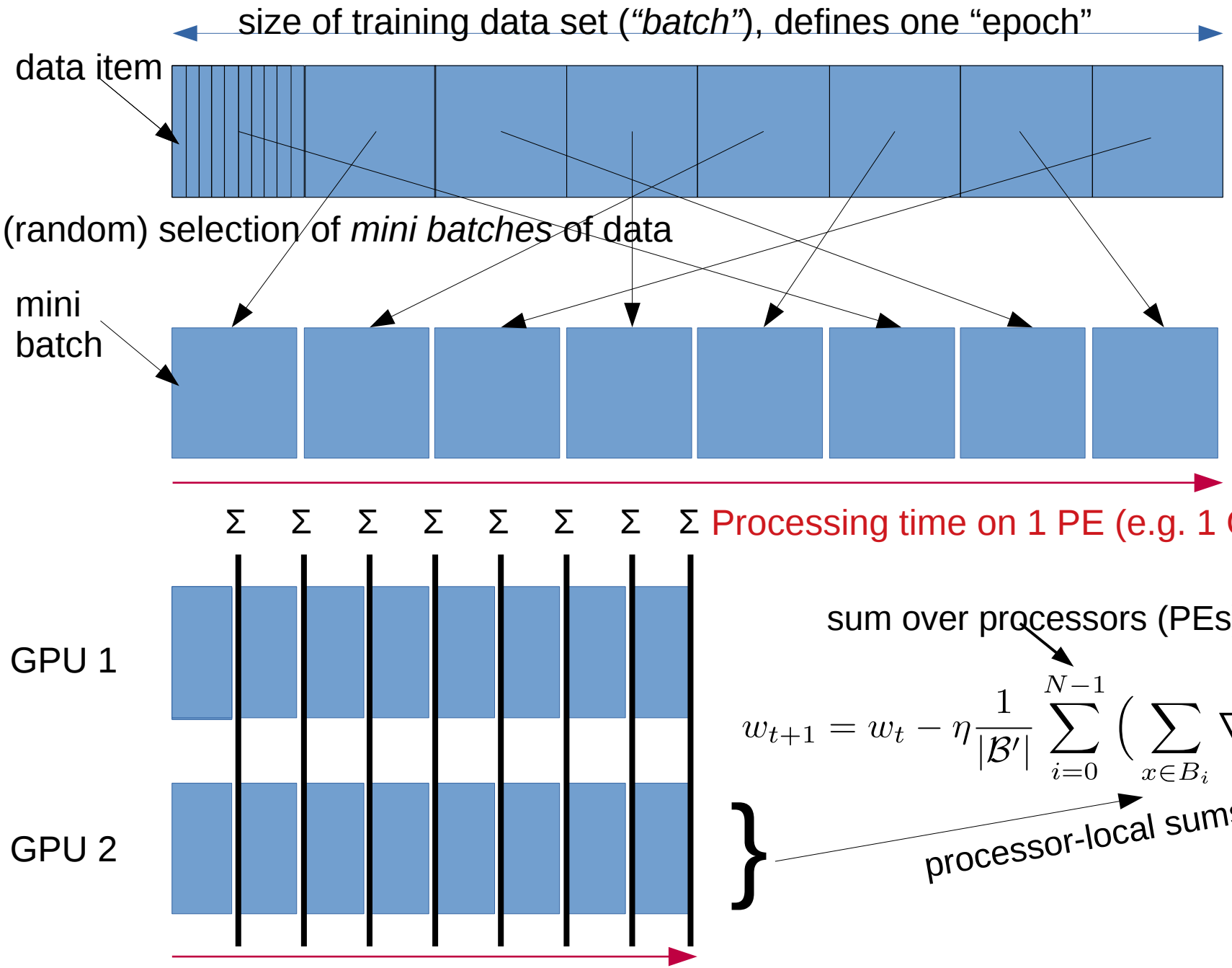


$$w_{t+1} = w_t - \eta \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} \nabla l(x, w_t)$$

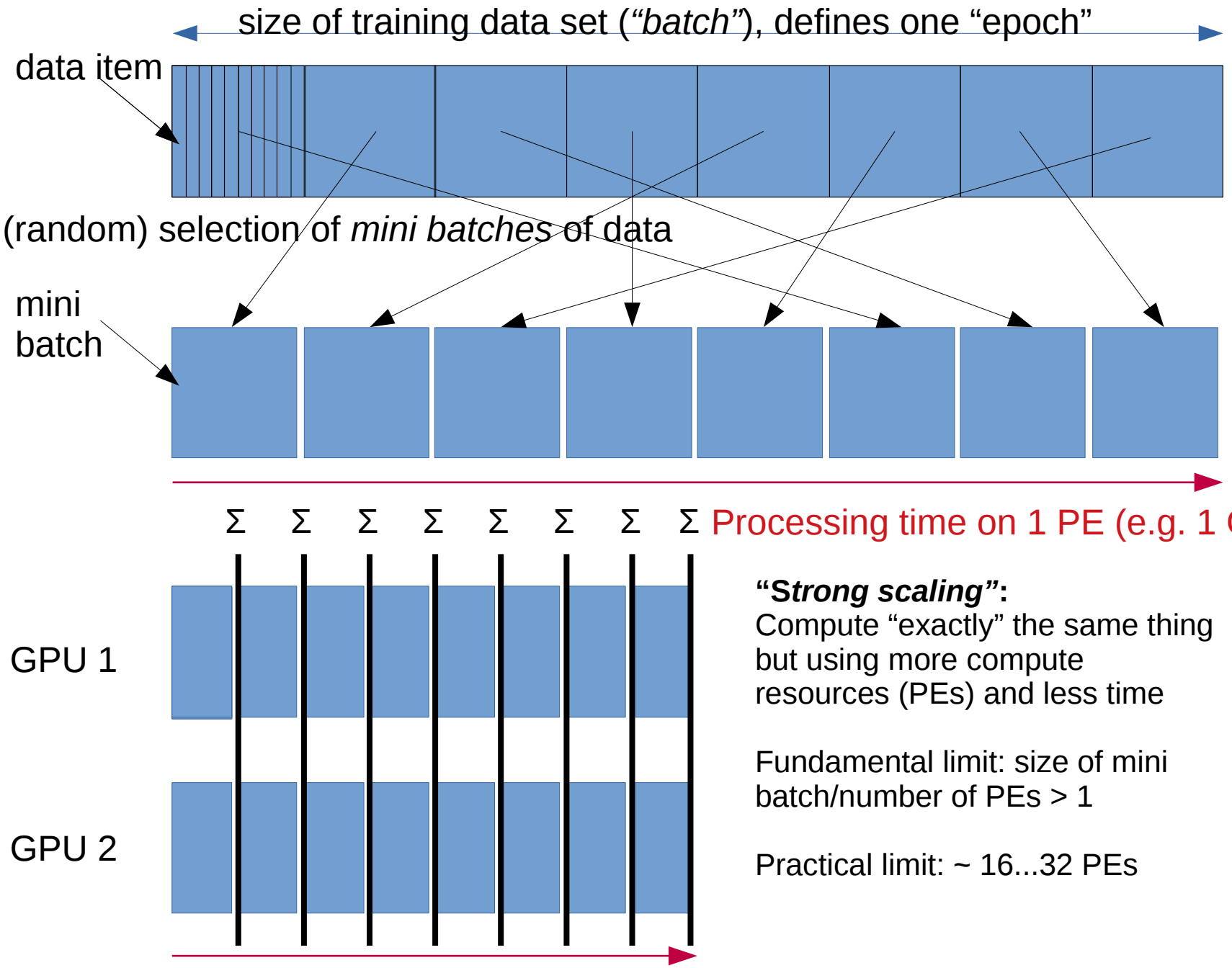
ANN training: data parallelism



ANN training: data parallelism



ANN training: data parallelism

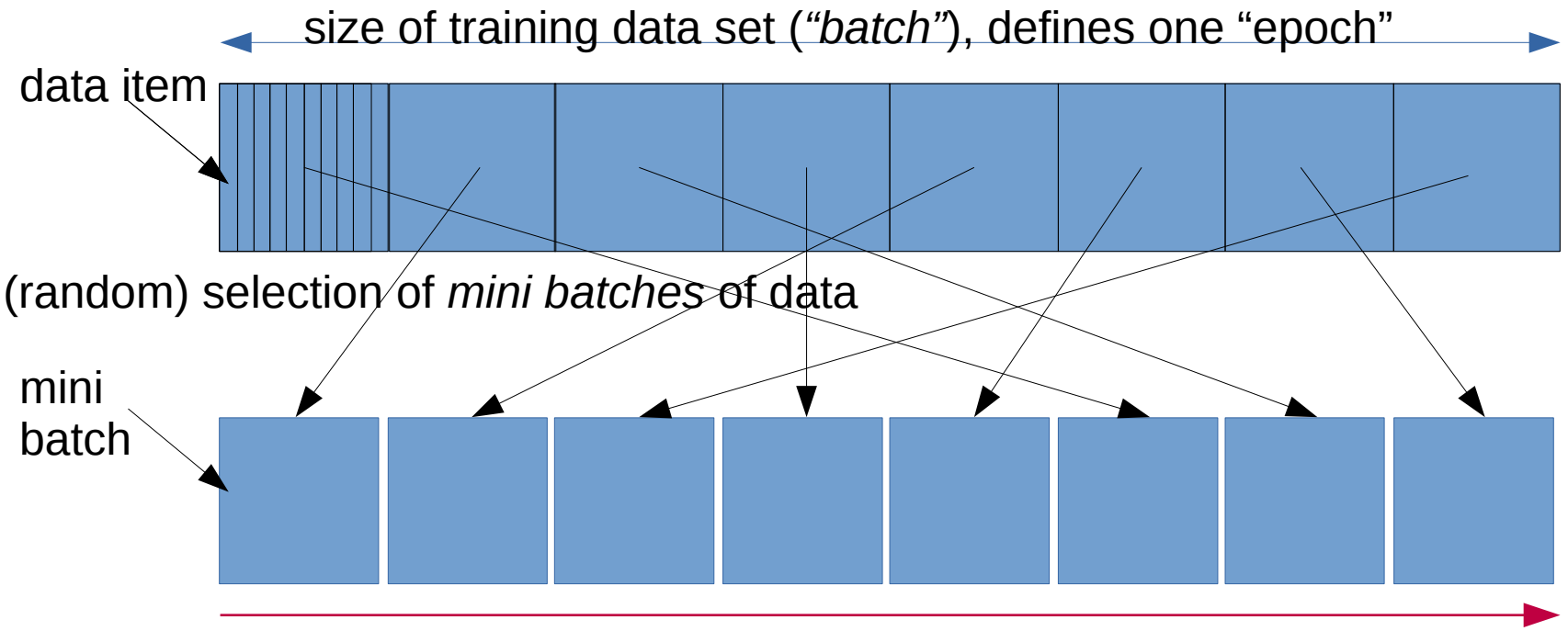


“*Strong scaling*”:
Compute “exactly” the same thing but using more compute resources (PEs) and less time

Fundamental limit: size of mini batch/number of PEs > 1

Practical limit: ~ 16...32 PEs

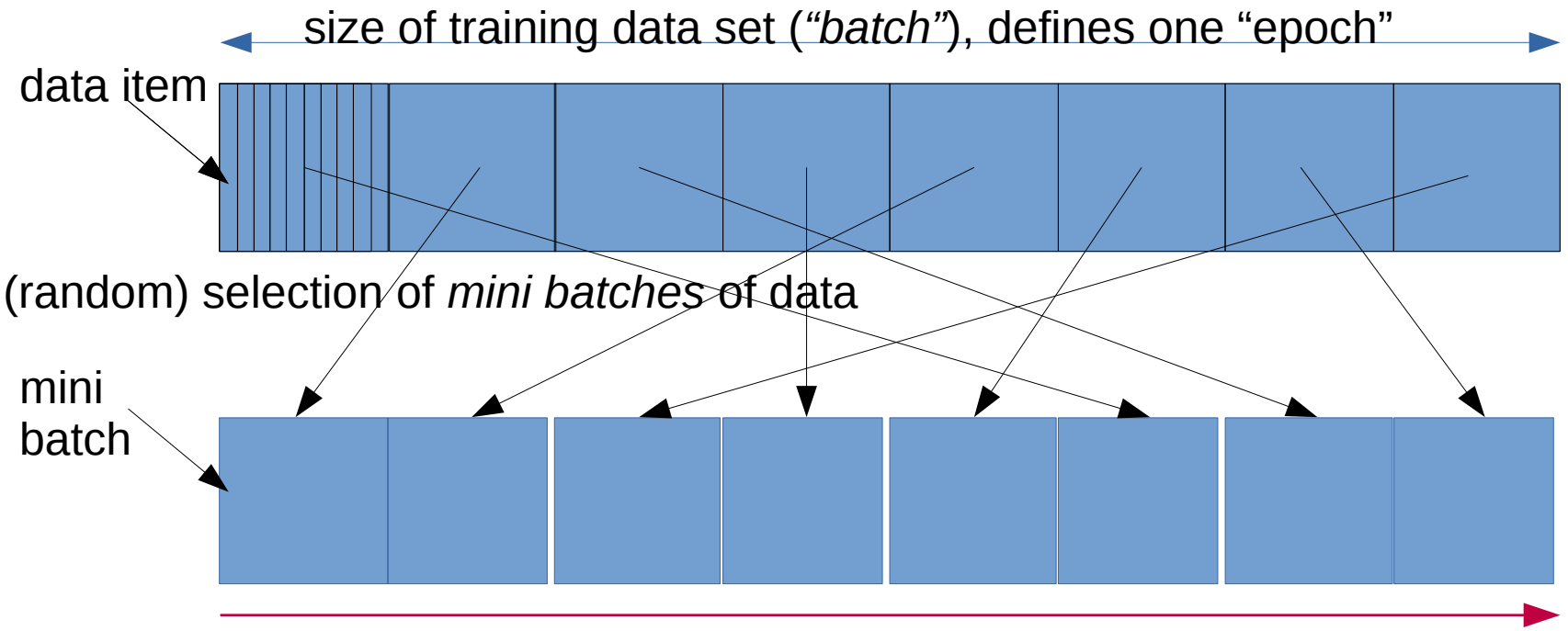
ANN training: data parallelism

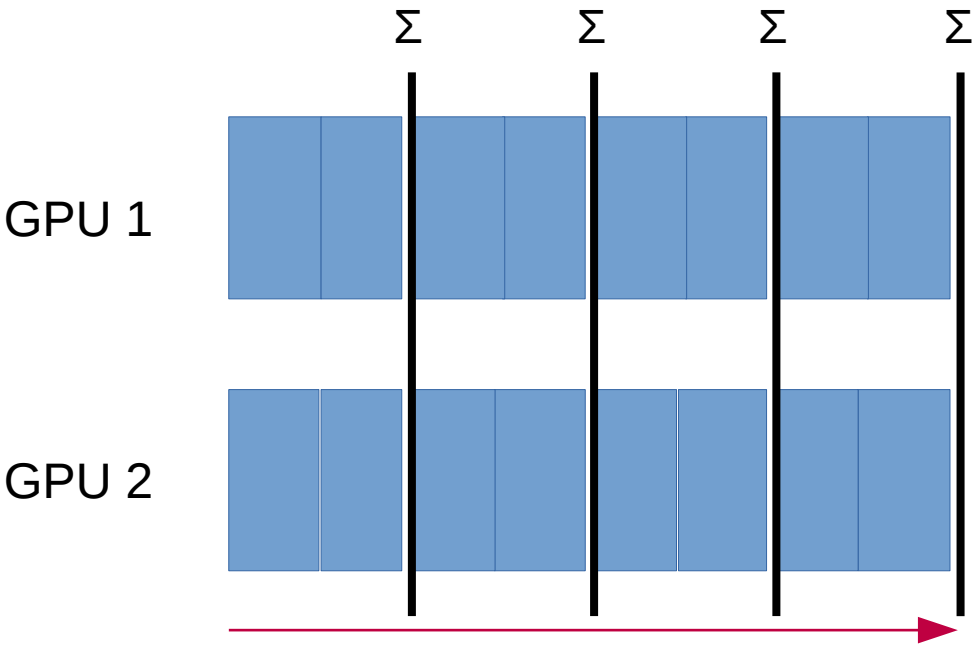


- communication & synchronization
- communication/computation ratio increases with number of PEs

=> parallelization overhead may dominate at large scale

ANN training: data parallelism



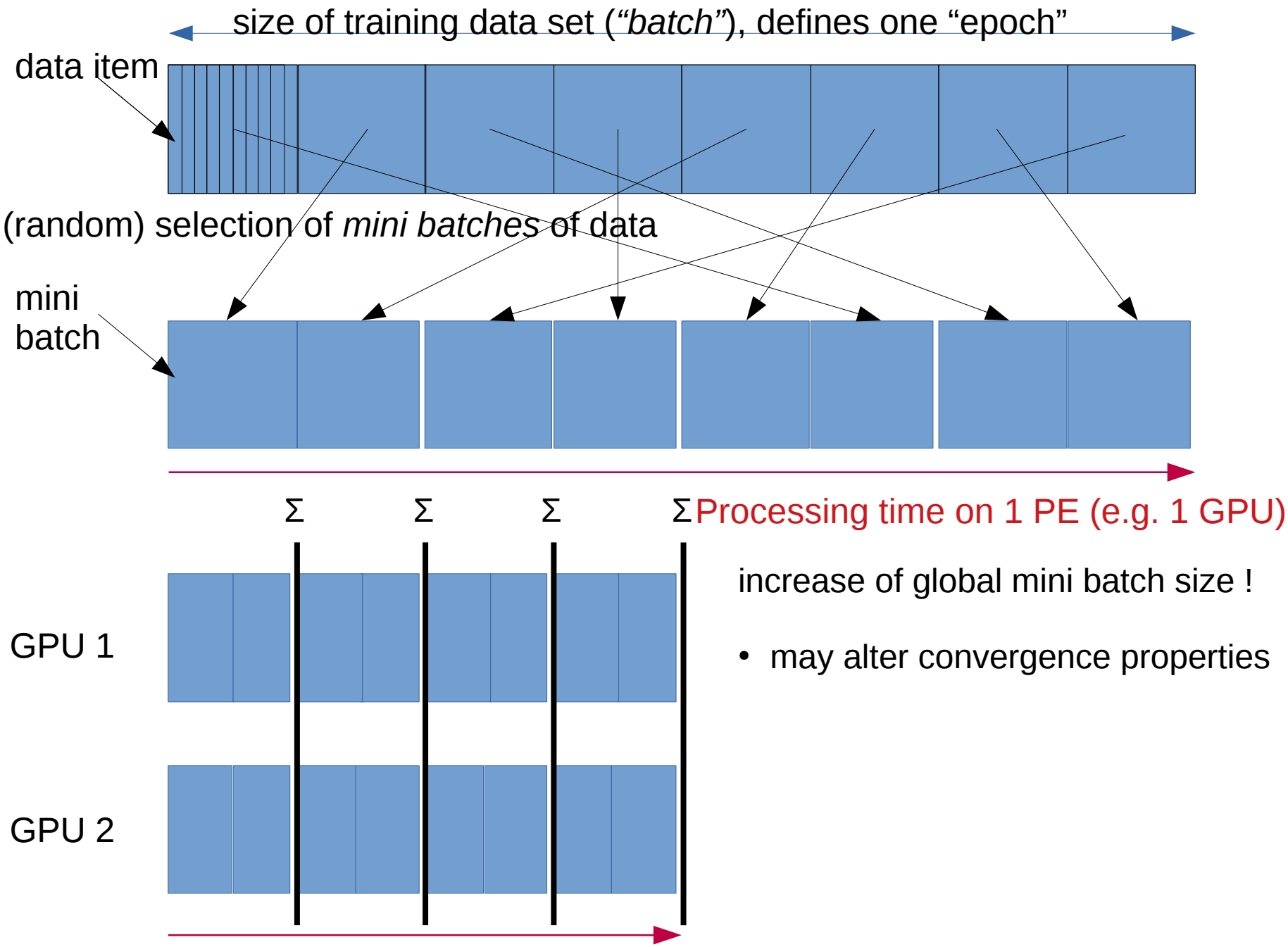


***“Weak scaling”*:**
Keep the size of the PE-local datasets constant(*) while increasing the number of PEs → “Large mini batch SGD”

Fundamental limit: size of entire data set/number of PEs > 1

* effective increase of mini batch size is compensated by a scaling of the learning rate to maintain convergence properties (arXiv:1706.02677)

ANN training: data parallelism



increase of global mini batch size !

- may alter convergence properties

Large mini-batch SGD has become most popular (weak scaling is easier to achieve than strong scaling: less frequent communication and synchronization) but changes the statistical properties (convergence, generalizability) of the algorithm!

→ consistency/reproducibility? (trained model depends on size of the compute cluster!)

Linear scaling rule (Goyal et al. arXiv:1706.02677)

$$w_{t+k} = w_t - \eta \frac{1}{|B_j|} \sum_{j<k} \sum_{x \in B_j} \nabla l(x, w_{t+j})$$

k steps with data size $|B_j|$ and learning rate η

<≈>

1 step with data size $|B|=k*|B_j|$ and learning rate $k*\eta$

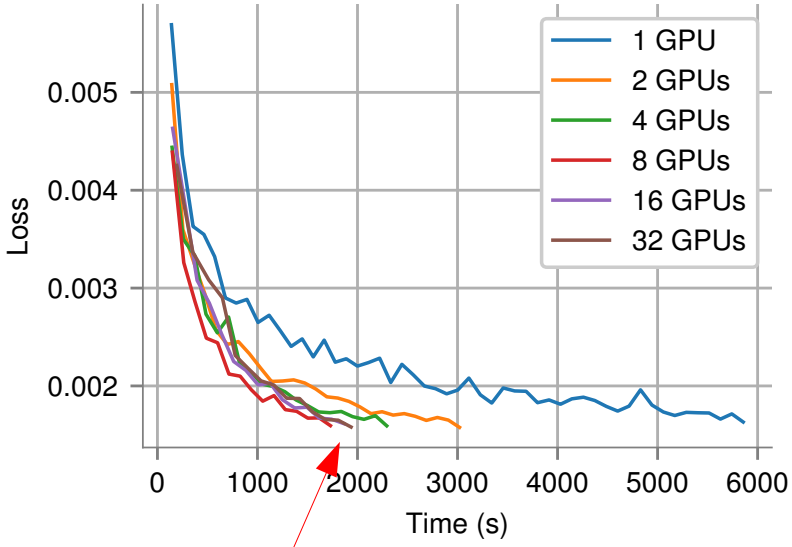
$$w_{t+1} = w_t - k\eta \frac{1}{|\mathcal{B}|} \sum_{j<k} \sum_{x \in B_j} \nabla l(x, w_t).$$

Data-parallel training of ANN

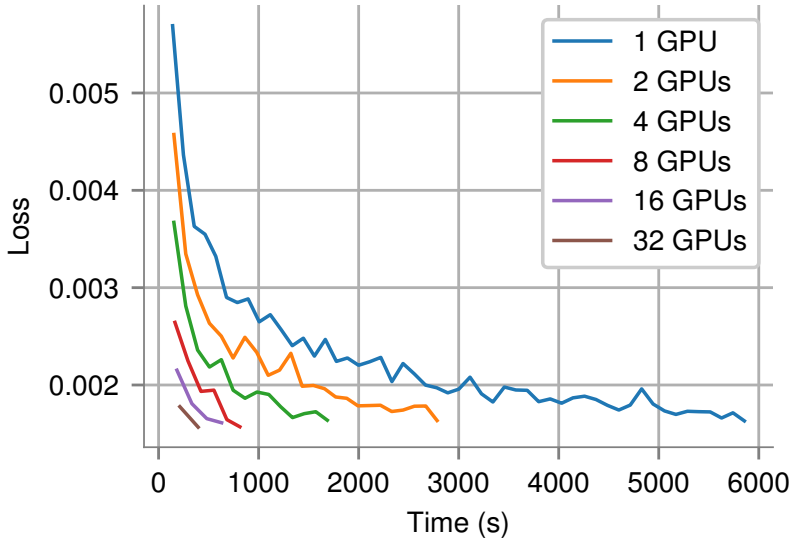
Potential issues with large mini batches

R. de F. Cunha et al.: An argument in favor of strong scaling for deep neural networks with small datasets (arXiv:1807.09161)

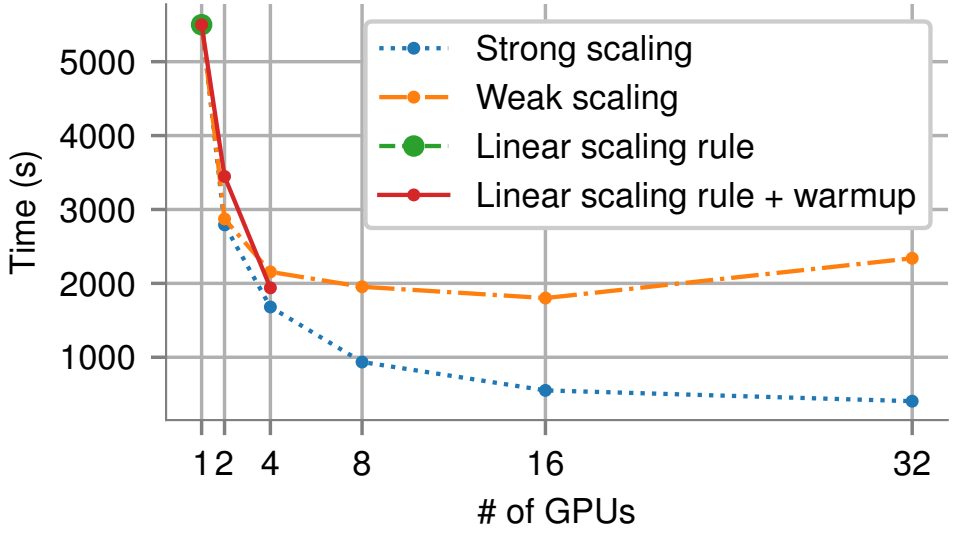
"weak scaling" of per-proc. mini-batch size



"strong scaling" of per-proc. mini-batch size



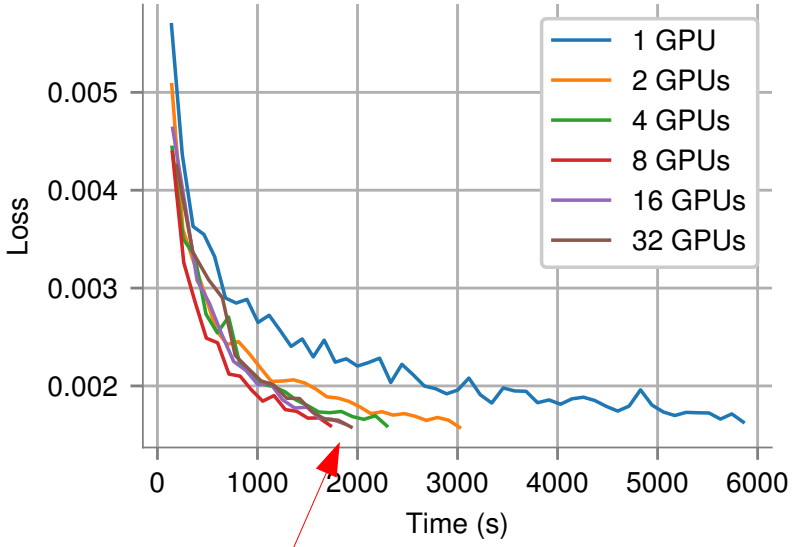
poor scalability



no convergence for a given accuracy ("loss")

Potential issues with large mini batches

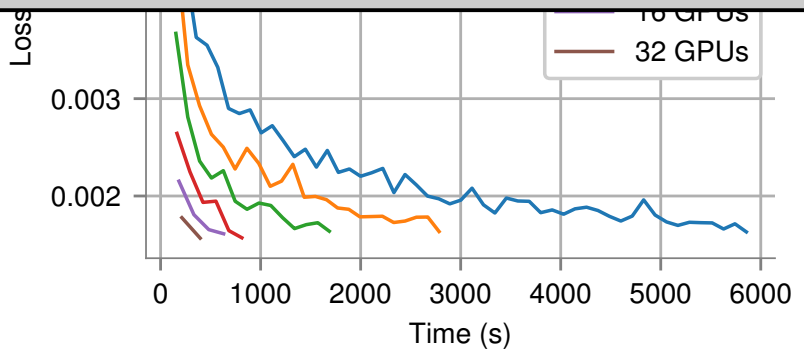
"weak scaling" of per-proc. mini-batch size

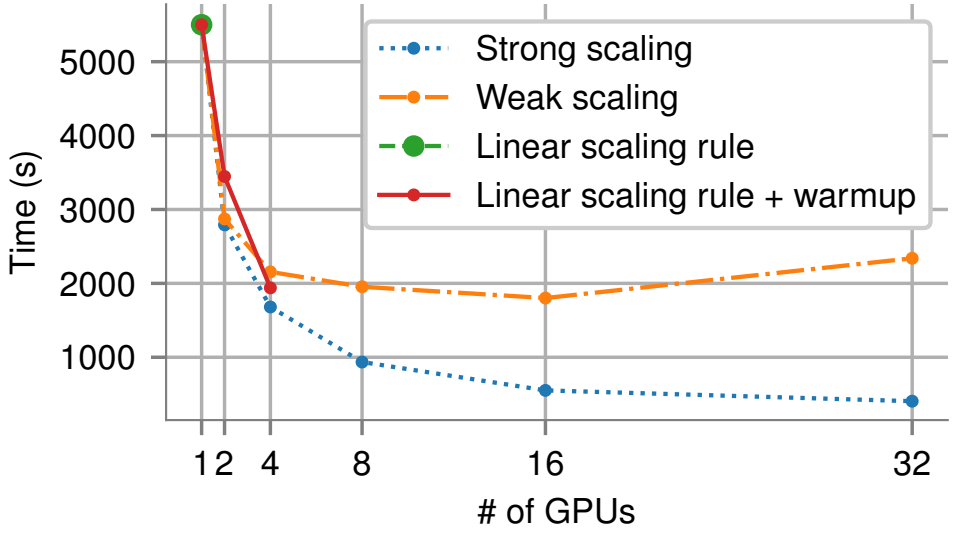


poor scalability

R. de F. Cunha et al.: An argument in favor of strong scaling for deep neural networks with small datasets (arXiv:1807.09161)

"We believe some results reported in the literature may not transfer to problems that lack large amounts of data, and may be biased towards the ImageNet benchmark."





no convergence for a given accuracy ("loss")

Benchmarking ANN: what is the right metric?

→time to solution ! = time to reach a specified accuracy (validation loss)

→ commonly used: images/second (= throughput)

→opens up many opportunities to cheat (ourselves)

→watch out !

https://htor.inf.ethz.ch/blog/index.php/2018/11/08/twelve-ways-to-fool-the-masses-when-reporting-performance-of-deep-learning-workloads/

Twelve ways to fool the masses … (by T. Hoefler)

1) Ignore accuracy when scaling up!

Our first guideline to report highest performance is seemingly one of the most common one. Scaling deep learning is very tricky because the best performing optimizer, stochastic gradient descent (SGD), is mostly sequential. Model parallelism can be achieved by processing the elements of a minibatch in parallel — however, the best size of the minibatch is determined by the statistical properties of the process and is thus limited. However, when one ignores the quality (or convergence in general), the model-parallel SGD will scale wonderfully to any size system out there! Weak scaling by adding more data can benefit this further, after all we can process all that data in parallel. In practice, unfortunately, test accuracy matters, not how much data one processed. One way around this may be to only report time for a small number of iterations because, at large scale, it's too expensive to run to convergence, right?

2) Do not report test accuracy!

The SGD optimization method optimizes the function that the network represents to the dataset used for learning. This minimizes the so called training error. However, it is not clear whether the training error is a useful metric. After all, the network could just learn all examples without any capability to work on unseen examples. This is a classic case of overfitting. Thus, real-world users typically report test accuracy of an unseen dataset because machine learning is not optimization! Yet, when scaling deep learning computations, one must tune many so called hyperparameters (batch size, learning rate, momentum, …) to enable convergence of the model. It may not be clear whether the best setting of those parameters benefits the test accuracy as well. In fact, there is evidence that careful tuning of hyperparameters may decrease the test accuracy by overfitting to a specific problem.

3) Do not report all training runs needed to tune hyperparameters!

…

Twelve ways to fool the masses … (by T. Hoefler)

9) Train on unreasonably large inputs!

This is my true favorite, the pinnacle of floptimization! It took me a while to recognize and it's quite powerful. The image classification community is almost used to scaling down high-resolution images to ease training. After all, scaling to 244×244 pixels retains most of the features and gains a quadratic factor (in the image width/hight) of computation time. However, such small images are rather annoying when scaling up because they require too little compute. Especially for small minibatch sizes, scaling is limited because processing a single small picture on each node is very inefficient. Thus, if flop/s are important then one shall process large, e.g., "high-resolution", images. Each node can easily process a single example now and the 1,000x increase on needed compute comes nicely to support scaling and overall flop/s counts! A win-win unless you really care about the science done per cost or time. In general, when procesing very large inputs, there should be a good argument why — one teraflop compute per example may be excessive.

…

11) Minibatch sizing for fun and profit – weak vs. strong scaling….

We all know about weak vs. strong scaling, i.e., the simpler case when the input size scales with the number of processes and the harder case when the input size is constant. At the end, deep learning is all strong scaling because the model size is fixed and the total number of examples is fixed. However, one can cleverly utilize the minibatch sizes. Here, weak scaling keeps the minibatch size per process constant, which essentially grows the global minibatch size. Yet, the total epoch size remains constant, which causes less iterations per epoch and thus less overall communication rounds. Strong scaling keeps the global minbatch size constant. Both have VERY different effects in convergence — weak scaling worsens convergence eventually because it reduces stochasiticity and strong scaling does not.

…

Basic communication pattern: **sum over all processors**

processor-local sum

$$w_{t+1} = w_t - \eta \frac{1}{|\mathcal{B}'|} \sum_{i=0}^{N-1} \left(\sum_{x \in B_i} \nabla l(x, w_t) \right)$$

Parameter server architecture (Distributed Tensorflow)

→ introduces communication bottleneck

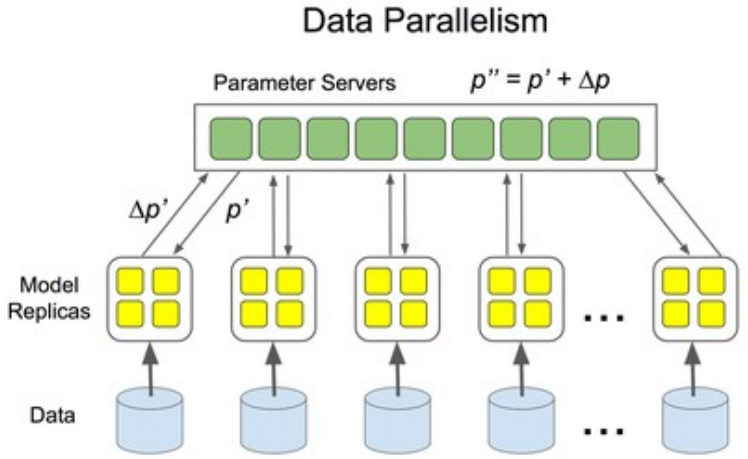


Image from henning.kropponline.de

Communication patterns

Basic communication pattern: `MPI_Allreduce`

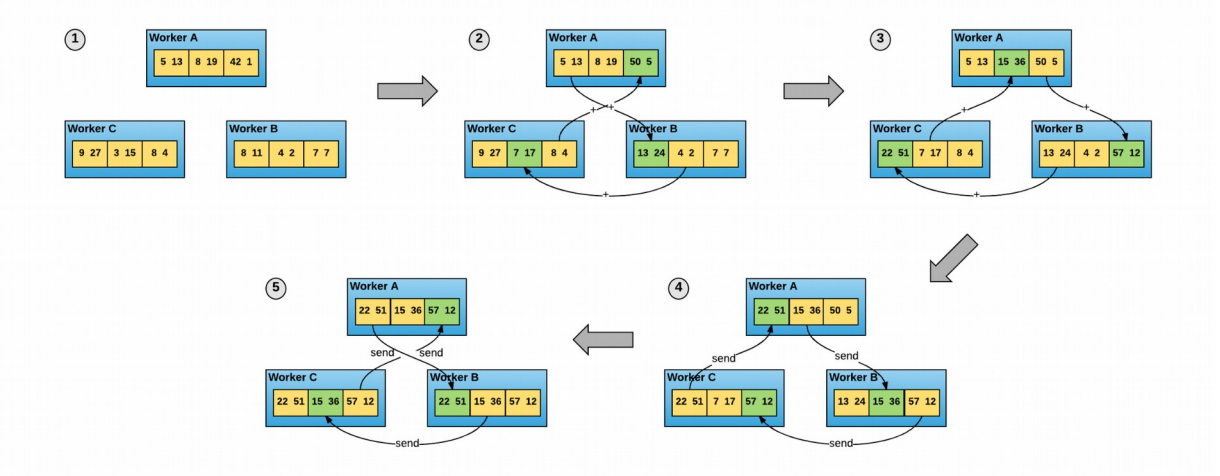
processor-local sum

$$w_{t+1} = w_t - \eta \frac{1}{|\mathcal{B}'|} \sum_{i=0}^{N-1} \left(\sum_{x \in B_i} \nabla l(x, w_t) \right)$$

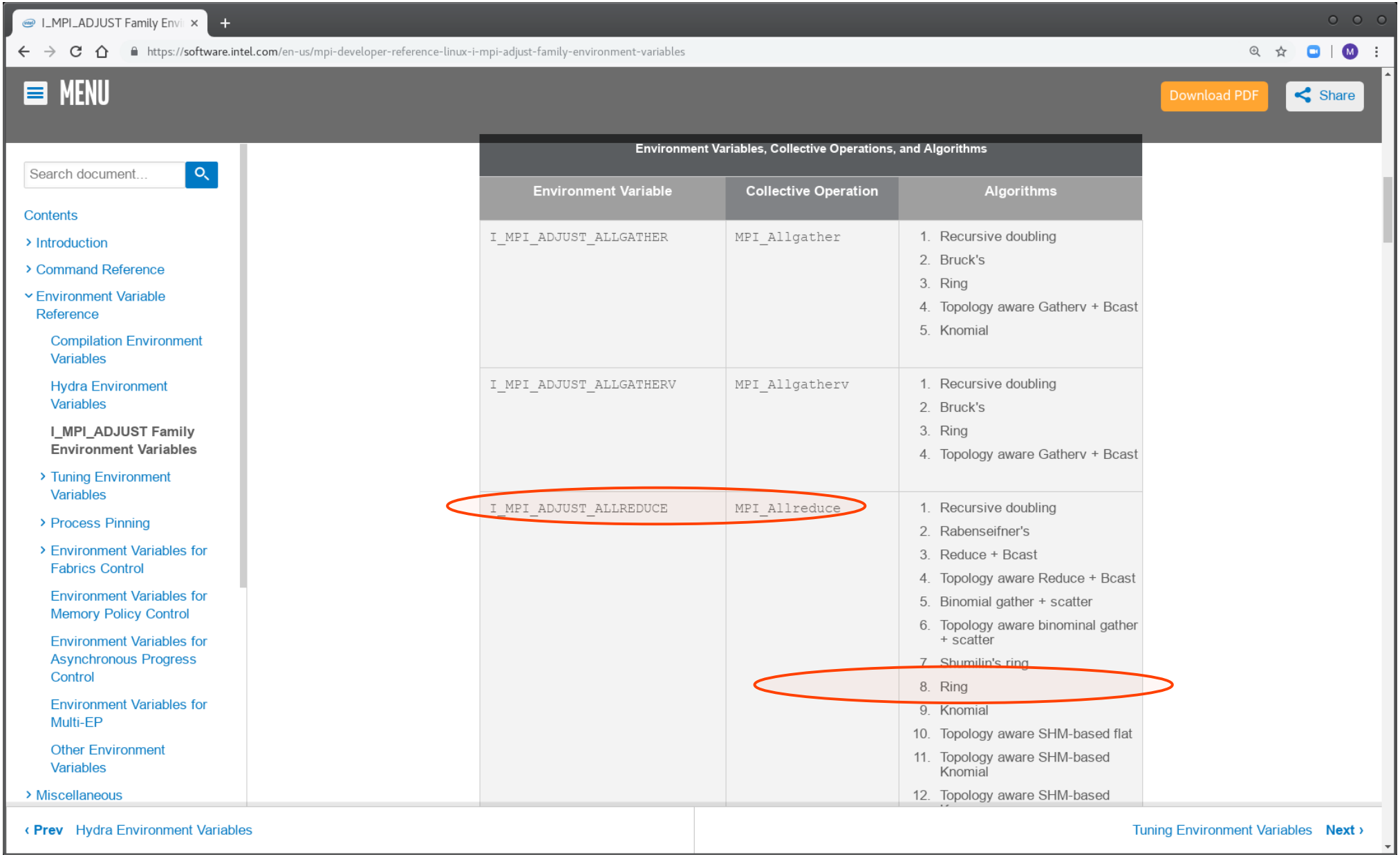
De-centralized architecture based on the well-known Message Passing Interface (MPI), and its high-performance library and runtime implementations (OpenMPI, IntelMPI, ...)

Baidu-allreduce (2017): TensorFlow fork (https://github.com/baidu-research/baidu-allreduce)

Horovod (2018): “ring-allreduce”, integrates with TensorFlow (arXiv: 1802.05799)



Welcome to HPC ...



I_MPI_ADJUST Family Env

https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-family-environment-variables

MENU

Download PDF | Share

Search document...

Contents

- Introduction
- Command Reference
- Environment Variable Reference
  - Compilation Environment Variables
  - Hydra Environment Variables
  - **I_MPI_ADJUST Family Environment Variables**
  - Tuning Environment Variables
  - Process Pinning
  - Environment Variables for Fabrics Control
  - Environment Variables for Memory Policy Control
  - Environment Variables for Asynchronous Progress Control
  - Environment Variables for Multi-EP
  - Other Environment Variables
- Miscellaneous

Environment Variables, Collective Operations, and Algorithms

Environment Variable	Collective Operation	Algorithms
I_MPI_ADJUST_ALLGATHER	MPI_Allgather	1. Recursive doubling 2. Bruck's 3. Ring 4. Topology aware Gatherv + Bcast 5. Knomial
I_MPI_ADJUST_ALLGATHERV	MPI_Allgatherv	1. Recursive doubling 2. Bruck's 3. Ring 4. Topology aware Gatherv + Bcast
I_MPI_ADJUST_ALLREDUCE	MPI_Allreduce	1. Recursive doubling 2. Rabenseifner's 3. Reduce + Bcast 4. Topology aware Reduce + Bcast 5. Binomial gather + scatter 6. Topology aware binominal gather + scatter 7. Shumilin's ring 8. Ring 9. Knomial 10. Topology aware SHM-based flat 11. Topology aware SHM-based Knomial 12. Topology aware SHM-based

‹ Prev Hydra Environment Variables | Tuning Environment Variables Next ›

Data parallel training with Horovod

Horovod (https://github.com/horovod/horovod) developed at Uber

Builds on the MPI communication API

Supported frameworks:

- TensorFlow
- Keras
- PyTorch
- MXNet

Execute with srun/mpirun/mpiexec/orterun python ….
(convenience wrapper for OpenMPI: horovodrun ...)

https://www.mpcdf.mpg.de/services/computing/software/data-analytics/machine-learning-software

Data parallel training with TF/Horovod

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
import math
import tensorflow as tf

# Horovod:
import horovod.keras as hvd

# Horovod: initialize Horovod.
hvd.init()

# Horovod: pin GPU to be used to process local rank (one GPU per process)
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(hvd.local_rank())
K.set_session(tf.Session(config=config))

batch_size = 128
num_classes = 10

# Horovod: adjust number of epochs based on number of GPUs.
epochs = int(math.ceil(12.0 / hvd.size()))

# Input image dimensions
img_rows, img_cols = 28, 28
```

https://github.com/horovod/horovod

Data parallel training with TF/Horovod

```
# The data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
 x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
 x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
 input_shape = (1, img_rows, img_cols)
else:
 x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
 x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
 input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
 activation='relu',
 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

https://github.com/horovod/horovod

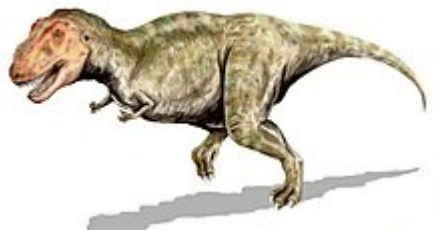
Data parallel training with TF/Horovod

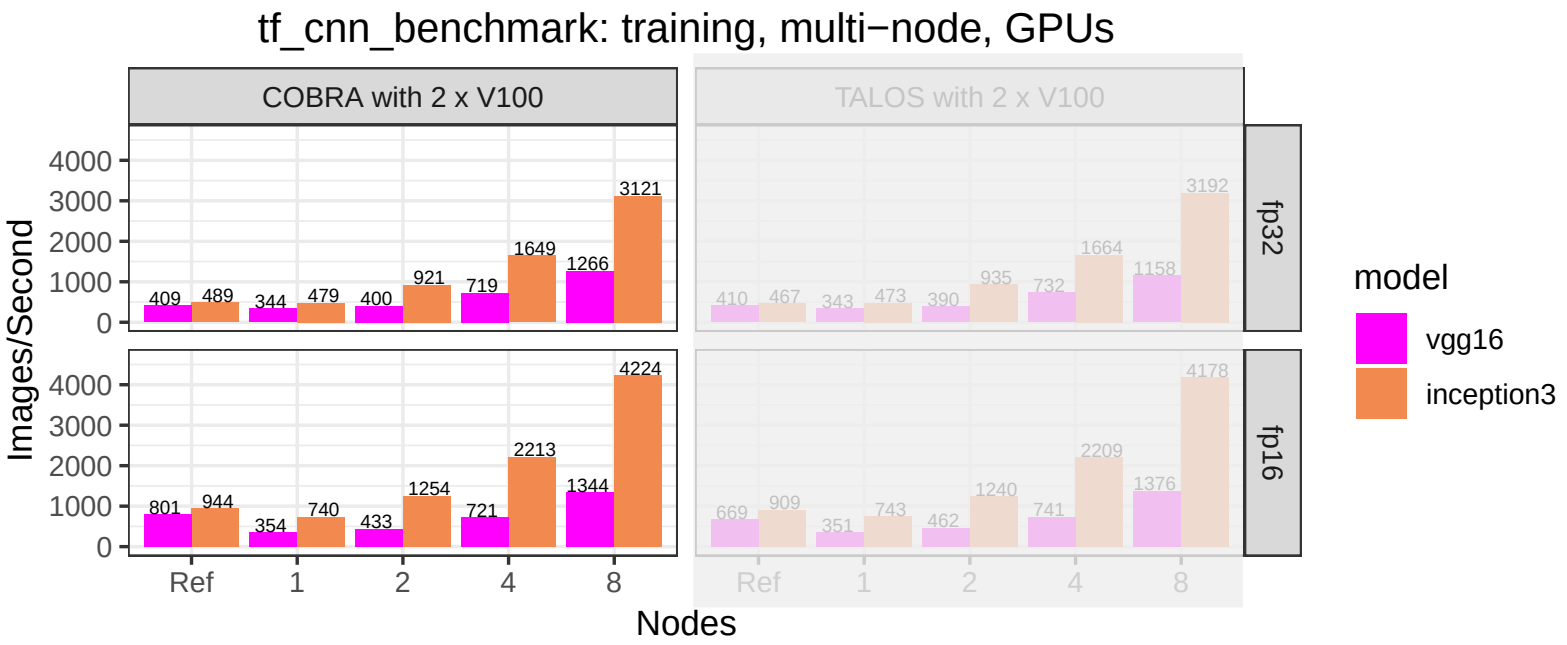
```
# Horovod: adjust learning rate based on number of GPUs.
opt = keras.optimizers.Adadelta(1.0 * hvd.size())

# Horovod: add Horovod Distributed Optimizer.
opt = hvd.DistributedOptimizer(opt)

model.compile(loss=keras.losses.categorical_crossentropy,
 optimizer=opt,
 metrics=['accuracy'])

callbacks = [
 # Horovod: broadcast initial variable states from rank 0 to all other processes.
 # This is necessary to ensure consistent initialization of all workers when
 # training is started with random weights or restored from a checkpoint.
 hvd.callbacks.BroadcastGlobalVariablesCallback(0),
]

# Horovod: save checkpoints only on worker 0 to prevent other workers from corrupting them.
if hvd.rank() == 0:
 callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-{epoch}.h5'))

model.fit(x_train, y_train,
 batch_size=batch_size,
 callbacks=callbacks,
 epochs=epochs,
 verbose=1,
 validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

https://github.com/horovod/horovod

Benchmarking TensorFlow/Horovod

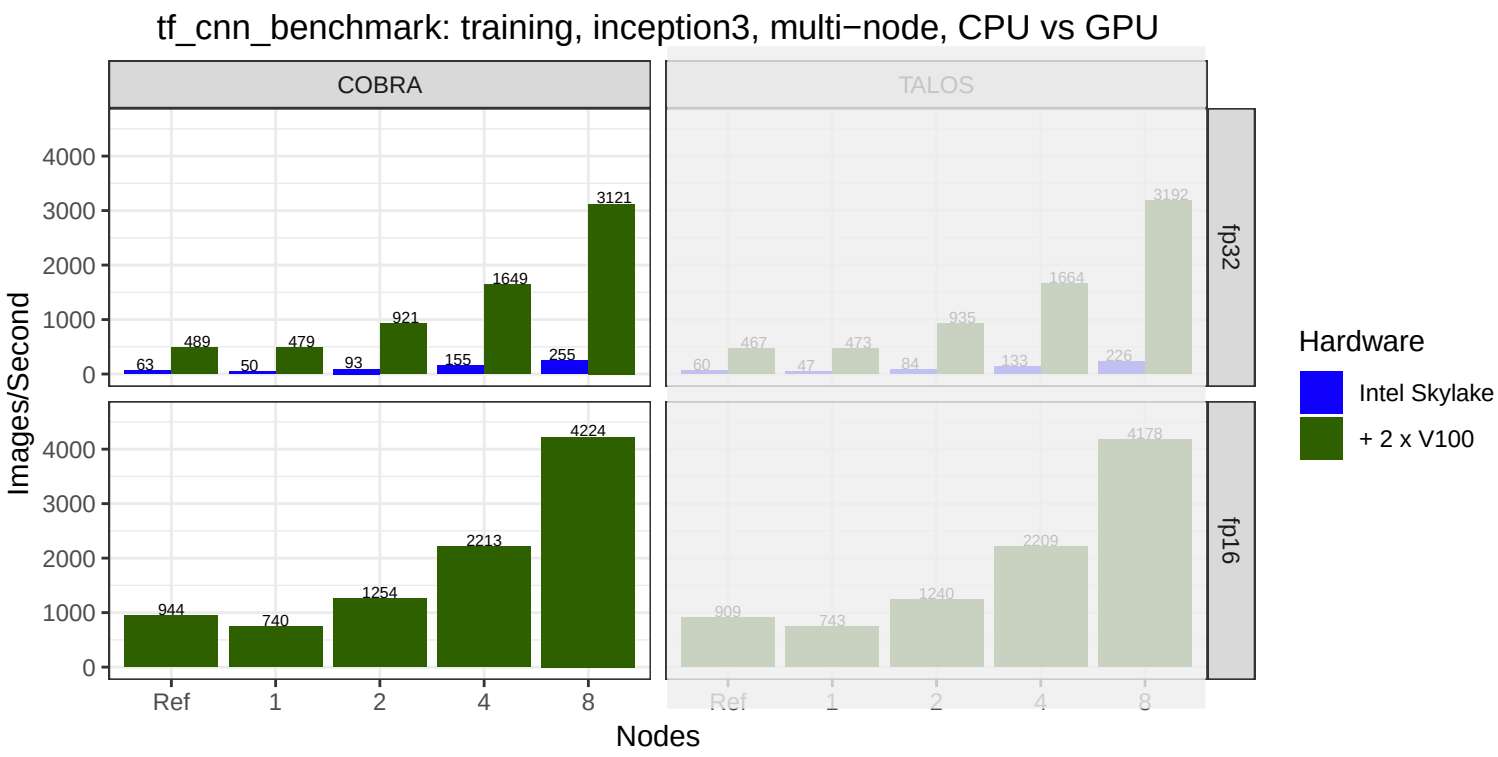
https://github.com/tensorflow/benchmarks





→ scaling across nodes works efficiently

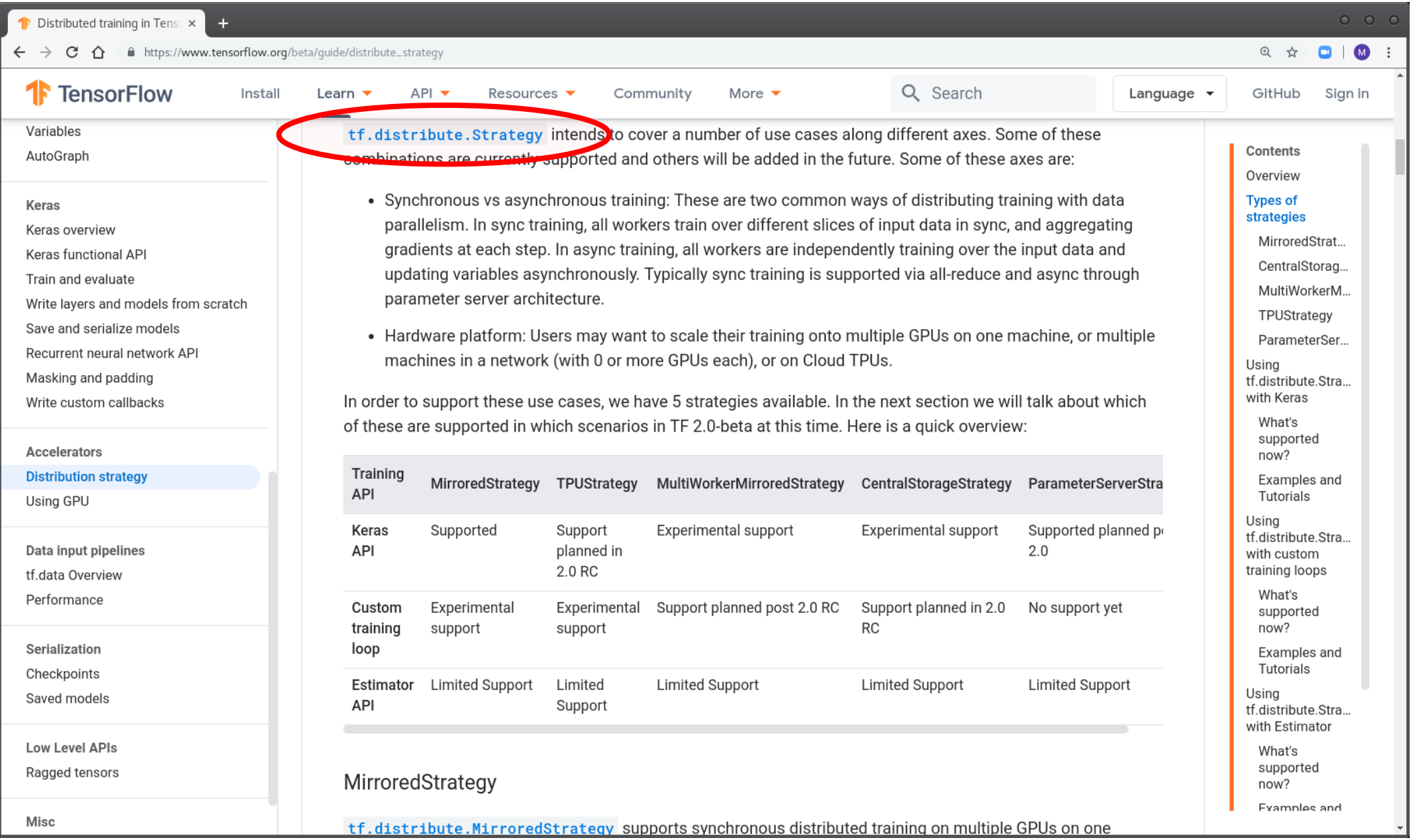
Benchmarking TensorFlow/Horovod



→ scaling across nodes works efficiently

→ GPUs provide significant speedup (wrt. CPU-only)

TensorFlow 2.0 beta



towards native MPI support? Horovod? new API ? obsoletes … ?

Model-parallelism in ANN inference:

- an illustrative example from MRI

Automatic segmentation of 3D medical images
MP Institute for Human Cognitive and Brain Sciences (Dept. N. Weiskopf)

- Goal: use a (deep) CNN to segment 3D data from histology samples of brain tissue

Our present knowledge of the cortical structure is based on the analysis of physical 2D sections .[…] Now with the combination of novel 3D imaging techniques and advanced image analysis methods, such as deep neural networks, the study of the fully three-dimensional structure of the brain is within Reach (K. Thierbach et al. 2019, publication in progress)

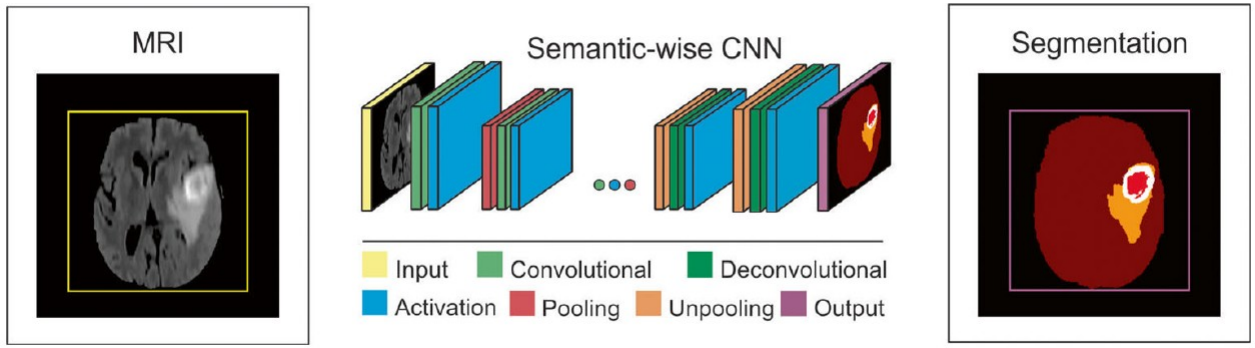


Figure from Z. Akkus et al. 2017: Deep Learning for Brain MRI Segmentation: State of the Art and Future Directions

Automatic segmentation of 3D medical images
MP Institute for Human Cognitive and Brain Sciences (Dept. N. Weiskopf)

- Challenges: compute power and memory requirements in the *inference* step, due to project requirements:

  - a fully convolutional mixed-scale dense convolutional neural network (MS-DNet) is used (100k parameters to train)

  - training can be done on (small) data sets of 96^3 voxels on one GPU node

  - inference is done on 2K x 1K x 1K voxels (estimate: needs 16 PFlop operations and 24 TB of memory in TensorFlow)

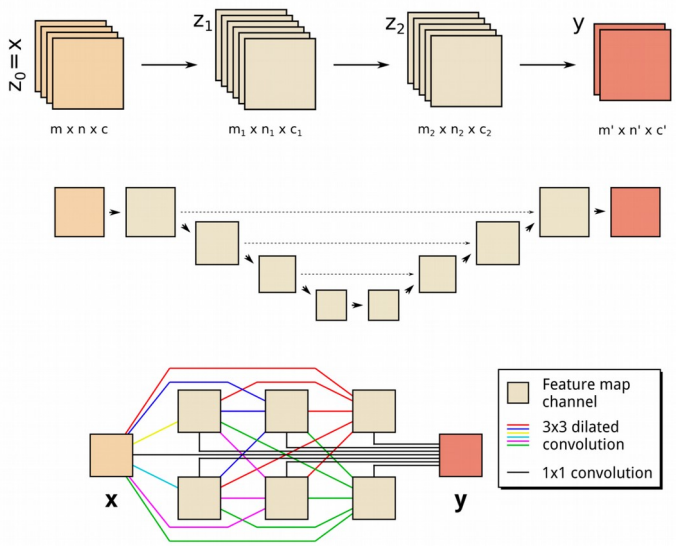


Figure from D.M.Pelt & J.A.Sethian, 2017, A mixed-scale dense convolutional neural network for image analysis

=> inference step must be parallelized over multiple nodes
=> standard setups with TensorFlow, PyTorch, … do not work, since they do not provide model-parallelism during inferencing

Automatic segmentation of 3D medical images
MP Institute for Human Cognitive and Brain Sciences (Dept. N. Weiskopf)

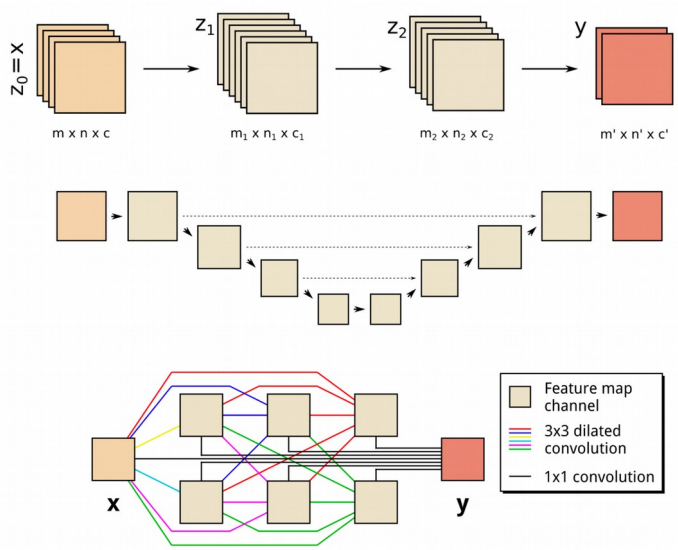
- Solution implemented at MPCDF:

  - HPC approach of a “domain-decomposition”

  - split the 3D data set in cubes of 120^3 voxels (maximum fitting into memory of V100 GPU); consider a configurable overlap between splitting

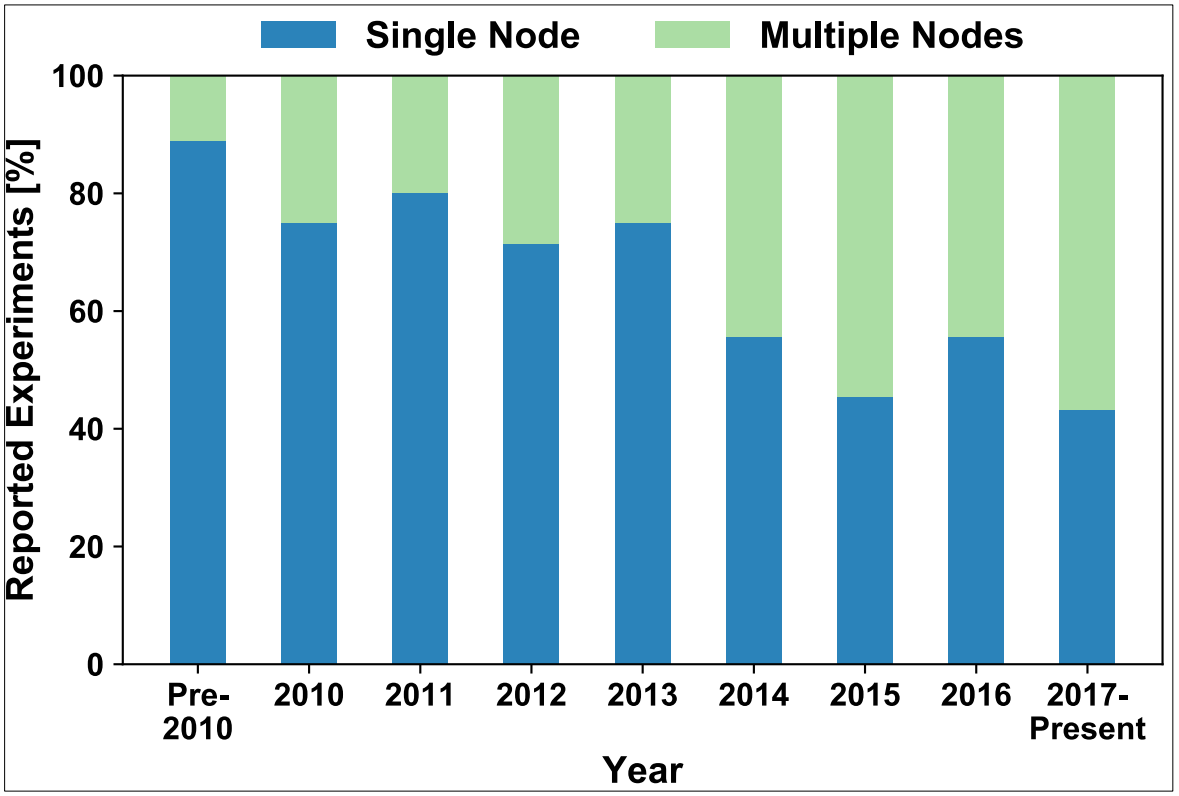
  - process each cube independently with TensorFlow, take care of (partially) detected objects in the overlap region

  - stitch all results to a final result of size 2K x 1K x 1K



=> “bookkeeping” of different inference jobs via SLURM job arrays
=> one batch of ca. 600 cubes can be processed in ~400 s on one GPU
=> we managed to run full problem in ca. 500 s on 16 compute nodes (32 GPUs)

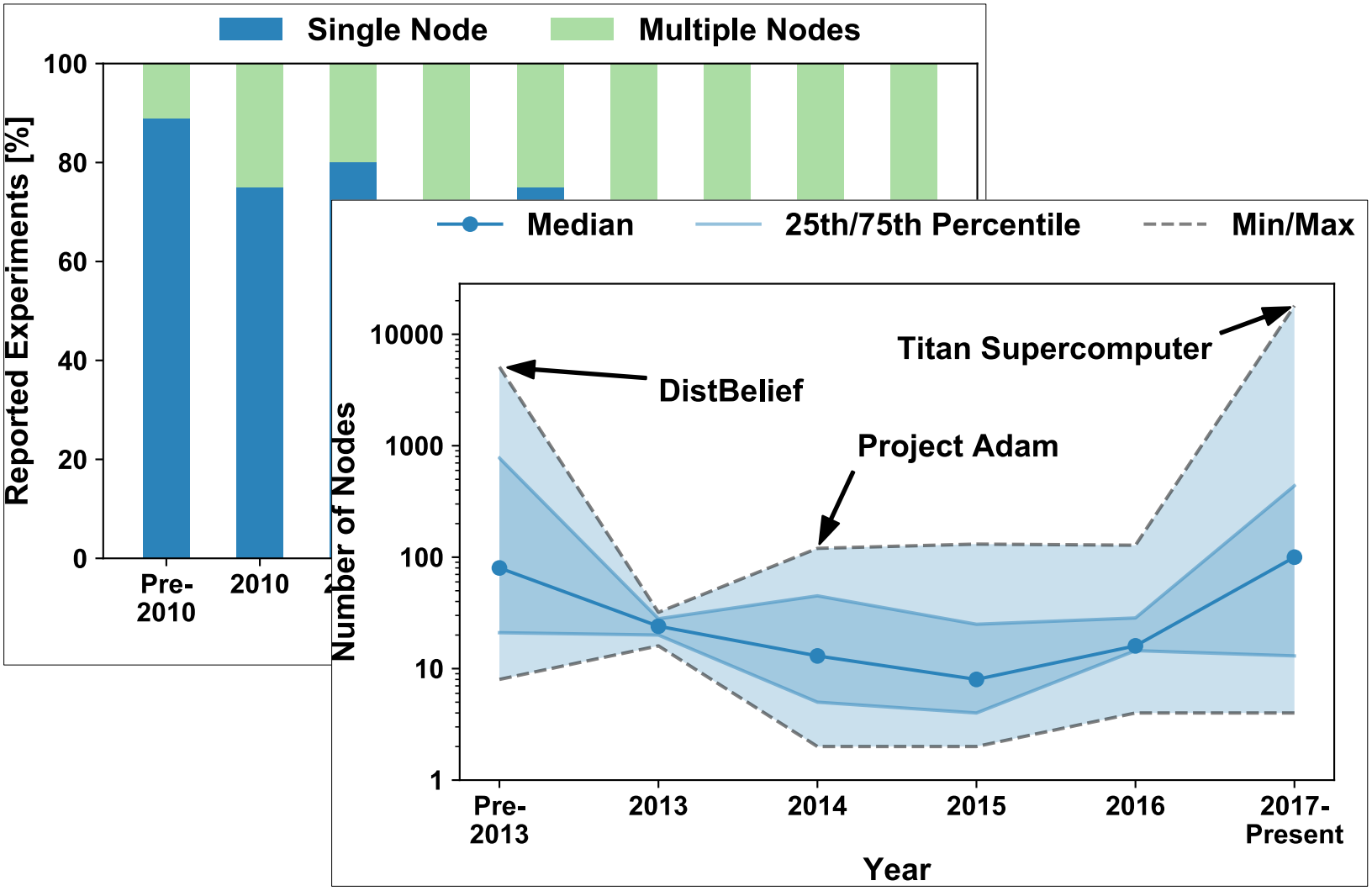
Relevance of distributed ANN computation



arXiv:1802.09941

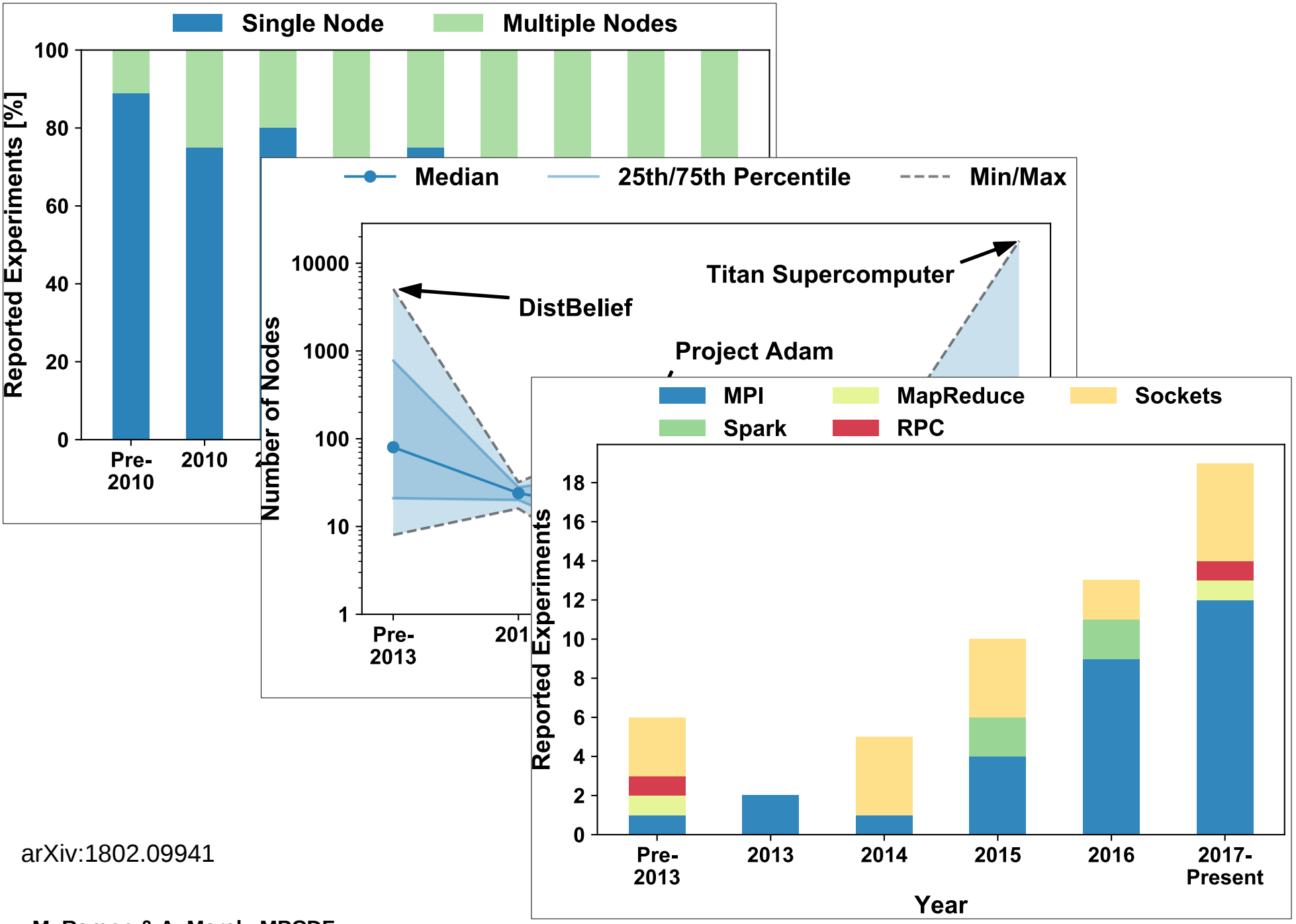
M. Rampp & A. Marek, MPCDF

Relevance of distributed ANN computation



arXiv:1802.09941

Relevance of distributed ANN computation



arXiv:1802.09941

References

- T. Ben-Nun & T. Hoefler: *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis* (arXiv:1802.09941)
- T. Lin et al.: *Don't Use Large Mini-Batches, Use Local SGD* (arXiv:1808.07217)
- R. de Cunha et al.: *An argument in favor of strong scaling for deep neural networks with small datasets* (arXiv:1807.09161)
- R. Mayer & H.-A. Jacobsen: *Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques and Tools* (arXiv:1903.11314)
- P. Sun et al.: *Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes* (arXiv:1902.06855)
- K. Chahal et al.: *A Hitchhiker's Guide On Distributed Training of Deep Neural Networks* (arXiv:1810.11787)
- A. Sergeev & M. Del Balso: *Horovod: fast and easy distributed deep learning in TensorFlow* (arXiv:1802.05799)