Learning Systems 2018:
Lecture **16** – Parallelism in Deep Learning

Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

Topics for Today

- Why parallelism?
- Taxonomy of approaches
- Understanding performance
- Examples
  - LBANN
  - MENNDL
- Parallelism in practice

Why parallelism?

- Parallelism is essential due to the enormous computational demands of deep learning – and sometimes because of large size of models

- Fortunately:
  - “The world of deep learning is brimming with concurrency. Nearly every aspect of training, from the computation of a convolution to the meta-optimization of DNN architectures, is inherently parallel. Even if an aspect is sequential, its consistency requirements can be reduced, due to the robustness of nonlinear optimization, to increase concurrency while still attaining reasonable accuracy, if not better.” [https://arxiv.org/abs/1802.09941]

- Challenge: Parallelize not paralyze

Implementing Neural Network Models on Parallel Computers

B. M. FORREST, D. ROWETH*, N. STROUD, D. J. WALLACE AND G. V. WILSON

Department of Physics, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ

The remarkable processing capabilities of the nervous system must derive from the large numbers of neurons participating (roughly 10^{10}), since the time-scales involved are of the order of a millisecond, rather than the nanoseconds of modern computers. The neural network models which attempt to capture this behaviour are inherently parallel.

We review the implementation of a range of neural network models on SIMD and MIMD computers. On the ICL Distributed Array Processor (DAP), a 4096-processor SIMD machine, we have studied training algorithms in the context of the Hopfield net, with specific applications including the storage of words and continuous text in content-addressable memory. The Hopfield and Tank analogue neural net has been used for image restoration with the Geman and Geman algorithm. We compare the performance of this scheme on the DAP and on a Meiko Computing Surface, a reconfigurable MIMD array of transputers. We describe also the strategies which we have used to implement the Durbin and Willshaw elastic net model on the Computing Surface.

Received June 1987

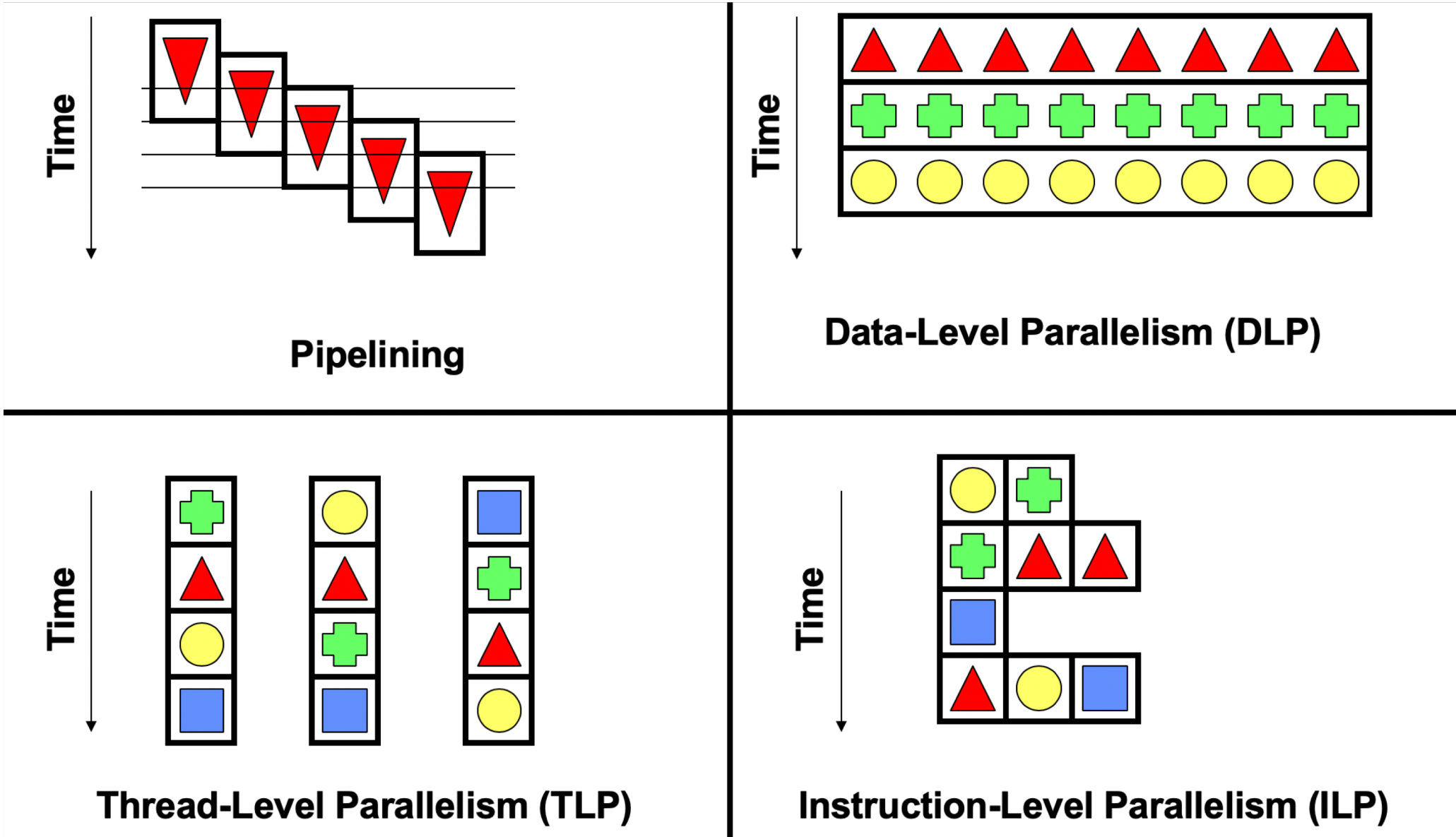
“Exploring the Limits of Language Modeling”

Table 1. Best results of single models on the 1B word benchmark. Our results are shown below previous work.

MODEL	TEST PERPLEXITY	NUMBER OF PARAMS [BILLIONS]
SIGMOID-RNN-2048 (JI ET AL., 2015A)	68.3	4.1
INTERPOLATED KN 5-GRAM, 1.1B N-GRAMS (CHELBA ET AL., 2013)	67.6	1.76
SPARSE NON-NEGATIVE MATRIX LM (SHAZEER ET AL., 2015)	52.9	33
RNN-1024 + MAXENT 9-GRAM FEATURES (CHELBA ET AL., 2013)	51.3	20
LSTM-512-512	54.1	0.82
LSTM-1024-512	48.2	0.82
LSTM-2048-512	43.7	0.83
LSTM-8192-2048 (NO DROPOUT)	37.9	3.3
LSTM-8192-2048 (50% DROPOUT)	32.2	3.3
2-LAYER LSTM-8192-1024 (BIG LSTM)	30.6	1.8
BIG LSTM+CNN INPUTS	**30.0**	**1.04**
BIG LSTM+CNN INPUTS + CNN SOFTMAX	39.8	**0.29**
BIG LSTM+CNN INPUTS + CNN SOFTMAX + 128-DIM CORRECTION	35.8	**0.39**
BIG LSTM+CNN INPUTS + CHAR LSTM PREDICTIONS	47.9	**0.23**

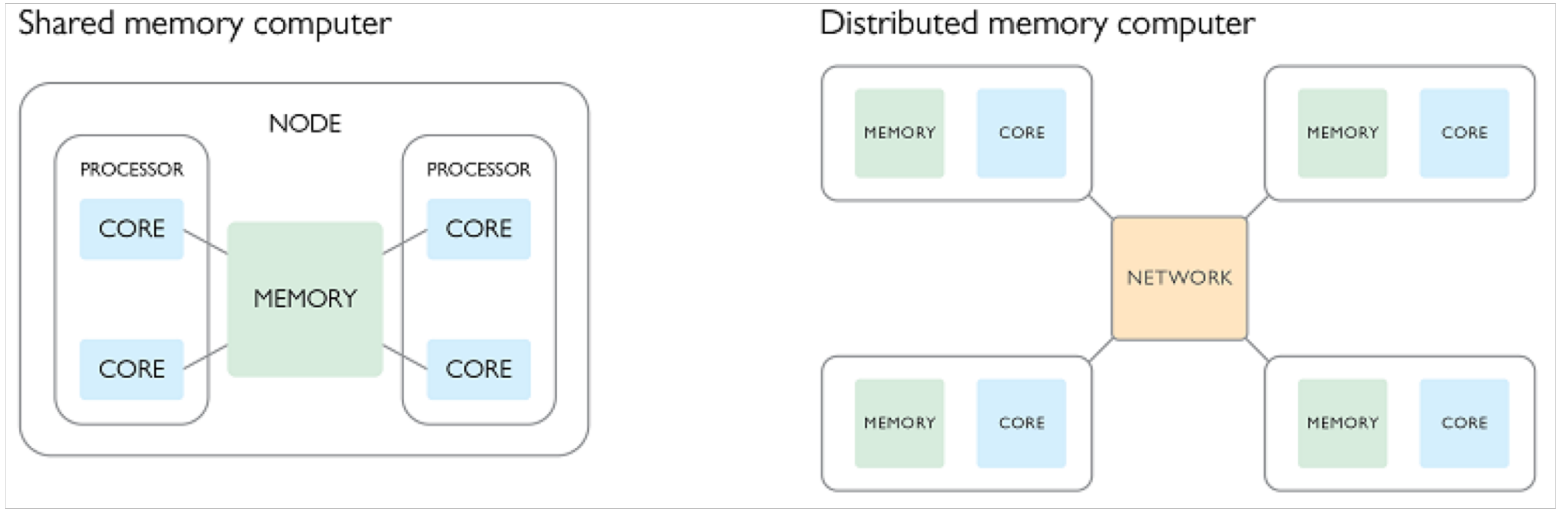
https://arxiv.org/abs/1602.02410

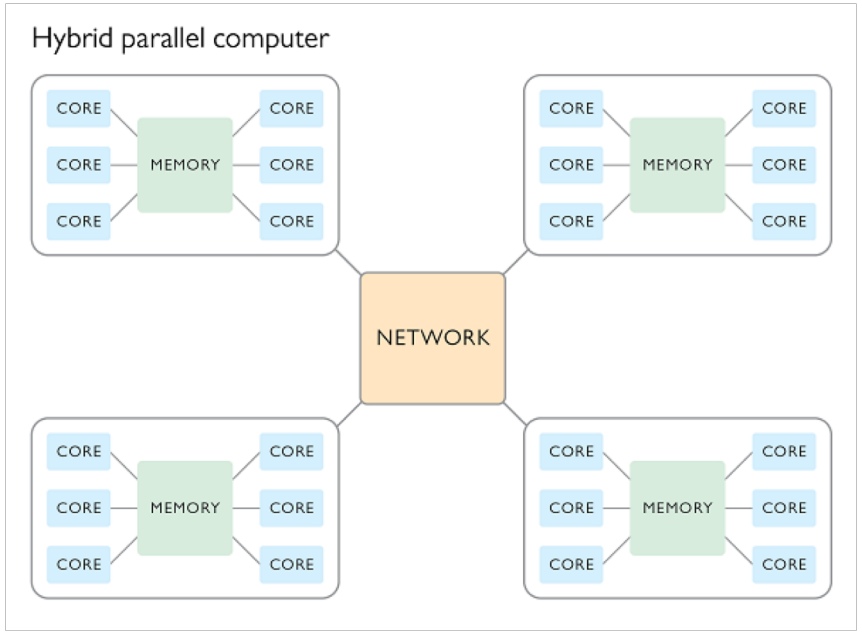
There are many types of parallelism within a modern CPU and GPU (i.e., on a **single machine**)



http://bit.ly/2qN2llQ

We focus on parallelism **across** nodes (i.e., on **multiple machines**)





https://www.comsol.jp/blogs/hybrid-parallel-computing-speeds-up-physics-simulations/

Why parallelism?

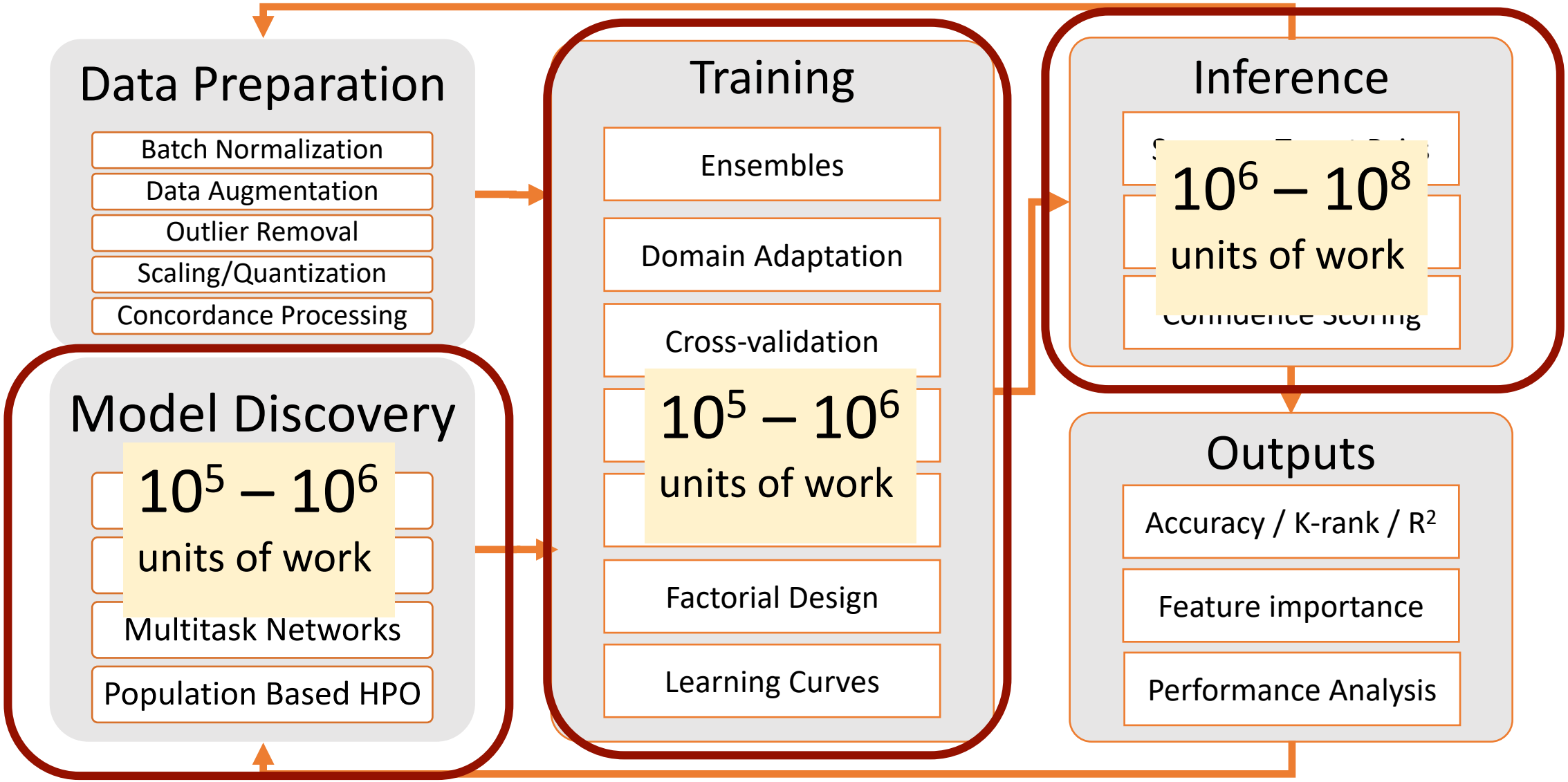
- Find a good model
  - Hyperparameter optimization
- Train a single model
  - Train faster
  - Train with more data
  - Train a more complex model
- Inference
  - Perform a single inference faster
  - Perform many inferences faster
- Other
  - Uncertainty quantification

CANDLE Challenge Problem Workflow Structure

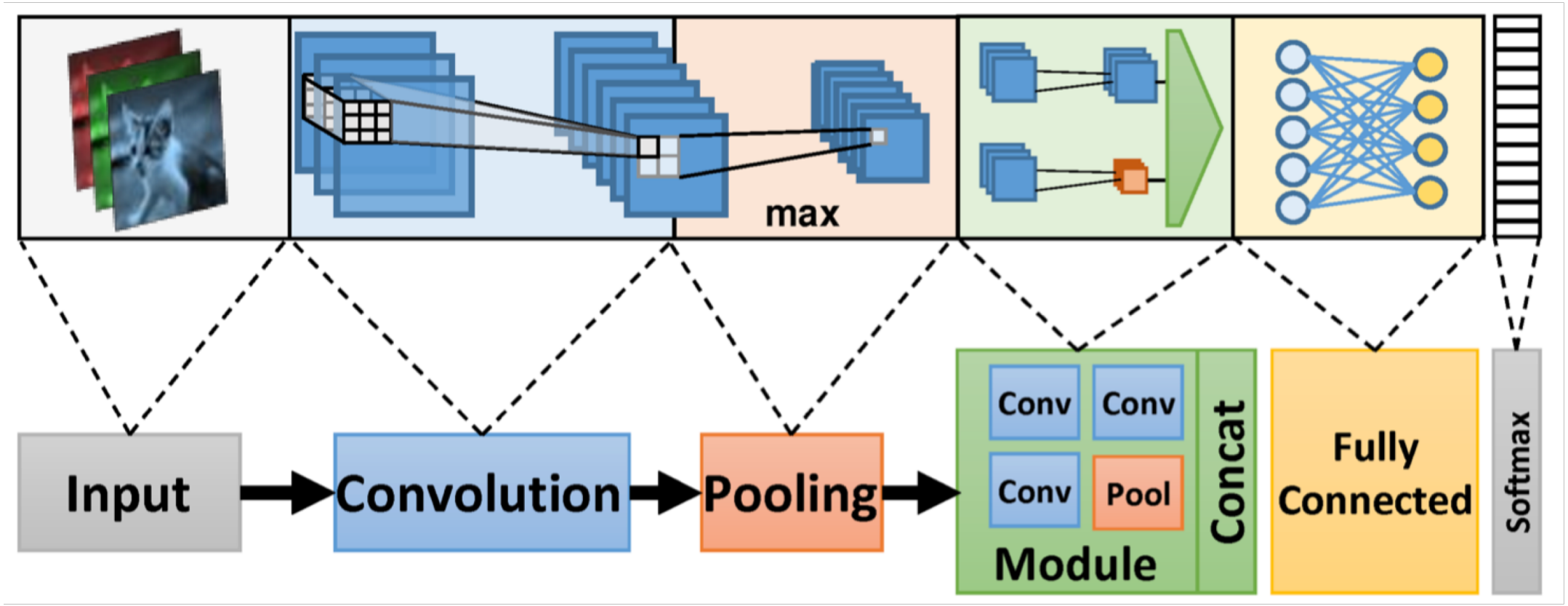
Each CP has three compute intensive phases and two data processing phases

1. Prepare input datasets, including normalization, outlier removal, joins and merges, etc.
2. Broad search of model space (AutoML+HPO) to find good performing model structures and hyper-parameter settings ⟹ running **$O(10^4) – O(10^6)$** model instances (HPO model selection ⟹ small number of model templates < 10)
3. Training a large-scale ensemble of the best model types on relevant training data for a "factorial study" ⟹ **$O(10^6)$** models on **$O(10^4)$** datasets (5x cross validation and optional bootstrap UQ) (CV model selection ⟹ small number of models < 10 per Source-Target pair)
4. Inferencing with selected models with UQ on new samples (**$O(10^7)$** in (and out) of each Source-Target Pair distributions (UQ implies sampling model space **$O(10^2)$** times)
5. Post-process inference output for actionable decisions and to capture performance, confidence and scoring (for validation)

P1 Challenge Problem Workflow(s) Specification



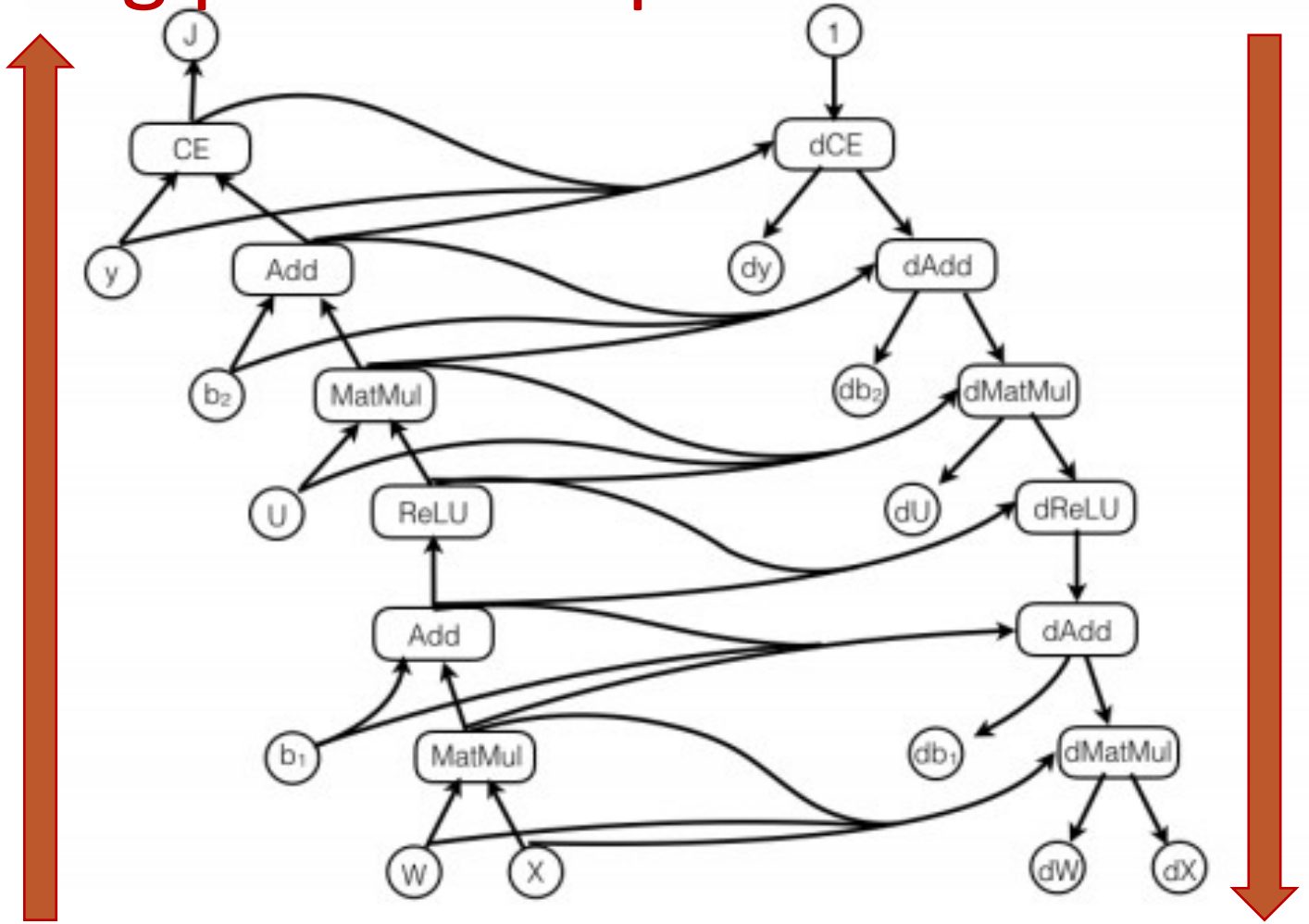
Let’s focus on training



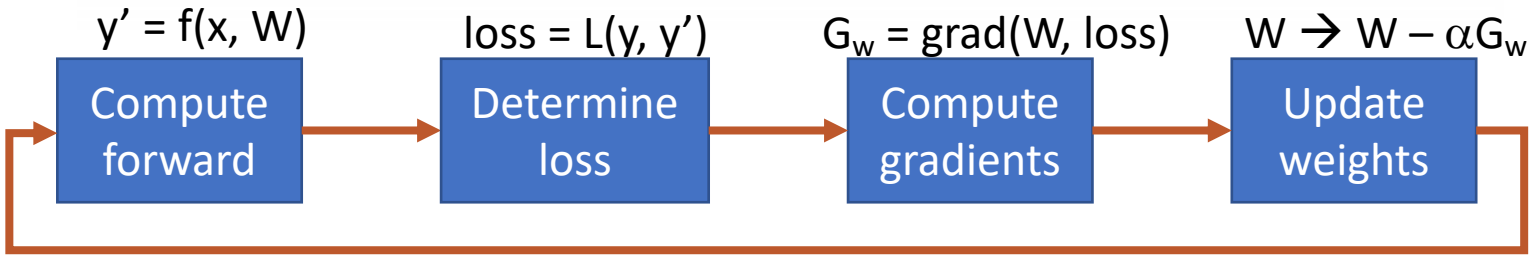
- The critical figure of merit is **time to specified accuracy**
- Others may include resource consumption, scalability with respect to amount of data and network size, …

Recall training process: repeated forward then backward

Forward propagation



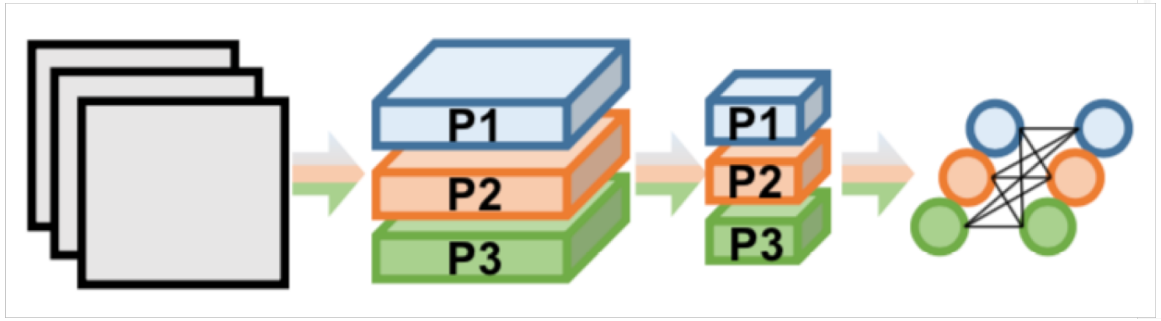
Backward propagation

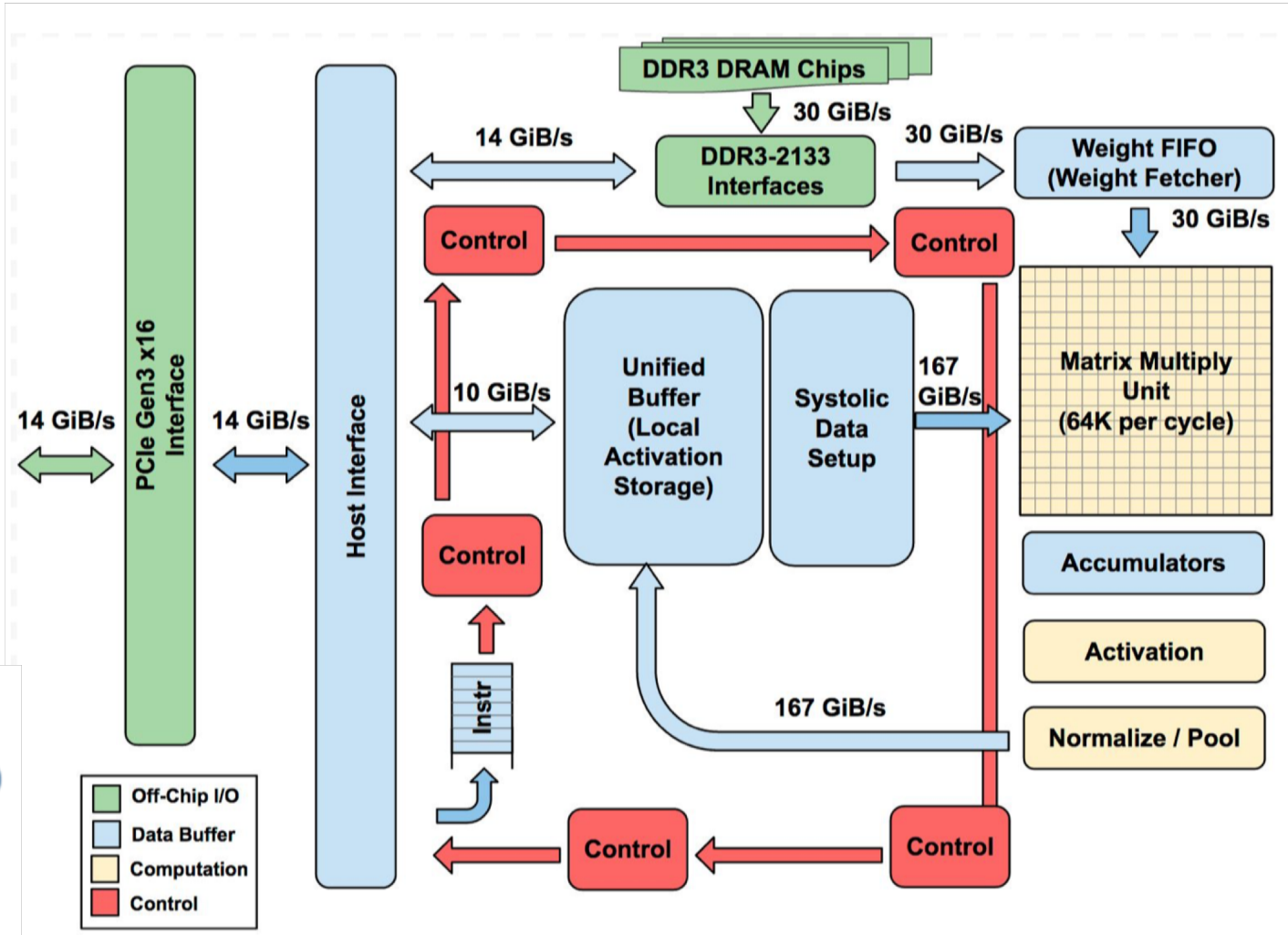


https://www.linkedin.com/pulse/how-tensorflow-calculate-gradients-shamane-siriwardhana/

Parallelization strategy 1

- Parallelize the linear algebra performed in the forward and backward steps
- This is what GPUs and TPUs do **within** a node
- Parallelization **across** nodes can also be done, if matrices are large …

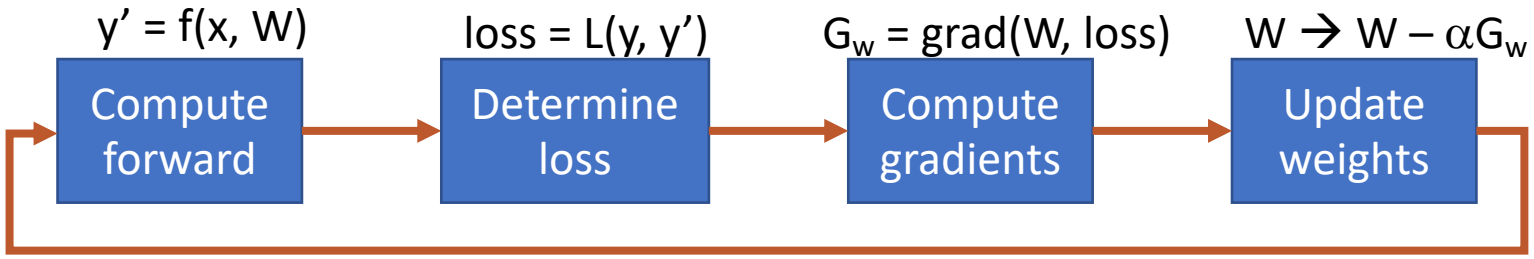


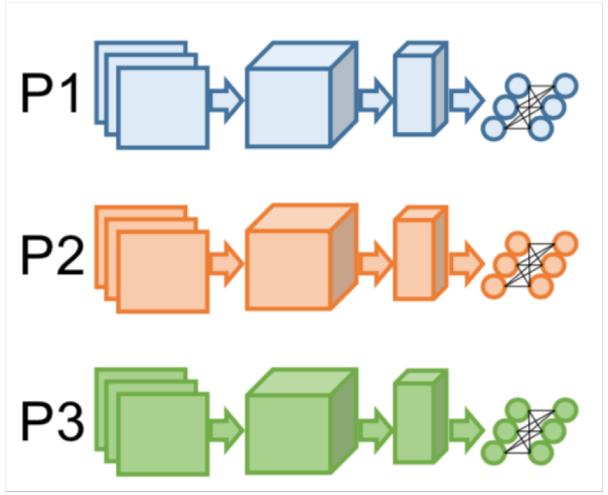


- Advantage: reduce memory

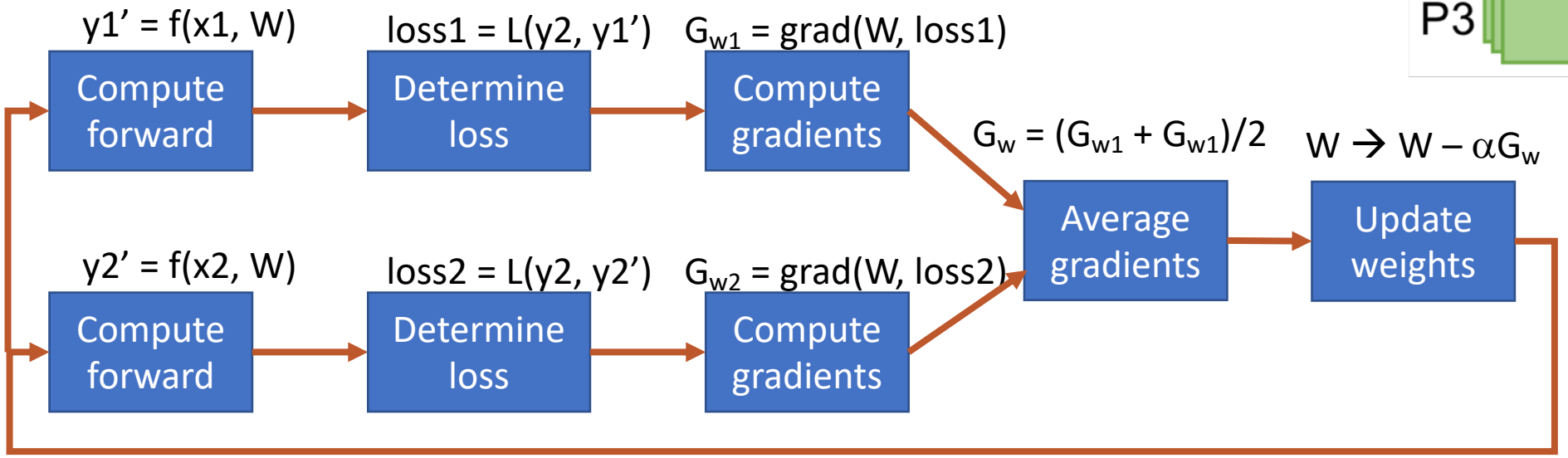
Parallelization strategy #2

- Relax the sequential ordering of the gradient descent process:



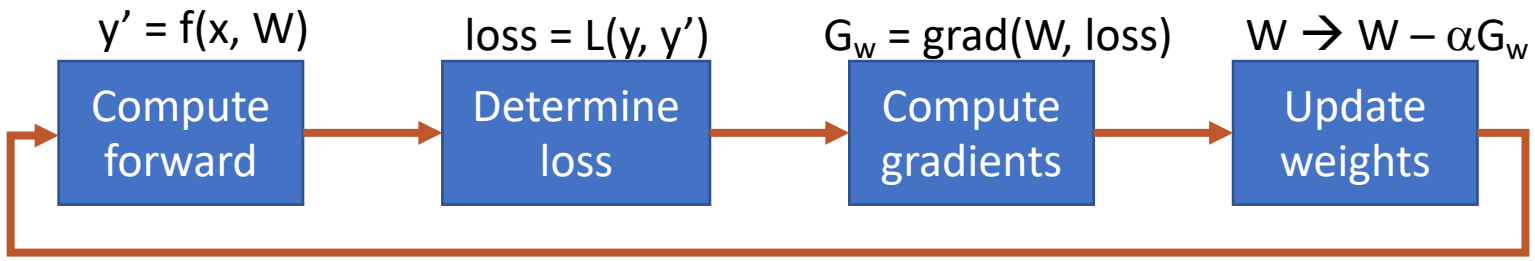


- For example:

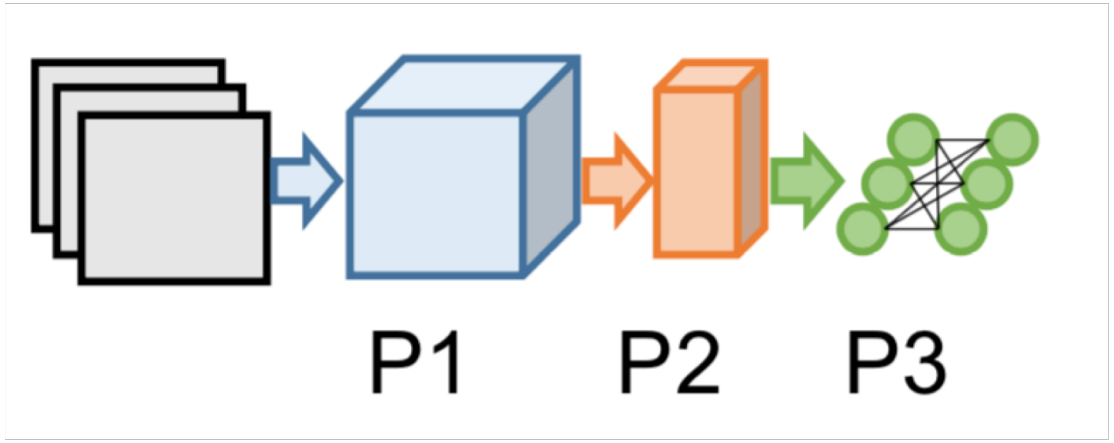


Parallelization strategy #3

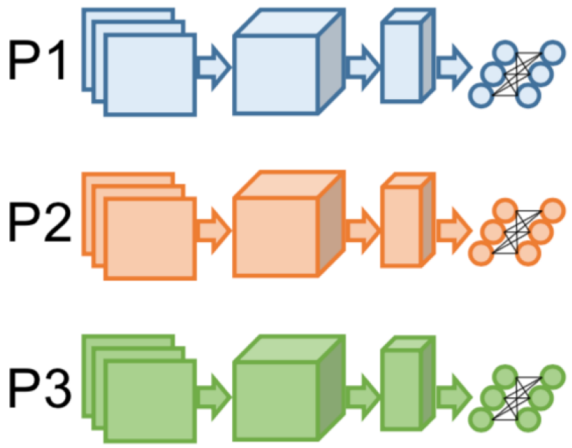
- Relax the sequential ordering of the gradient descent process:



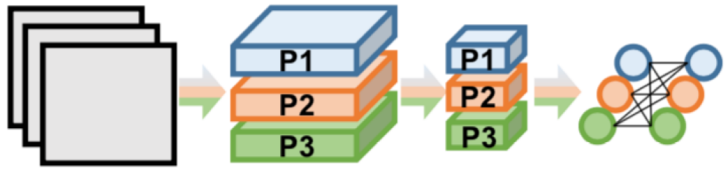
- Parallelize by levels, and pipeline minibatches through:



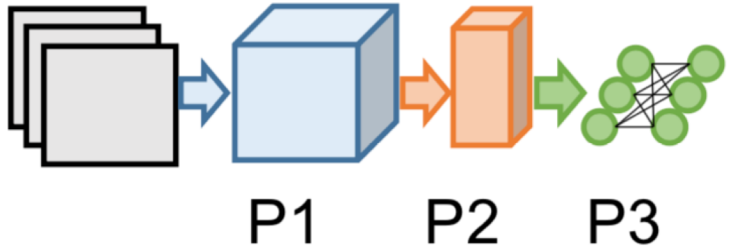
Three parallelism schemes



(a) Data Parallelism



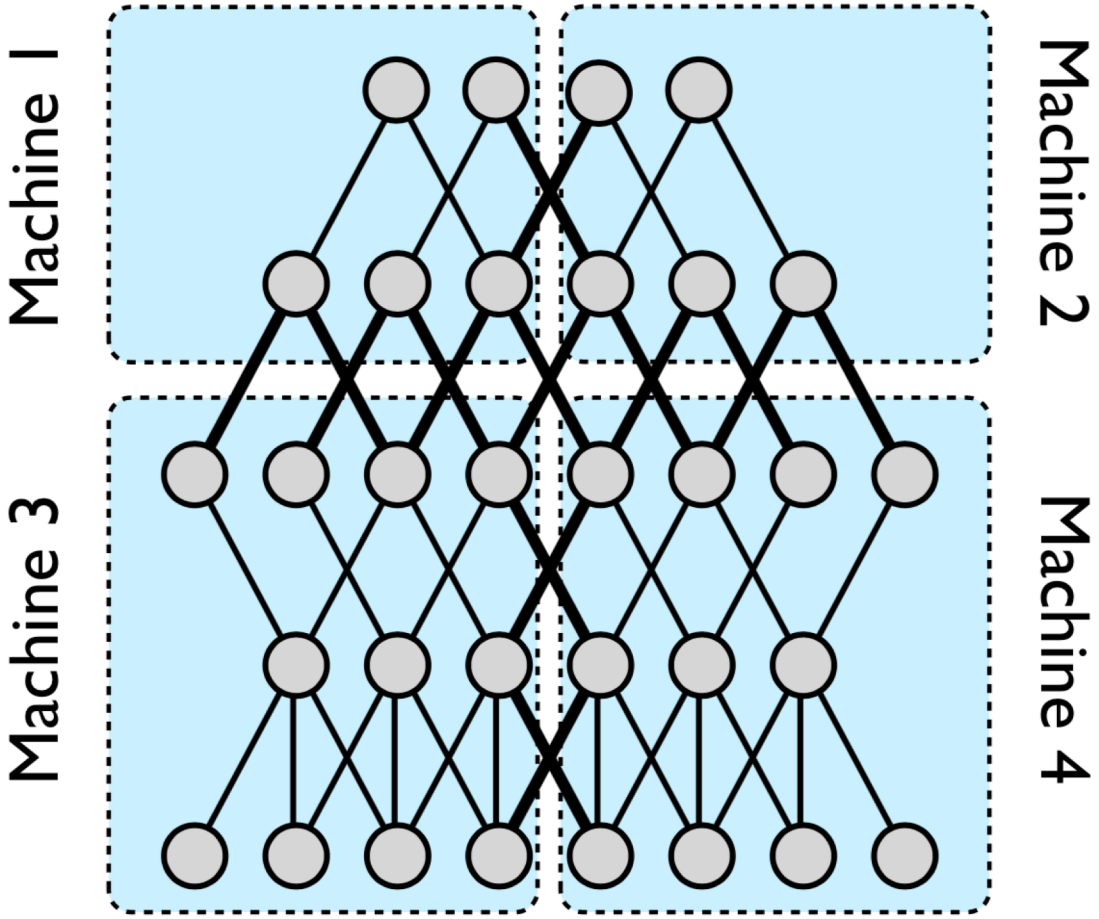
(b) Model Parallelism



(c) Layer Pipelining

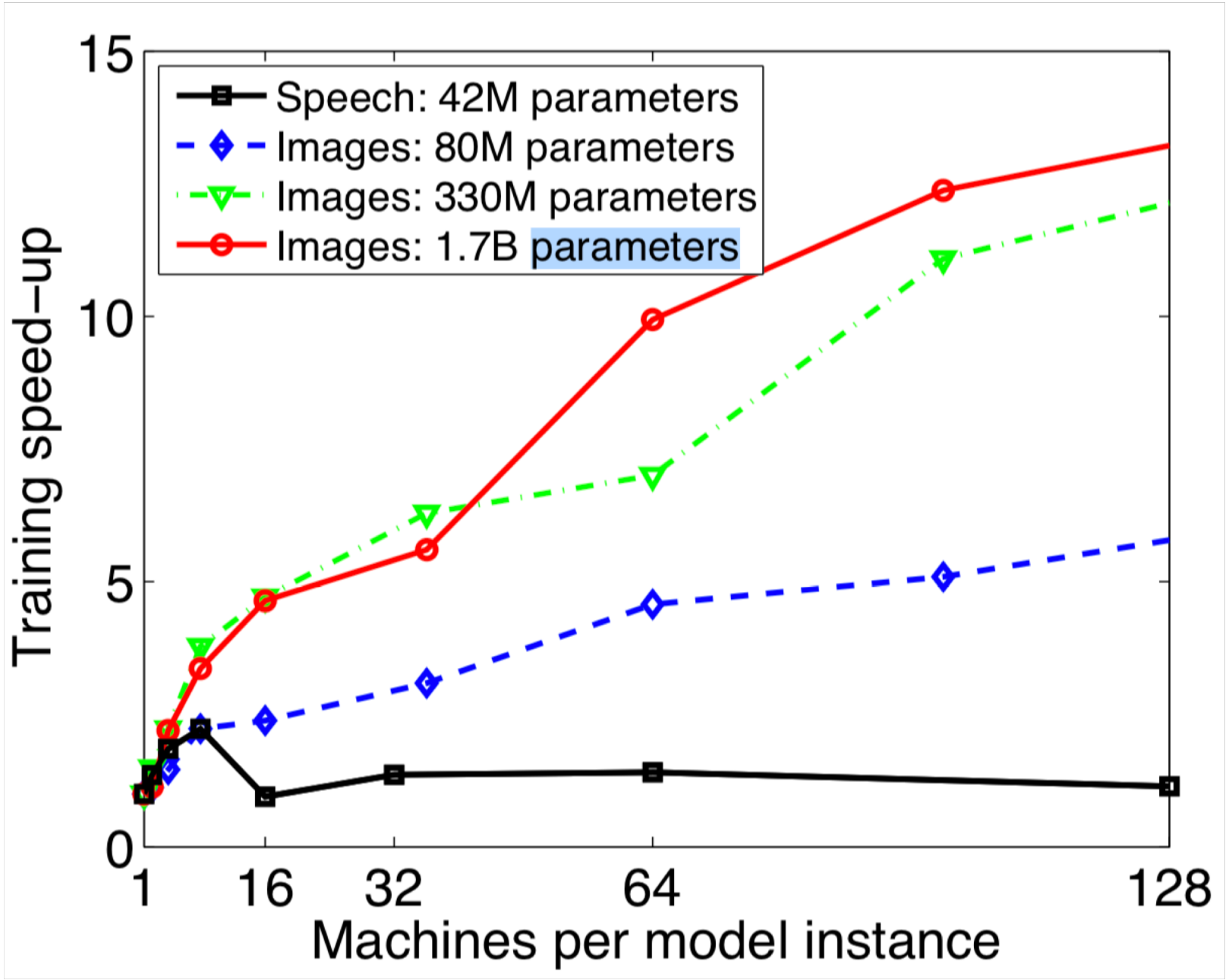
https://arxiv.org/abs/1802.09941

A hybrid scheme



https://ai.google/research/pubs/pub40565

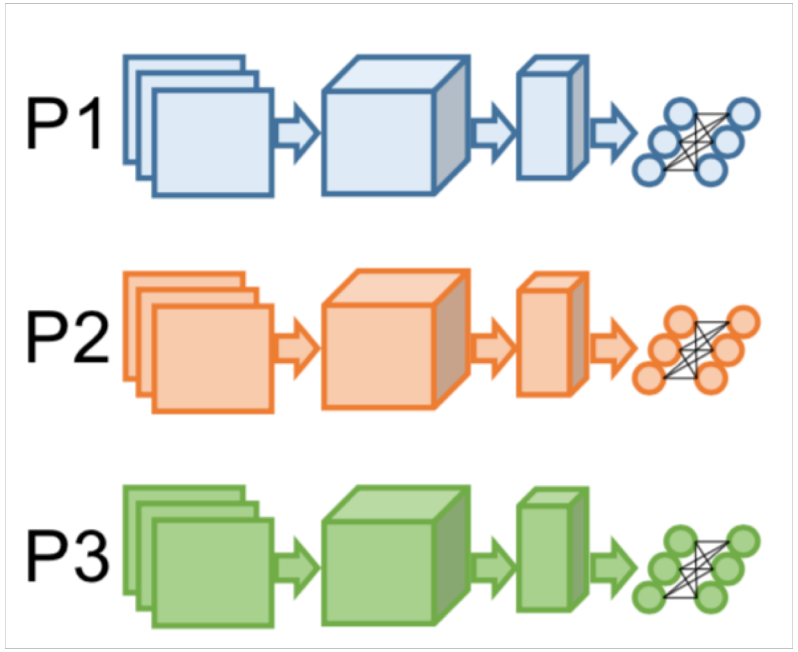
Figure 1: An example of model parallelism in DistBelief. A five layer deep neural network with local connectivity is shown here, partitioned across four machines (blue rectangles). Only those nodes with edges that cross partition boundaries (thick lines) will need to have their state transmitted between machines. Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once. Within each partition, computation for individual nodes will the parallelized across all available CPU cores.



https://ai.google/research/pubs/pub40565

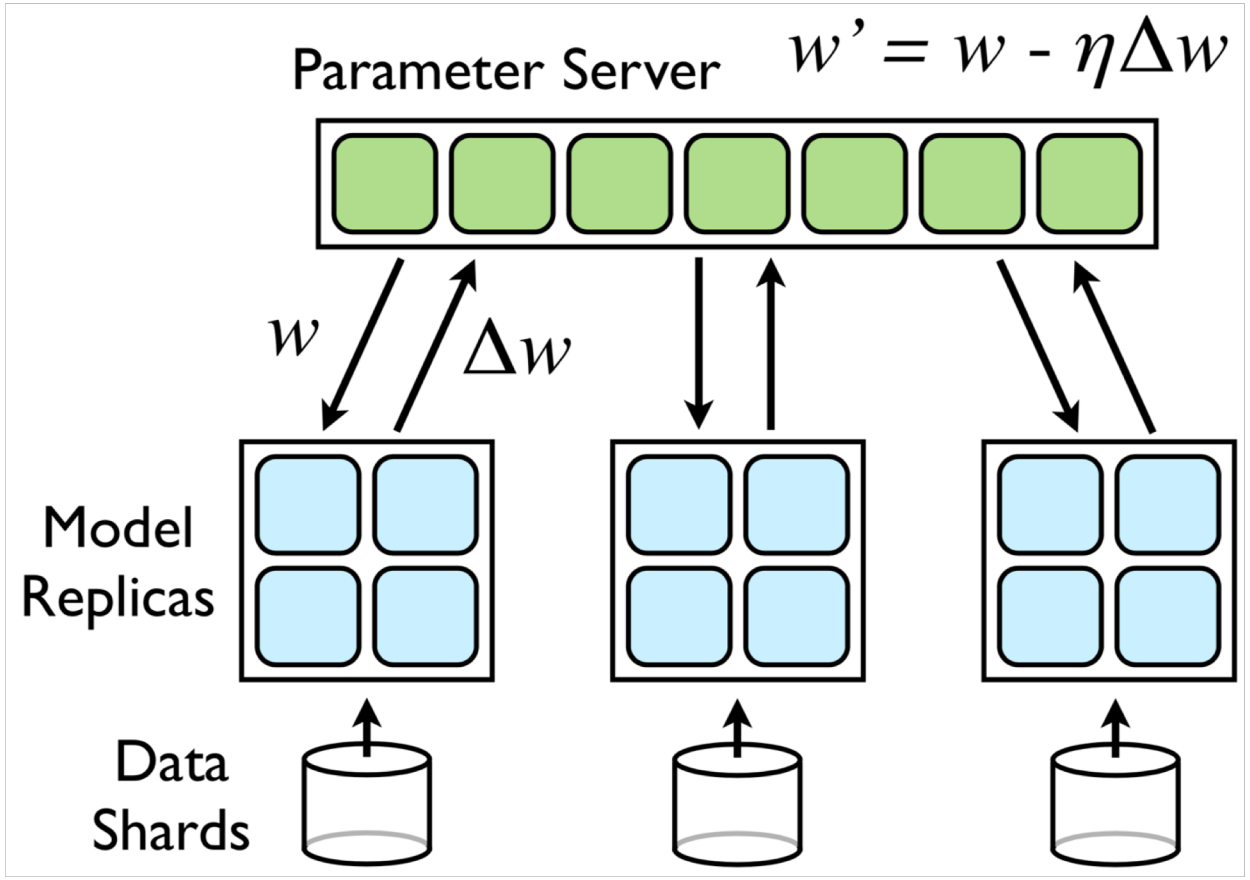
Data parallel (strategy #2): Issues

- How do we manage the weights? Must we replicate them on every node?
- How do we organize the communications required to combine gradients and to distribute weights?
  - How do we reduce the impact of those communications on overall performance?
- We are in effect increasing the minibatch size. What is the impact on training performance?



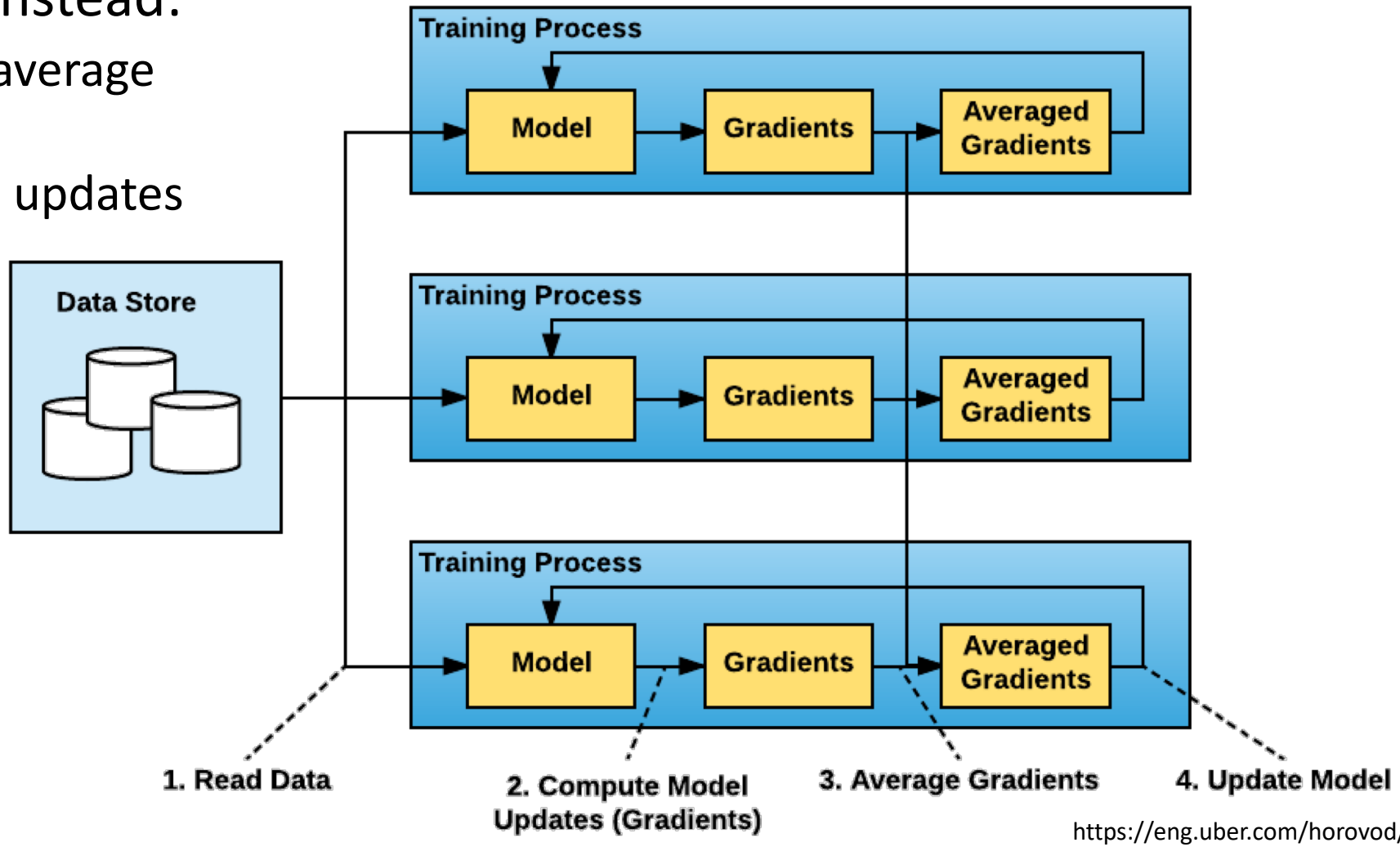
Managing parameters: Parameter server

- Introduce a **parameter server** to handle weight updates
- Request updated parameters from server before each step
- Note that in hybrid schemes, only a subset of parameters are needed at each node
- Advantage: Requires little modification to model; simplifies asynchronous access
- Disadvantage: Central bottleneck



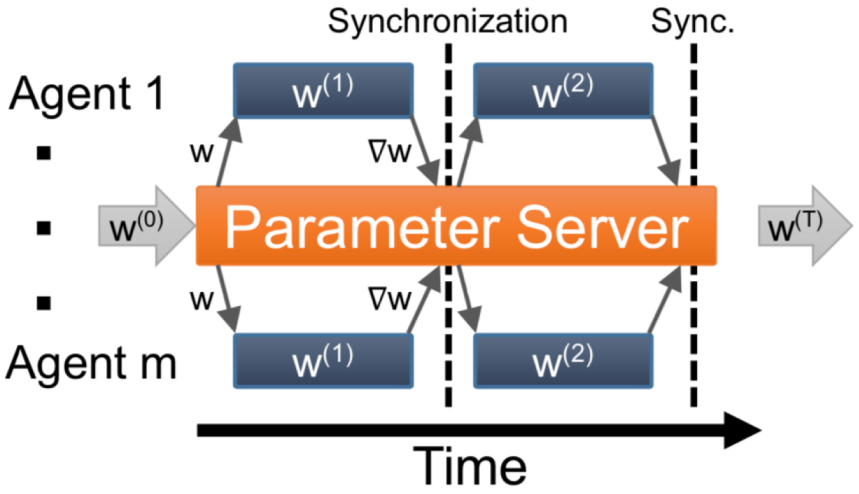
Managing parameters without server

- No parameter server. Instead:
  - Perform all-reduce to average gradients
  - Each model node then updates its own weights
- Advantage: Faster
- Disadvantage: Need efficient all-reduce

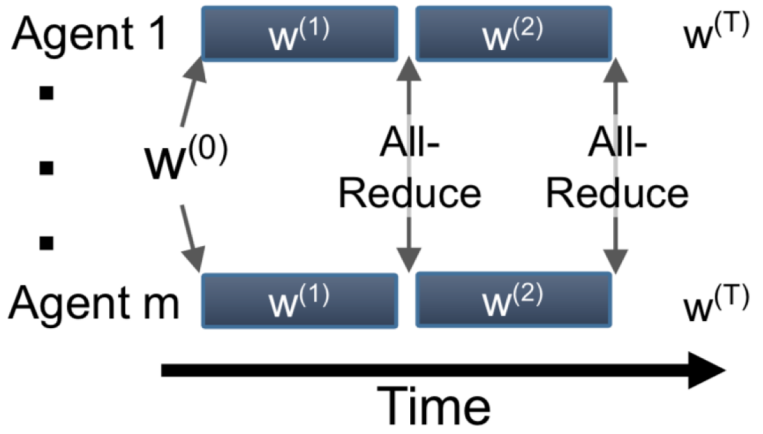


https://eng.uber.com/horovod/

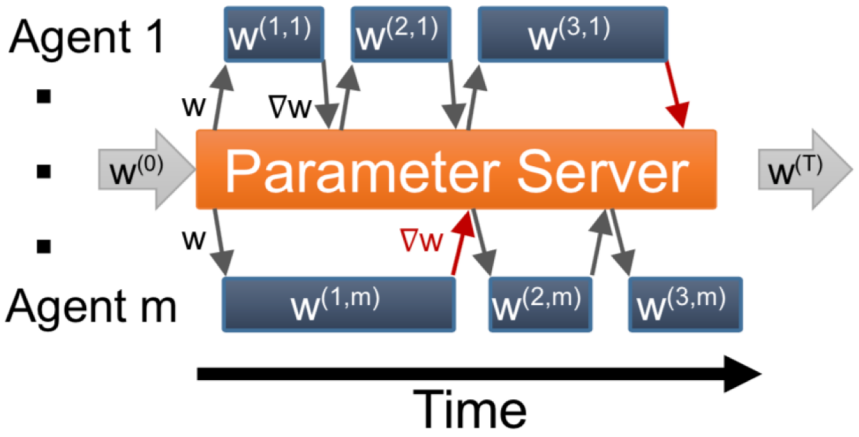
Alternative parameter distribution approaches



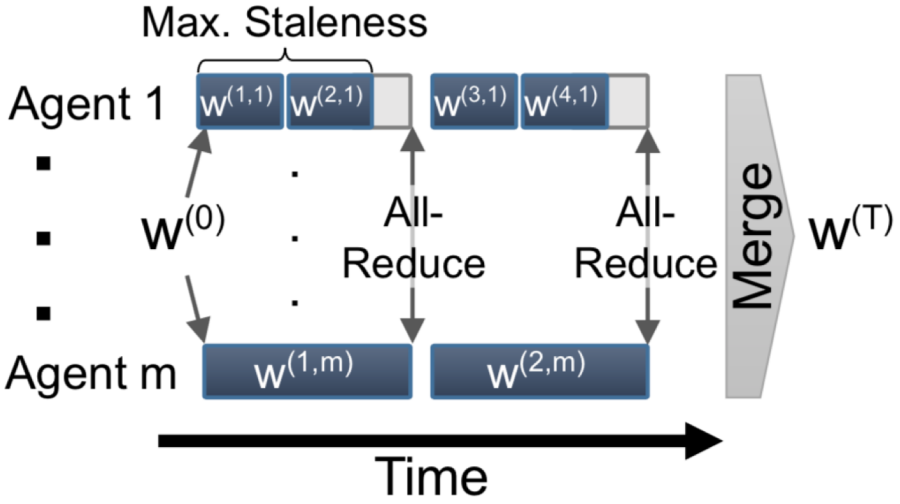
(a) Synchronous, Parameter Server



(b) Synchronous, Decentralized



(c) Asynchronous, Parameter Server



(d) Stale-Synchronous, Decentralized

https://arxiv.org/abs/1802.09941

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**

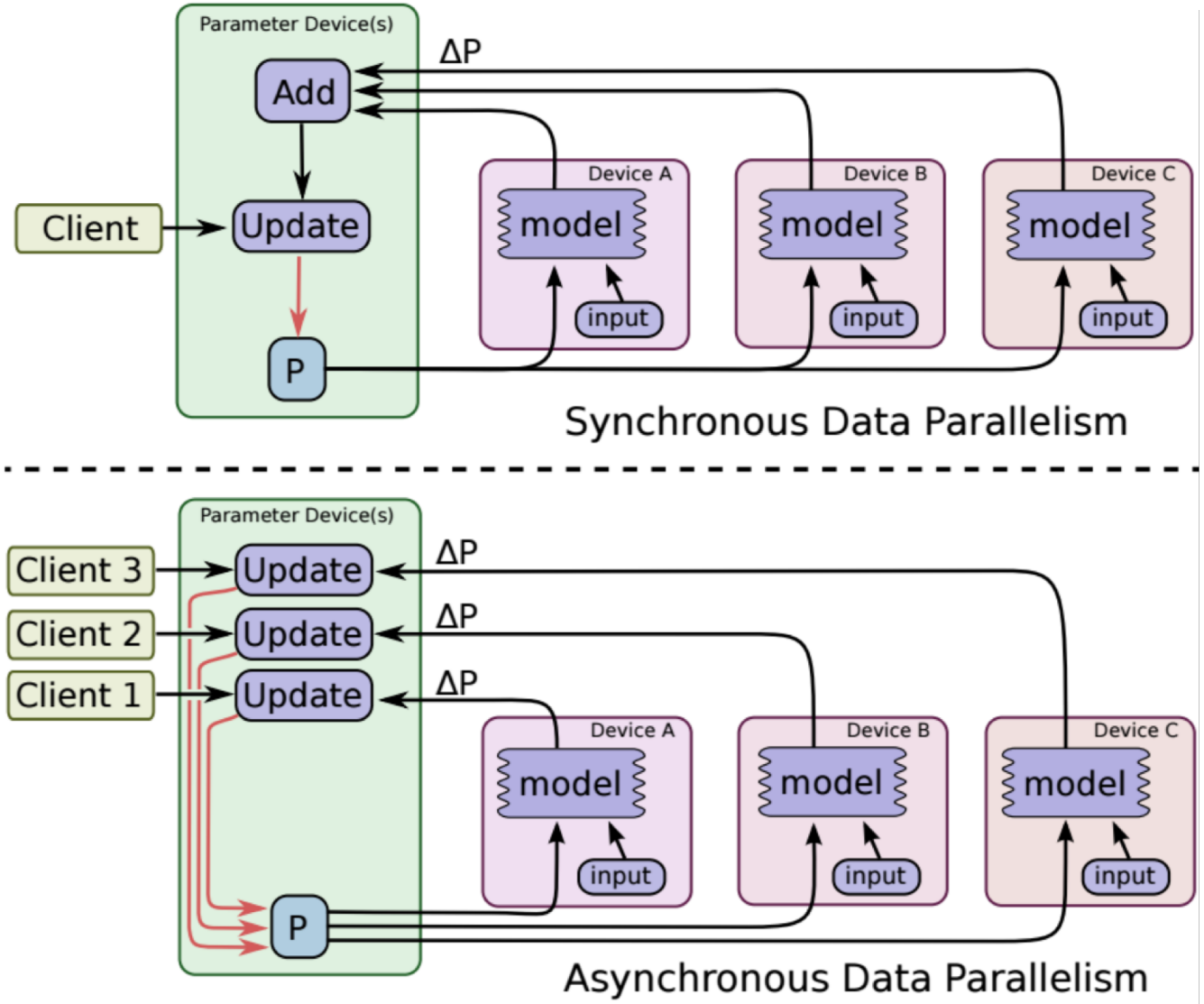
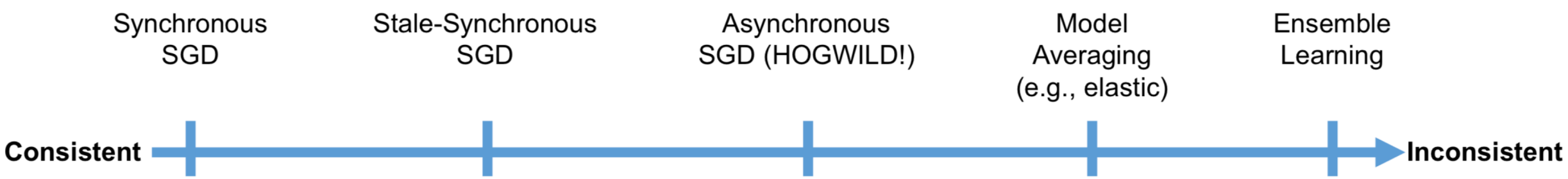


Figure 7: Synchronous and asynchronous data parallel training

https://www.alcf.anl.gov/files/mendygral_ALCF_ScalingDL.pdf



+ (Often) faster convergence

Fig. 23. Parameter Consistency Spectrum

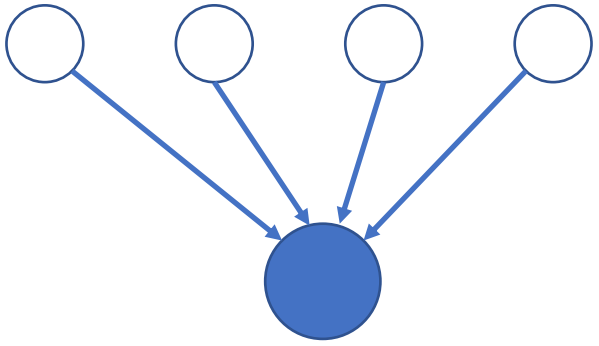
+ Faster
+ Fault tolerant

https://arxiv.org/abs/1802.09941

Efficient communications: Parallel all-reduce

- Compute **y = reduce(x_i)**, where the *N* $\mathbf{x_i}$, each of size *S*, are distributed over *N* nodes, and distribute **y** to every node.
- **reduce(x_i)** = $x_0 \otimes x_1 \otimes \ldots \otimes x_{N-1}$, and $\otimes$ is commutative
- How? Let t_b be the cost of communicating a byte [assuming no contention and no message startup cost]

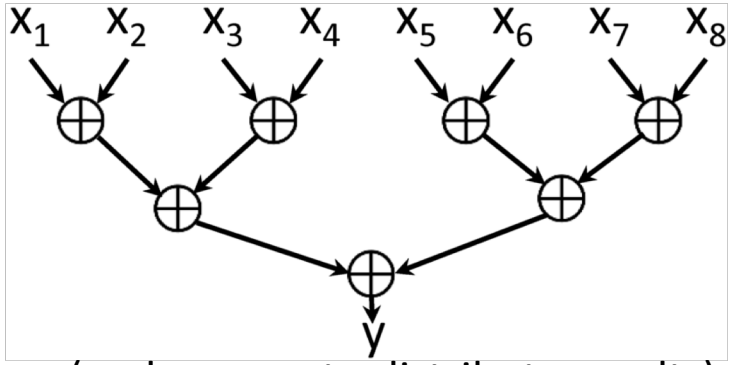
Collect to central node, reduce, distribute



(and reverse to distribute results)

$T_{reduce} = 2\ N\ S\ t_b$

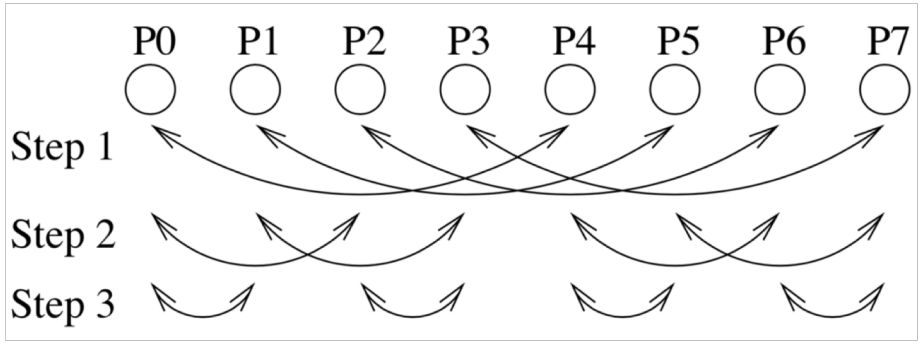
Use binary tree to collect and distribute



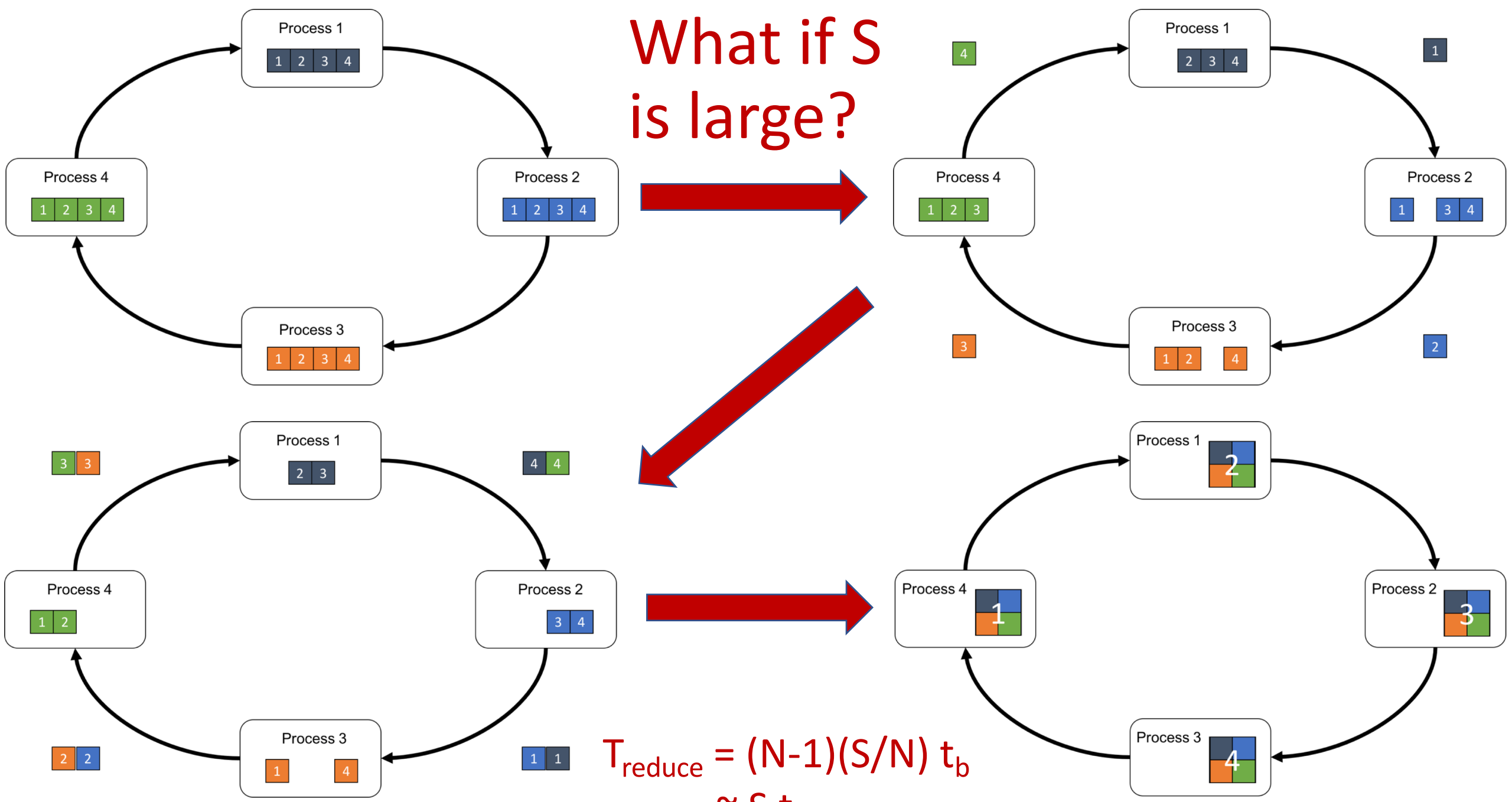
(and reverse to distribute results)

$T_{reduce} = 2\ \log(N)\ S\ t_b$

Repeatedly exchange and reduce



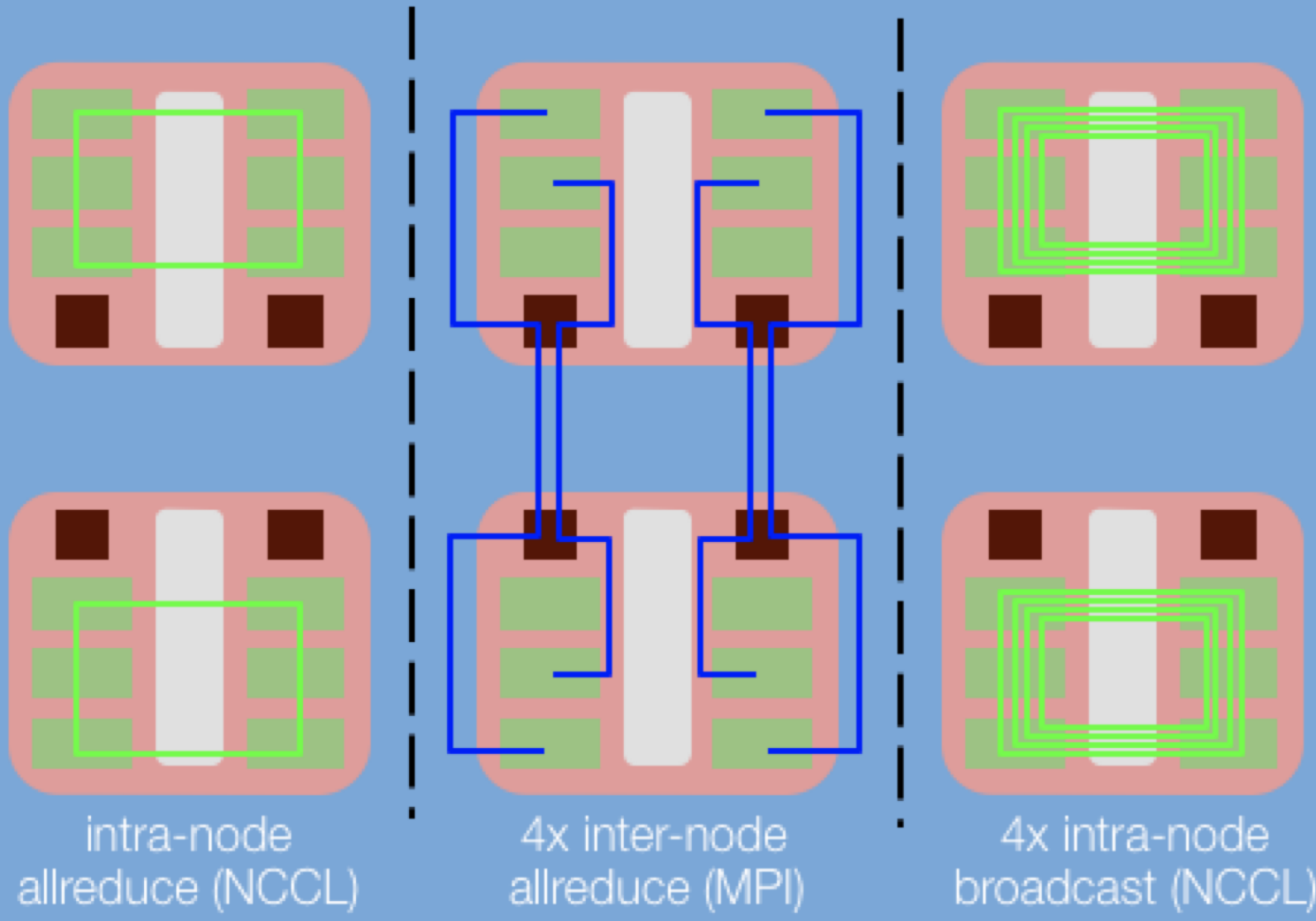
$T_{reduce} = \log(N)\ S\ t_b$



See: http://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

MPI = Message Passing Interface
NCCL = NVIDIA Collective Communication Library

Hybrid All-Reduce



- NCCL uses NVLink for high throughput, but ring-based algorithms latency-limited at scale
- hybrid NCCL/MPI strategy uses strengths of both
- one inter-node allreduce per virtual NIC
- MPI work overlaps well with GPU computation

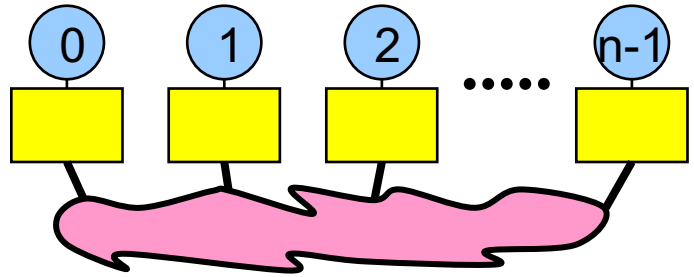
Credit: Steven Farrell, LBNL

Message Passing Interface (MPI)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, total;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Reduce(&rank, &total, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD);
    printf("I am %d of %d with total %d\n", rank, size, total);

    MPI_Finalize();
    return 0;
}
```



Gradient compression

- Simple strategy exchanges gradients every steps
- But not all gradients are equal: some are big, some are small. And do we need every to send every bit of accuracy?
- Thus:
  - Compaction: Only send gradients that exceed a threshold: (index, gradient) pairs
  - Quantization: Instead of a float, send a quantized gradient (1 bit: +/- τ works!)
  - In both cases, accumulate "error" and send periodically
  - Another option: entropy coding of update messages (on indexes)

Table 3. *Performance for varying thresholds τ. Compression ratio is the size of the full gradient divided by the size of our sparse weight updates.*

WER = word error rate

τ	Frames/second (epoch 2-10)	Update size/mini-batch [KB]	Compression ratio	Elapsed [minutes]	Relative WER reduction
2.0	303,000	210	278	182	1.8%
4.0	331,000	69	846	180	1.6%
6.0	334,000	33	1,770	179	1.1%
9.0	335,000	15	3,893	179	0.2%
12.0	335,000	7.8	7,487	165	-0.5%
15.0	341,000	4.5	12,978	162	-1.9%

https://pdfs.semanticscholar.org/a38f/d727637fcfb99a8c7e0df2c819c2d3c6773a.pdf

Pipelining to overlap computation, communication

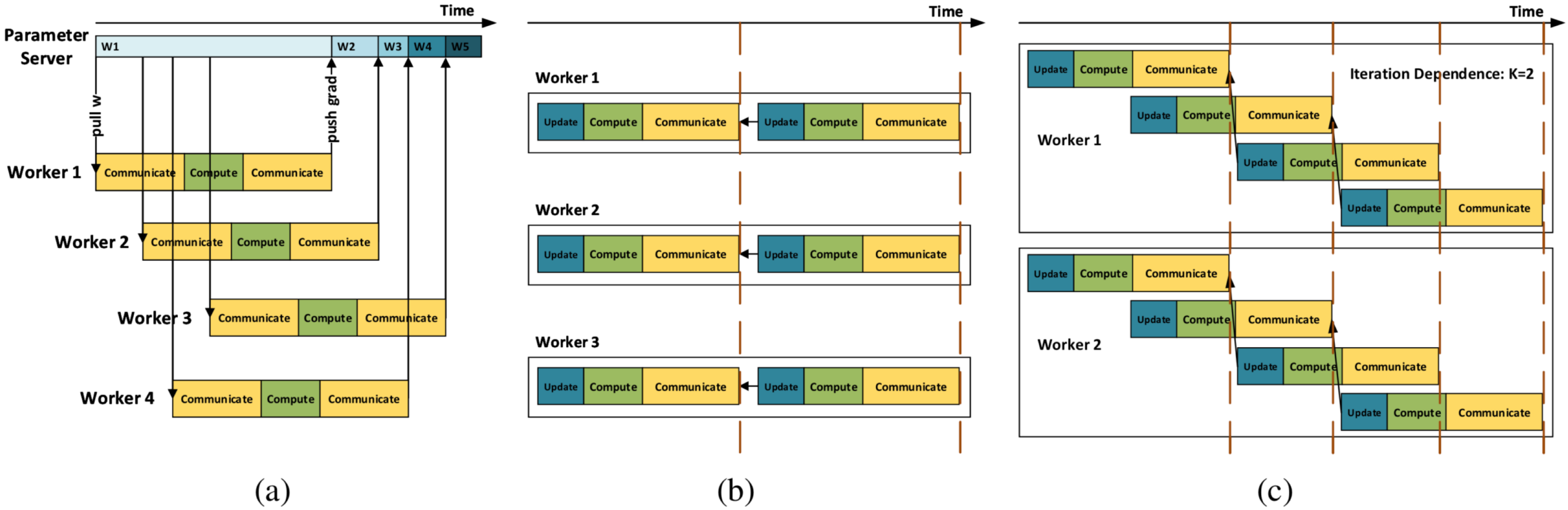
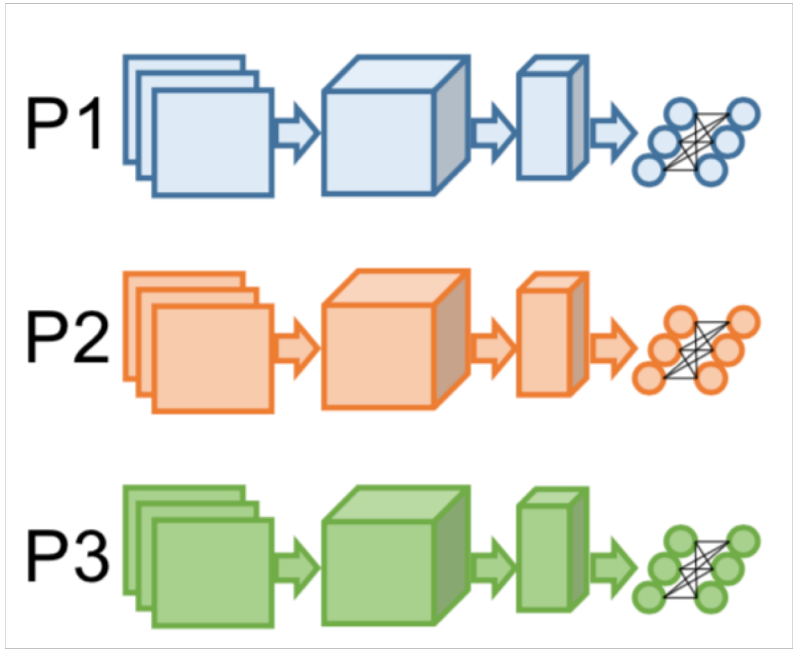


Figure 1: Comparison between different distributed learning frameworks: (a) parameter server with asynchronous training, (b) decentralized synchronous training, and (c) decentralized pipeline training.

https://arxiv.org/pdf/1811.03619.pdf

Data parallel (strategy #2): Issues

- How do we manage the weights? Must we replicate them on every node?
- How do we organize the communications required to combine gradients and to distribute weights?
  - How do we reduce the impact of those communications on overall performance?
- **We are in effect increasing the minibatch size. What is the impact on training performance?**



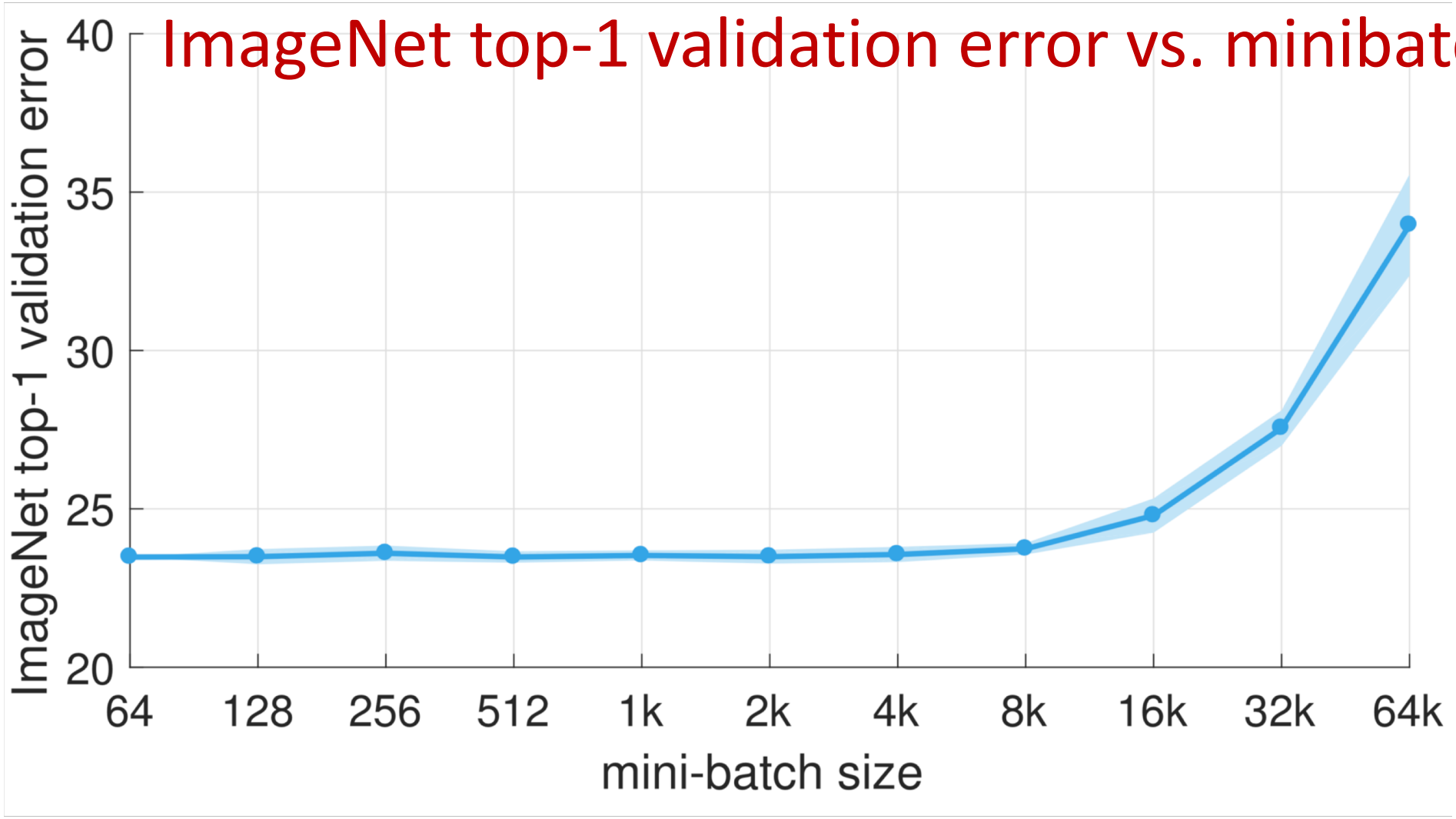
Challenges of increased minibatch size

"Due to the highly non-linear nature of the objective function, reasonable convergence can only be achieved by [using] *minibatches* of randomly sampled frames from the training corpus. Minibatch sizes of 1000 or less lead to best results in our experiments." [Chen et al., http://bit.ly/2Dvt51J]

- Chen et al. claim that increasing minibatch size above a limit causes convergence to slow notably, and eventually fail
- Clearly complex relationships between minibatch size, parallelization, single node performance, learning rate, and convergence
- Fortunately:

6.1.1 Neural Architecture Support for Large Minibatches. By applying various modifications to the training process, recent works have successfully managed to increase minibatch size to 8k samples [83], 32k samples [249], and even 64k [218] without losing considerable accuracy.

https://arxiv.org/abs/1802.09941



Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. Our techniques enable a linear reduction in training time with ~90% efficiency as we scale to large minibatch sizes, allowing us to train an accurate 8k mini-batch ResNet-50 model in 1 hour on 256 GPUs.

https://arxiv.org/pdf/1706.02677.pdf

Increasing minibatch size: Goyal et al., 2017

- **Linear scaling**: "When minibatch size is multiplied by k, multiply learning rate by k."
- **Warmup**: "for large minibatches (e.g., 8k) the linear scaling rule breaks down when the network is changing rapidly, which commonly occurs in early stages of training. We find that this issue can be alleviated by … *warmup*, namely, a strategy of using less aggressive learning rates at the start of training." [e.g., first 5 epochs]

https://arxiv.org/pdf/1706.02677.pdf

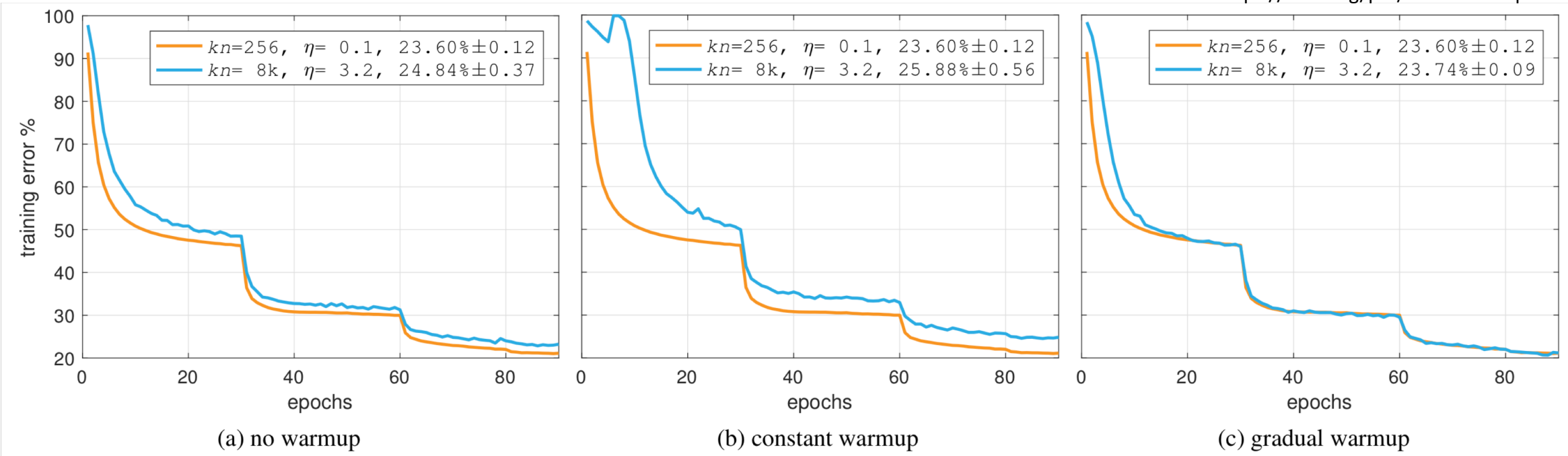


Figure 2. **Warmup.** Training error curves for minibatch size 8192 using various warmup strategies compared to minibatch size 256. *Validation* error (mean±std of 5 runs) is shown in the legend, along with minibatch size kn and reference learning rate η.

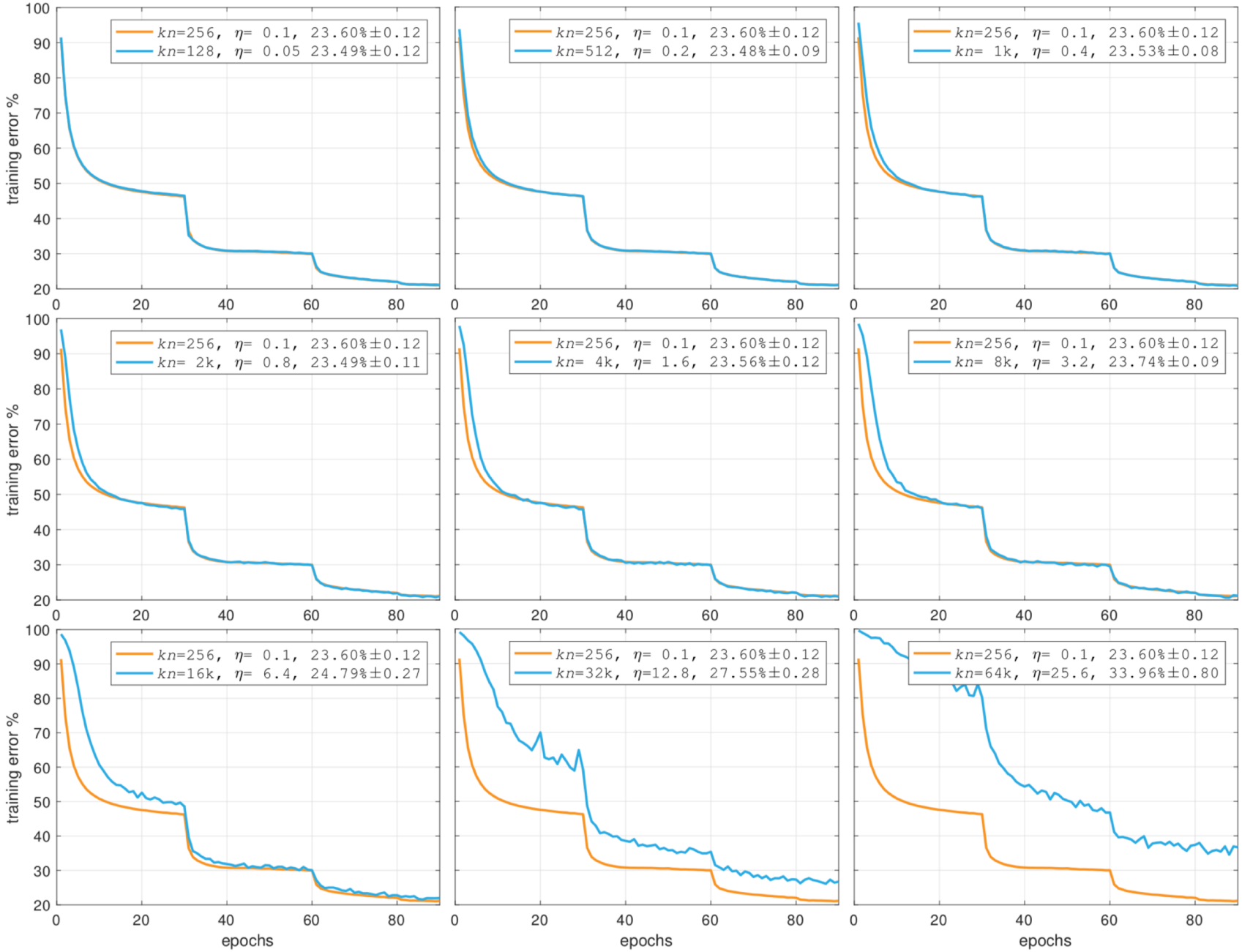


Figure 3. **Training error *vs*. minibatch size.** Training error curves for the 256 minibatch baseline and larger minibatches using gradual warmup and the linear scaling rule. Note how the training curves closely match the baseline (aside from the warmup period) up through 8k minibatches. *Validation* error (mean±std of 5 runs) is shown in the legend, along with minibatch size kn and reference learning rate η.

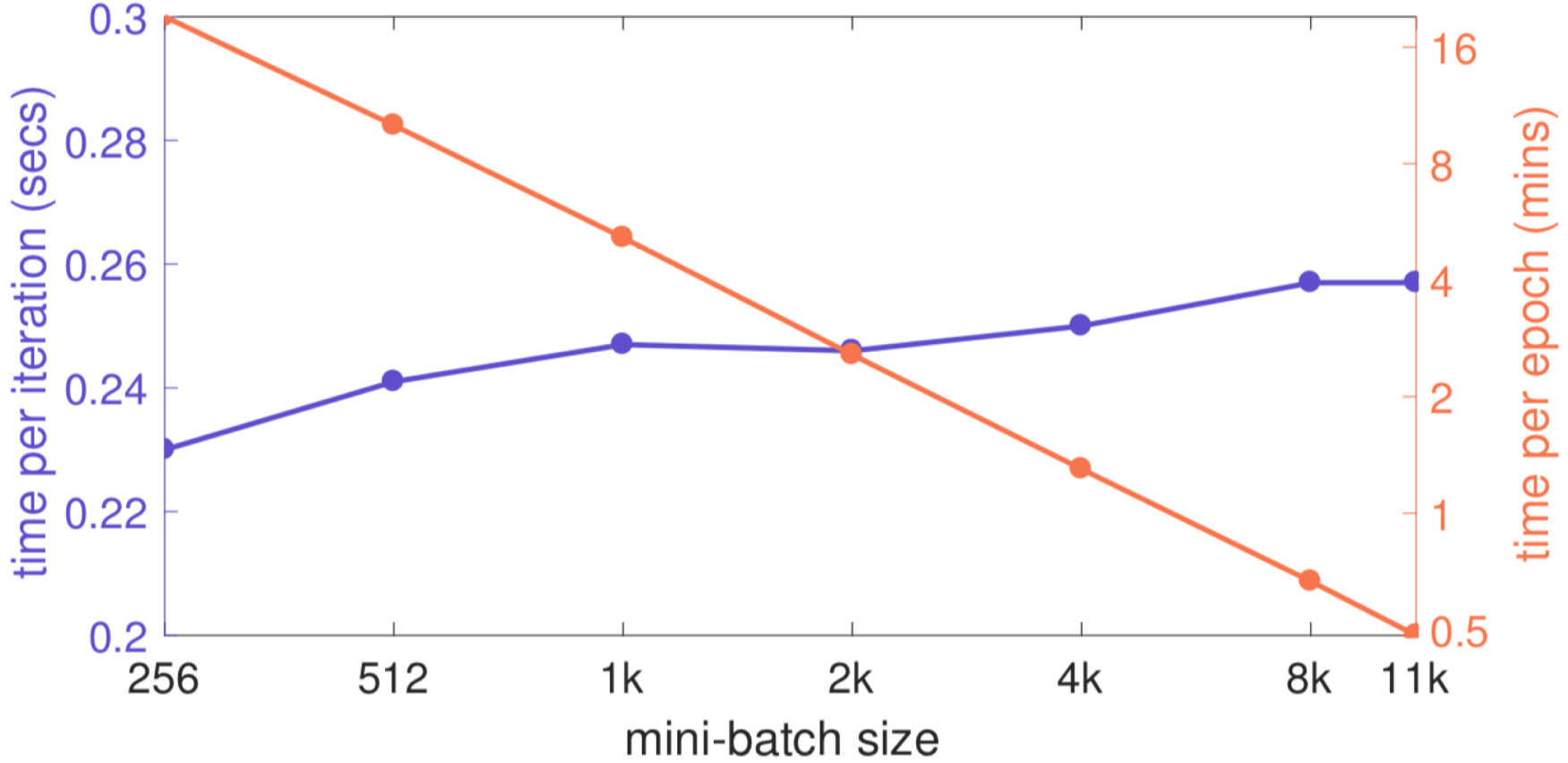
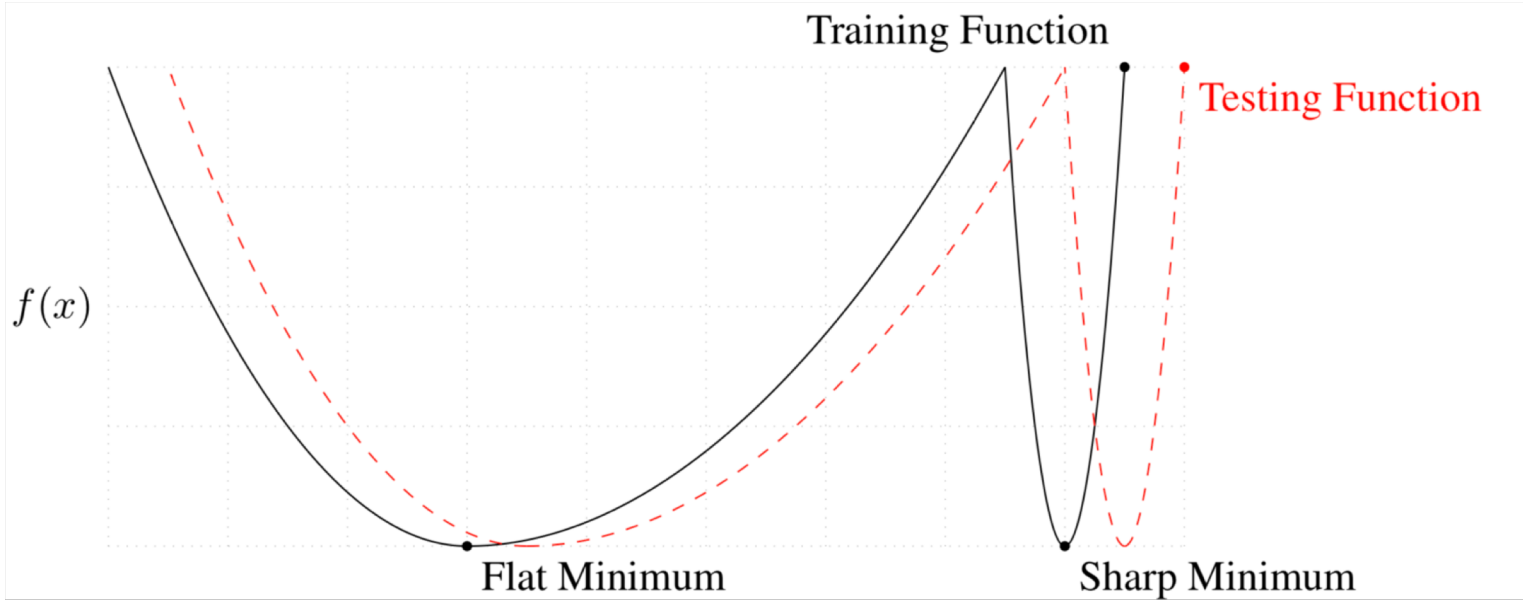


Figure 7. **Distributed synchronous SGD timing.** Time per iteration (seconds) and time per ImageNet epoch (minutes) for training with different minibatch sizes. The baseline ($kn = 256$) uses 8 GPUs in a single server , while all other training runs distribute training over ($kn/256$) server. With 352 GPUs (44 servers) our implementation completes one pass over all ~1.28 million ImageNet training images in about 30 seconds.

Keskar et al., 2017

- When using a larger batch there is a degradation in the quality of the model, as measured by its ability to generalize
- Large-batch methods tend to converge to sharp minimizers of the training and testing functions—and as is well known, sharp minima lead to poorer generalization.
- In contrast, small-batch methods consistently converge to flat minimizers; our experiments support a commonly held view that this is due to the inherent noise in the gradient estimation



https://openreview.net/pdf?id=H1oyRlYgg

Why is there a generalization gap with large batch (LB) relative to small batch (SB)?

Possible reasons:

1) LB methods over-fit the model;
2) LB methods are attracted to saddle points;
3) **LB methods lack the *explorative* properties of SB methods and tend to zoom-in on the minimizer closest to the initial point;**
4) **SB and LB methods converge to qualitatively different minimizers with differing generalization properties.**

Keskar et al. argue for the latter two reasons.

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 (e.g., 1024 nodes each with local batch size of 32)

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

Parameters: base LR γ_0, momentum m, weight decay β, LARS coefficient η, 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 λ^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

https://arxiv.org/pdf/1708.03888.pdf

Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes

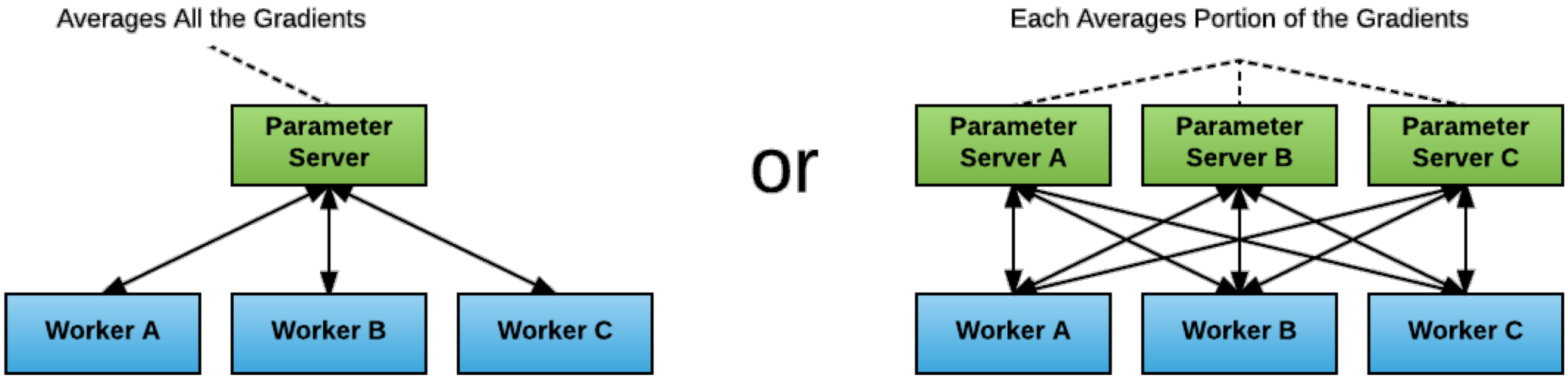
Table 1: 90-epoch training time and single-crop validation accuracy of ResNet-50 for ImageNet reported by different teams.

Team	Hardware	Software	Minibatch size	Time	Accuracy
He *et al.* [5]	Tesla P100 × 8	Caffe	256	29 hr	75.3 %
Goyal *et al.* [4]	Tesla P100 × 256	Caffe2	8,192	1 hr	76.3 %
Codreanu *et al.* [3]	KNL 7250 × 720	Intel Caffe	11,520	62 min	75.0 %
You *et al.* [10]	Xeon 8160 × 1600	Intel Caffe	16,000	31 min	75.3 %
This work	Tesla P100 × 1024	Chainer	32,768	15 min	74.9 %

https://arxiv.org/pdf/1711.04325.pdf

Parallelism in practice: TensorFlow

- Standard distributed TensorFlow uses a parameter server approach to averaging gradients
  - Each process is either a worker or a parameter server
  - Workers process training data, compute gradients, and send them to parameter servers to be averaged



- Difficulties:
  - Choosing right ratio of parameter servers to workers
  - Configuring and managing different servers

https://eng.uber.com/horovod/

Parallelism in practice: TensorFlow + Horovod

- Replaces TensorFlow's native optimizer class with a new class in which all-reduce is performed between gradient computation and model update
- Uses ring all-reduce and native HPC communication methods (MPI, NCCL)

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

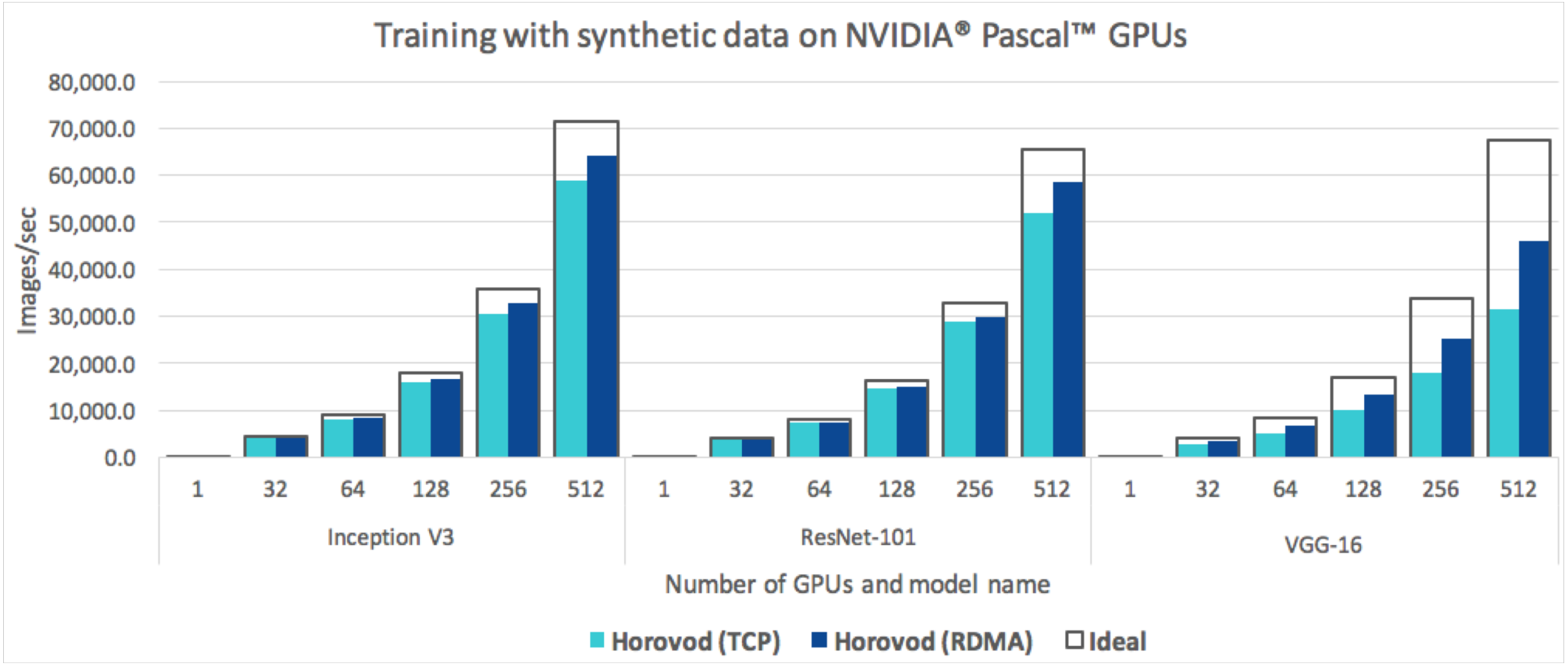
Compute intensive

Communication intensive

Typically not much compute

https://github.com/uber/horovod

Parallelism in practice: TensorFlow + Horovod



RDMA = Remote Direct Memory Access

https://github.com/uber/horovod

Parallelism in practice: Parameter server vs. MPI all-reduce

Framework	Distributed Communication Mechanism
Tensorflow	PS + mpi-allreduce(baidu allreduce) (uber horovod)
MXNet	PS + Horovod in progress
Caffe	mpi-allreduce
Torch + PyTorch	mpi-allreduce
Chainer	mpi-allreduce(mpi4py)

http://bit.ly/2BhhQbG

Mxnet/Horovod integration: Why

- **Usability** - Users do not have to experiment with number of workers and number of servers to get best performance out-of-the-box.
- **Performance** - Horovod + Tensorflow has shown 2x performance of Distributed Tensorflow [1], so we expect it to show similar gains.
- **Cost savings** - Parameter servers are not needed when they use Horovod.
- **Simplified architecture** - Leverage battle-tested libraries such as MPI and NCCL, as well as network optimizations such as RDMA.
- **Profiler** - Horovod has an excellent profiler for finding bottlenecks.
- **Online learning** - Due to its MPI paradigm, Horovod can save checkpoints which enables online learning and fine-tuning of your model. With parameter server, it takes some additional work to save Optimizer state located on servers, but with Horovod this feature comes for free. Note: this feature is not currently supported.
- **Community** - Horovod is a way for MXNet to leverage the Deep Learning community for advancements in distributed training, and for increasing MXNet's visibility.

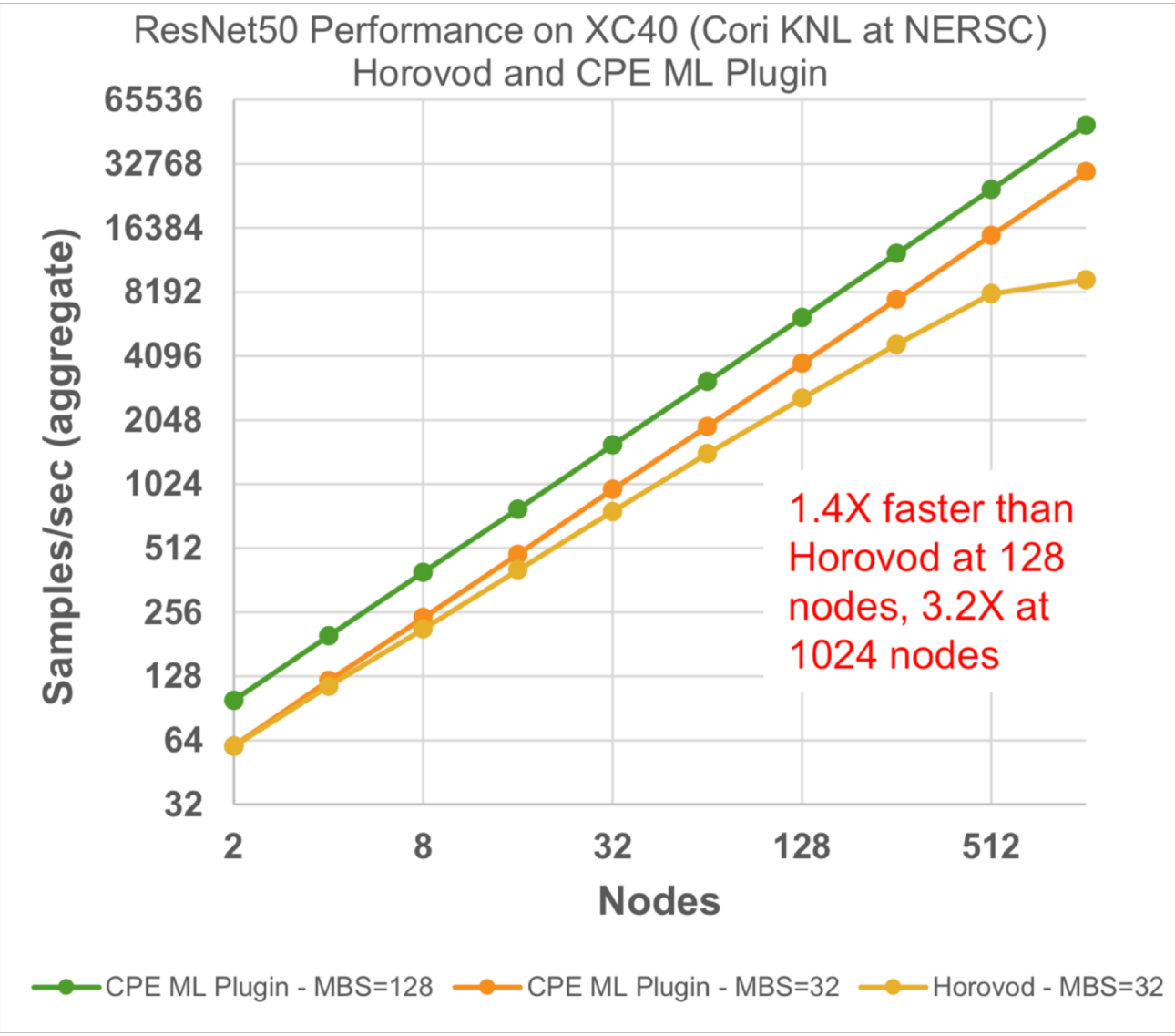
https://cwiki.apache.org/confluence/display/MXNET/Horovod-MXNet+Integration

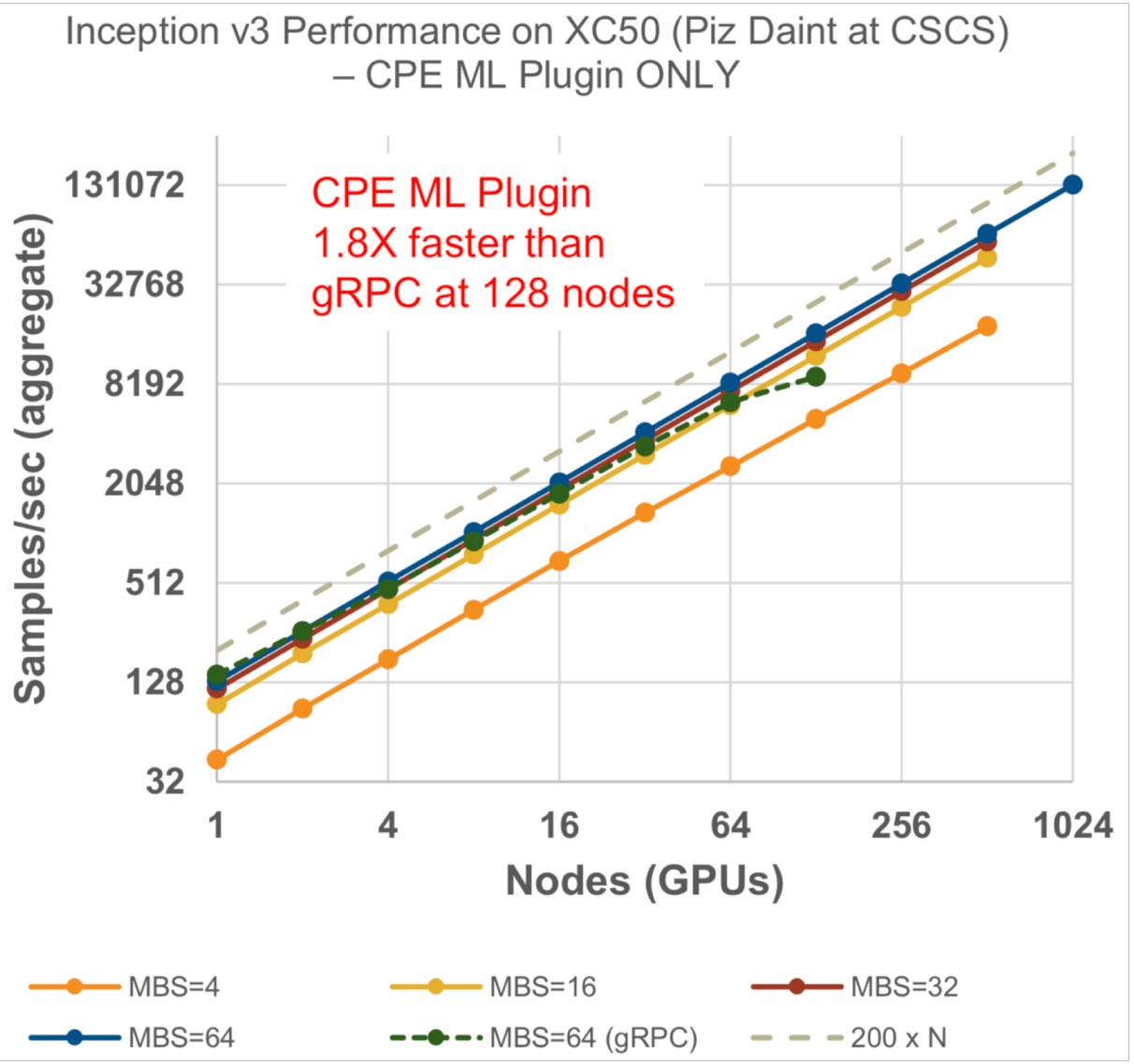
Disadvantages of Horovod

For sparse models that are too big to fit onto one machine, we do not yet know of a good way to implement this using Horovod. While Horovod does support dense model and sparse weights with TensorFlow, we are also interested in sparse model and sparse weights. For some of these problems, parameter server may be a more natural programming model for thinking about the problem: The model can be saved on the server. Workers pull only the model rows they need.

https://cwiki.apache.org/confluence/display/MXNET/Horovod-MXNet+Integration

Horovod/CPE ML Plugin: Throughput scaling





CPE = Cray Programming Environment

Parallelism in practice: Keras on multiple GPUs on a single node

```
from keras.utils.training_utils import multi_gpu_model
G = number-of-GPUs-to-use-for-training
# If G > 1:
with tf.device("/cpu:0"):
    # initialize the model
    model = MiniGoogLeNet.build(width=32, height=32, depth=3, classes=10)
# make the model parallel
model = multi_gpu_model(model, gpus=G)
H = model.fit_generator(
    aug.flow(trainX, trainY, batch_size=64 * G),
    validation_data=(testX, testY),
    steps_per_epoch=len(trainX) // (64 * G),
    epochs=NUM_EPOCHS)
```

http://bit.ly/2ORWtRP

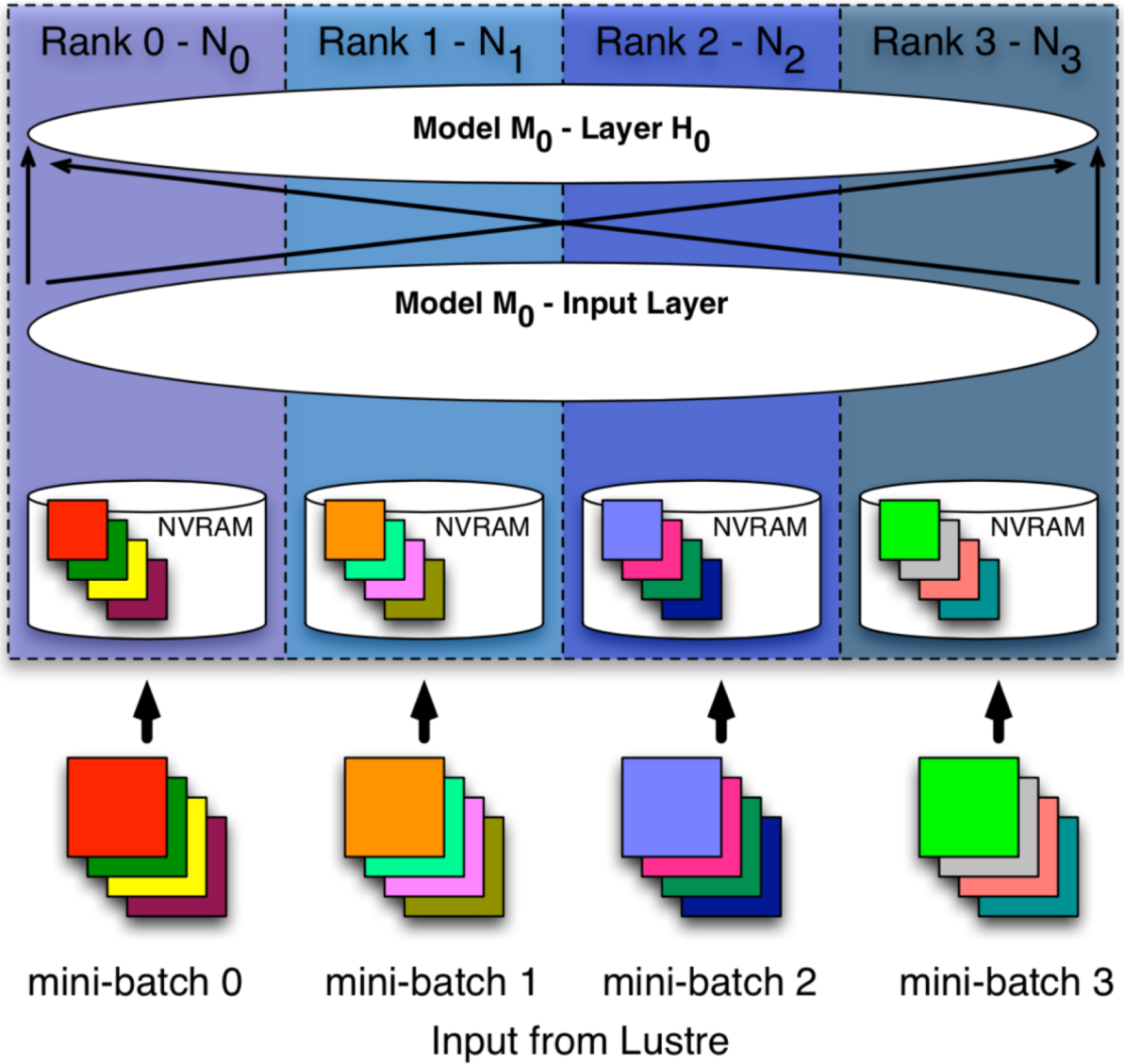
Parallelism in practice: MXnet

- By default, *MXNet* uses data parallelism to partition the workload over multiple devices. Each of I devices receives a copy of the complete model and trains it on *1/N* of the data. Gradients and updated model are communicated across these devices.
- MXNet also supports model parallelism

Parallelism in practice: LBANN*

Distributing DNN across HPC nodes

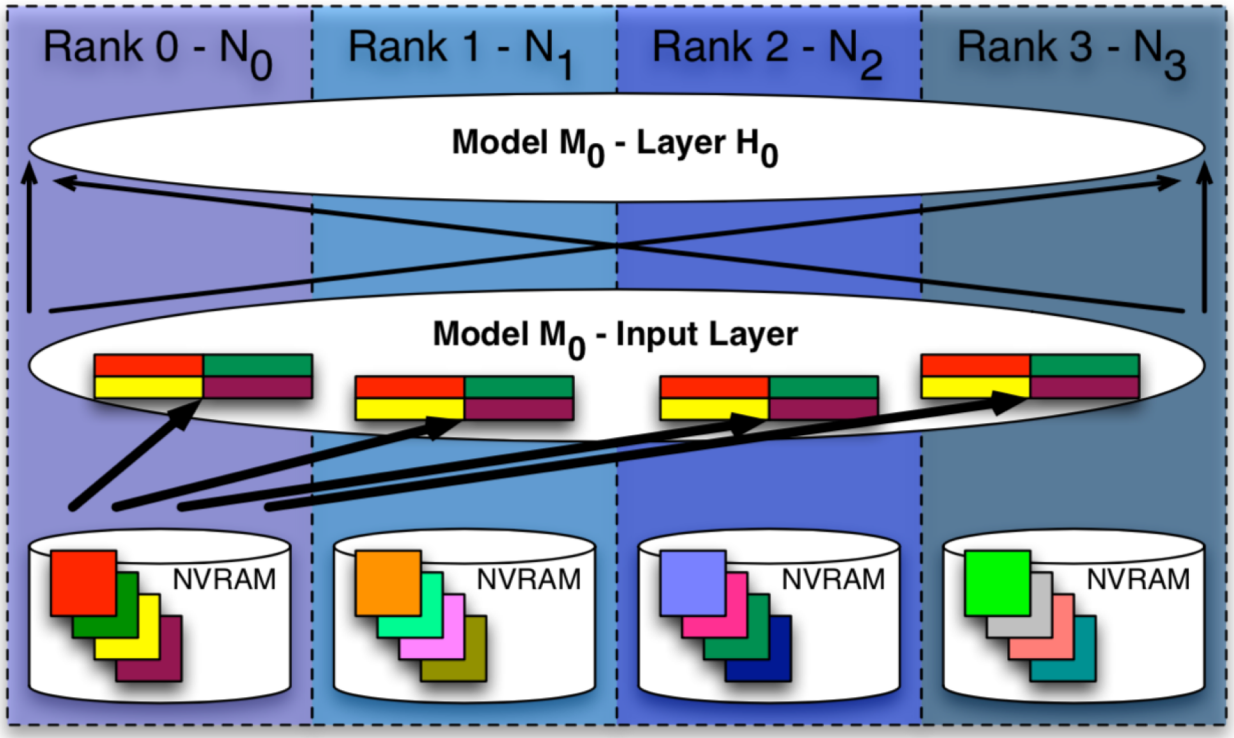
- Each layer of model is distributed across nodes
  - Distributed matrix library (Elemental) provides dense matrix operations
- Input data is staged into node-local NVRAM
  - Each node stages a separate mini-batch

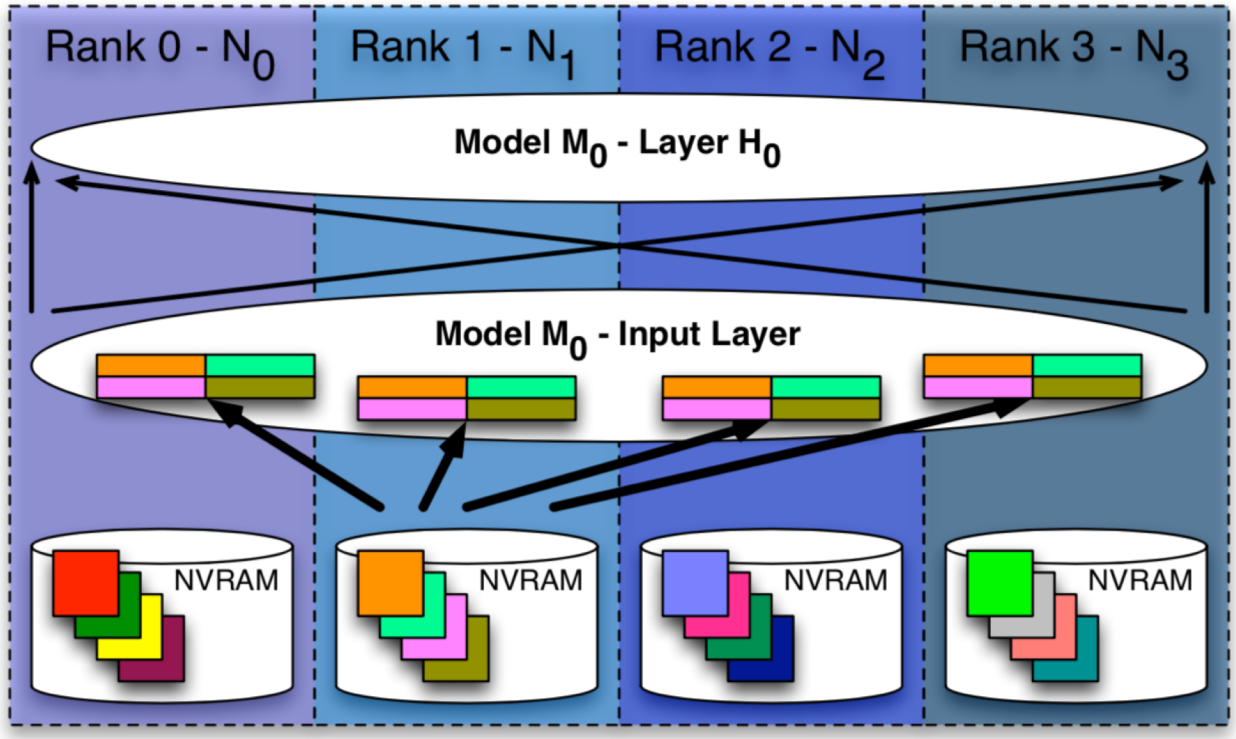


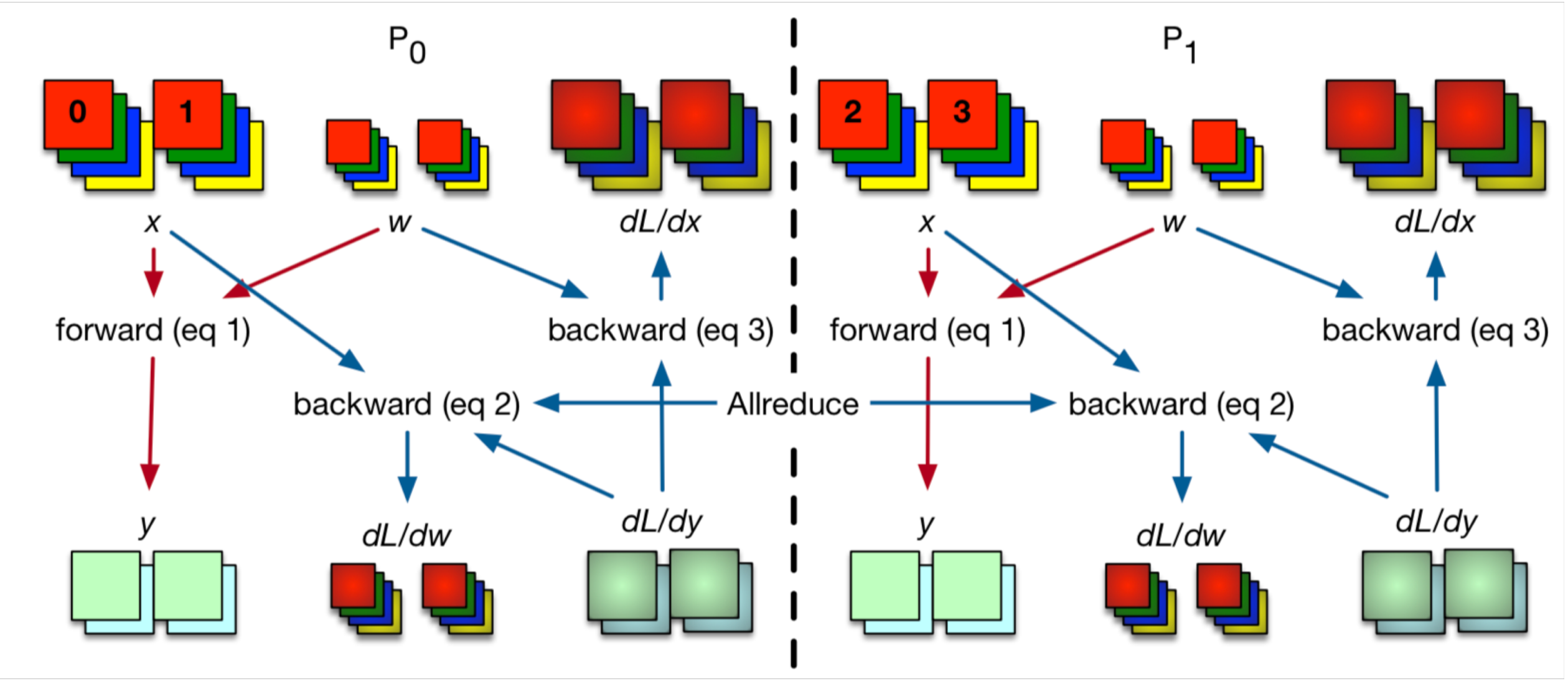
*Livermore Big Artificial Neural Network

Distributing data

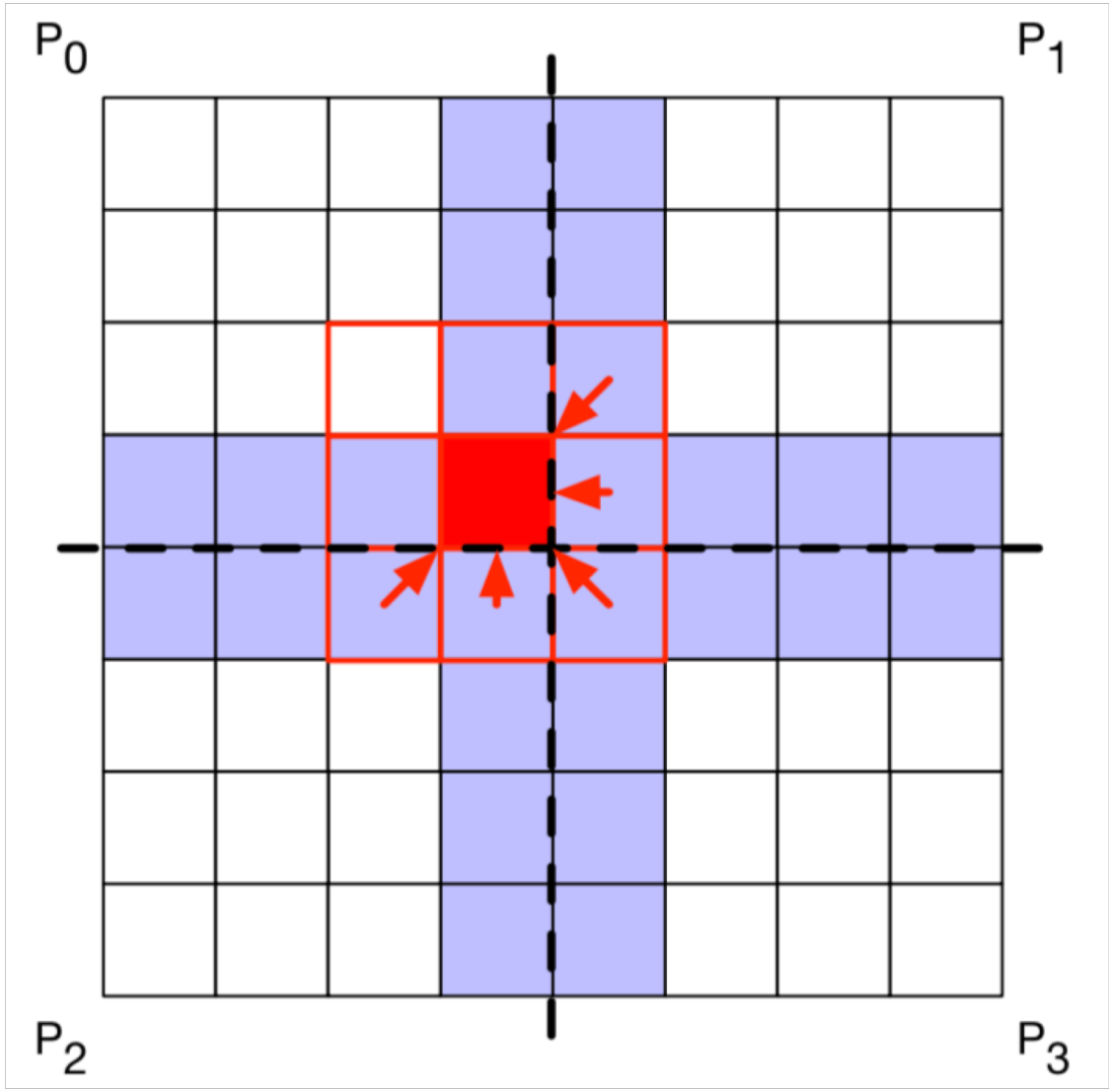
- Active mini-batch is replicated from source node to each MPI rank
- First layer multiplies distributed matrix with replicated input data



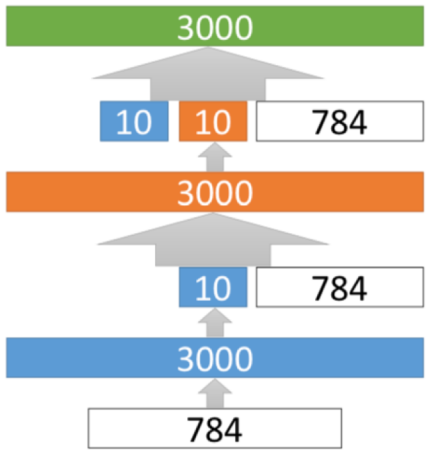




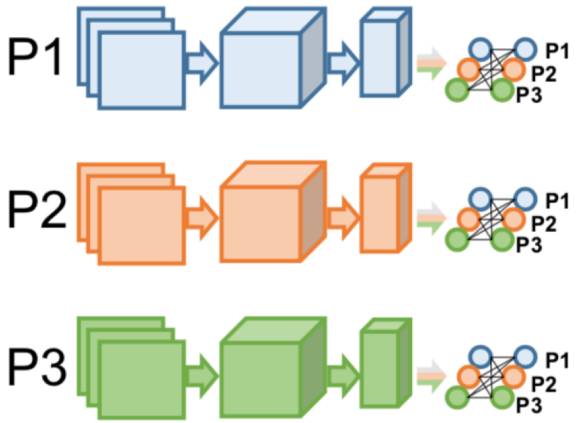
LBANNN forward & backprop phases (red & blue arrows) for sample parallelism with two processors and a global mini-batch of size 4.



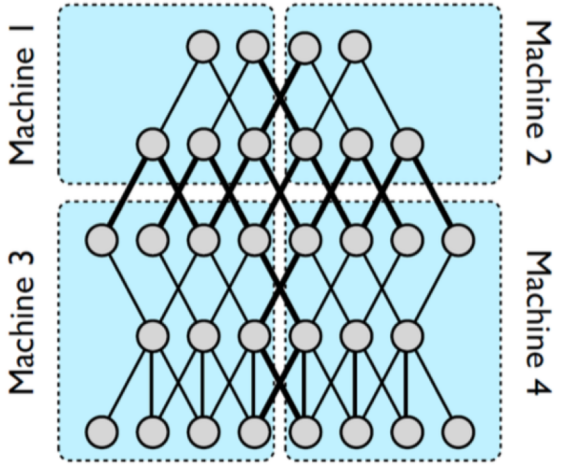
LBANN example halo exchange for spatial parallelism on four processors. The solid red box is where the convolutional filter is centered, red arrows indicate data movement.



(a) Deep Stacking Network [62]



(b) Hybrid Parallelism



(c) DistBelief Replica [56]

Fig. 17. Pipelining and Hybrid Parallelism Schemes

https://arxiv.org/abs/1802.09941

Future challenges

- More parallelism continues to be needed
- Networks are becoming more complex