Lecture 19:

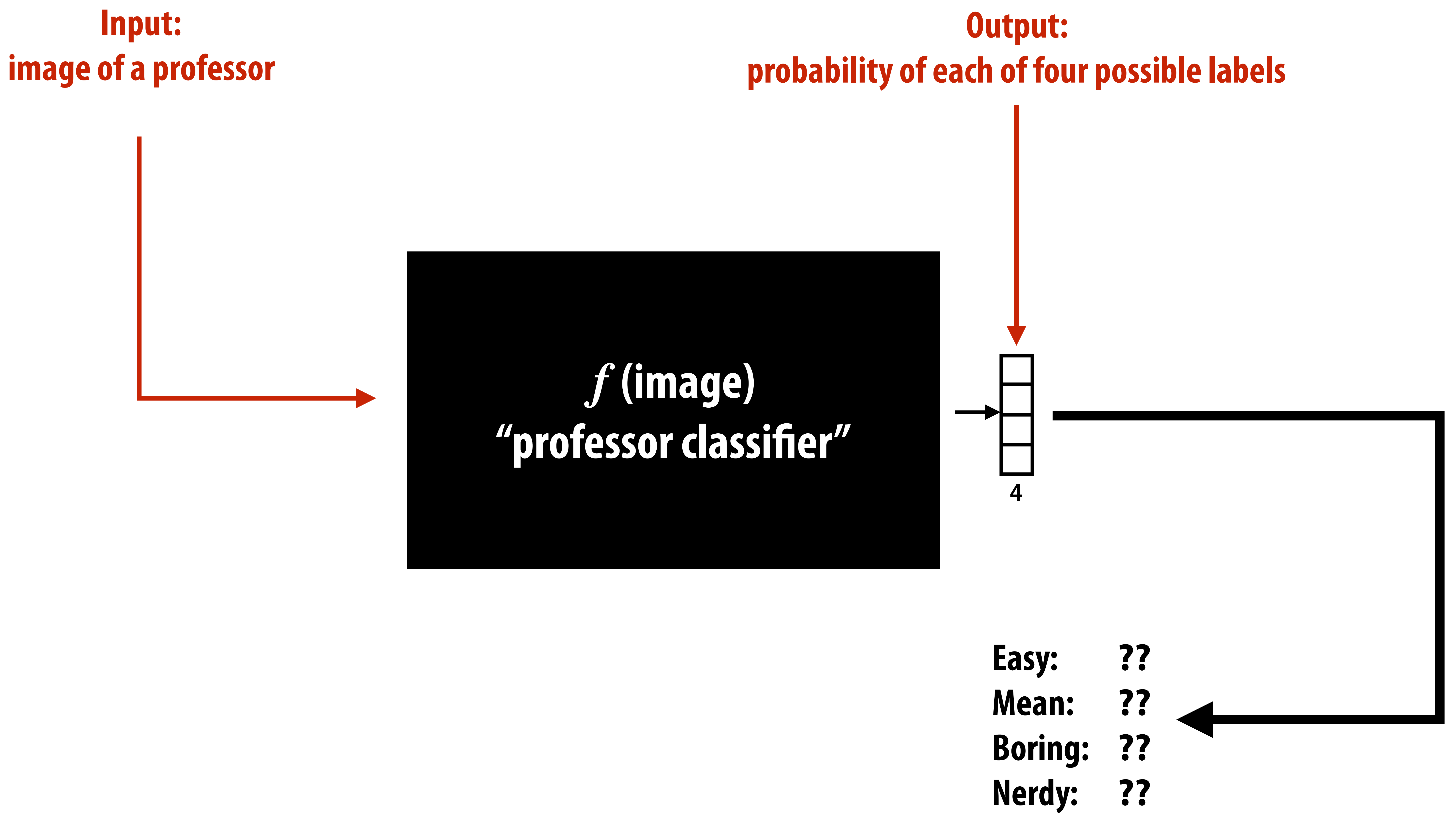
Parallel Deep Network Training + Course Recap

Parallel Computing

Stanford CS149, Winter 2019

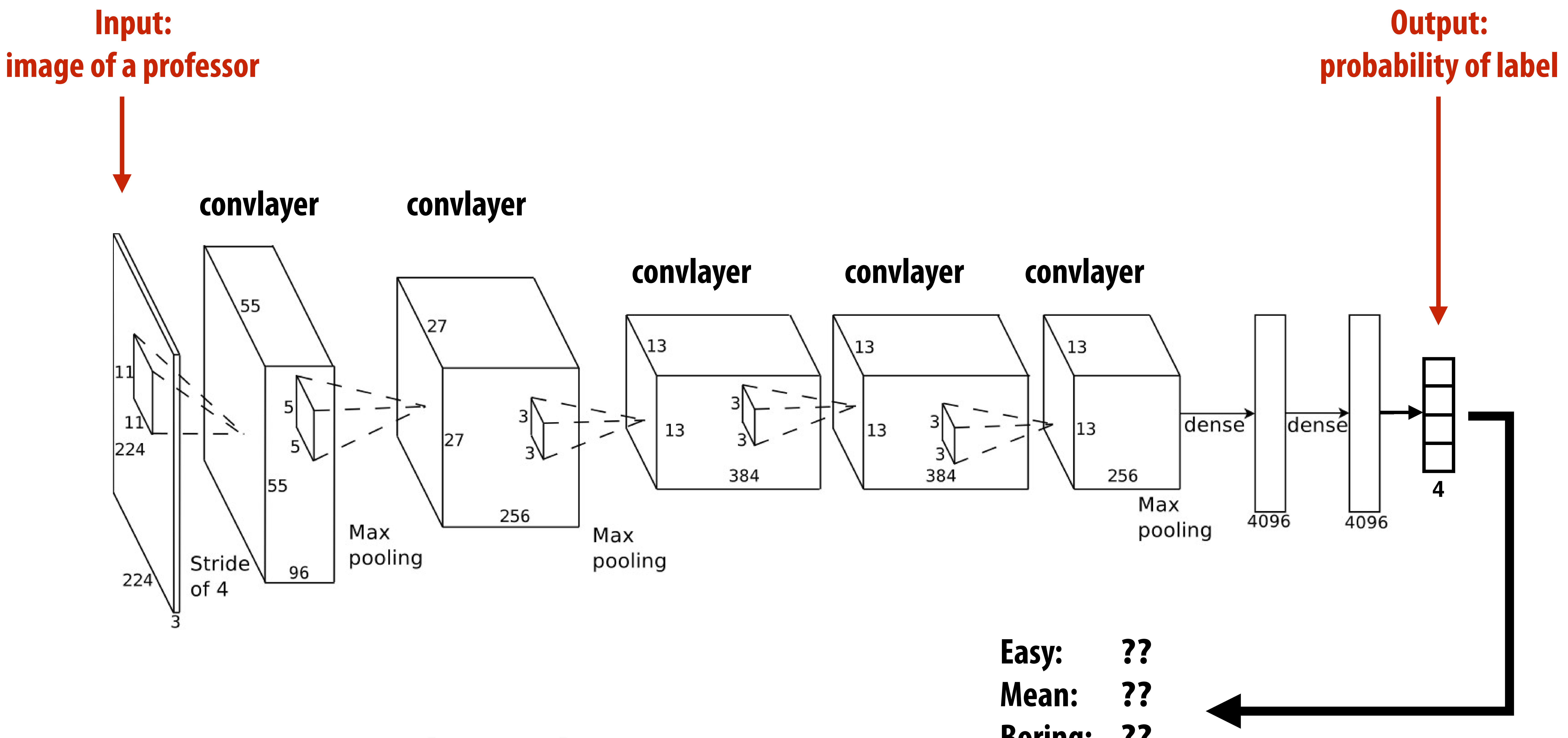
Professor classification task

Classifies professors as easy, mean, boring, or nerdy based on their appearance.



Professor classification network

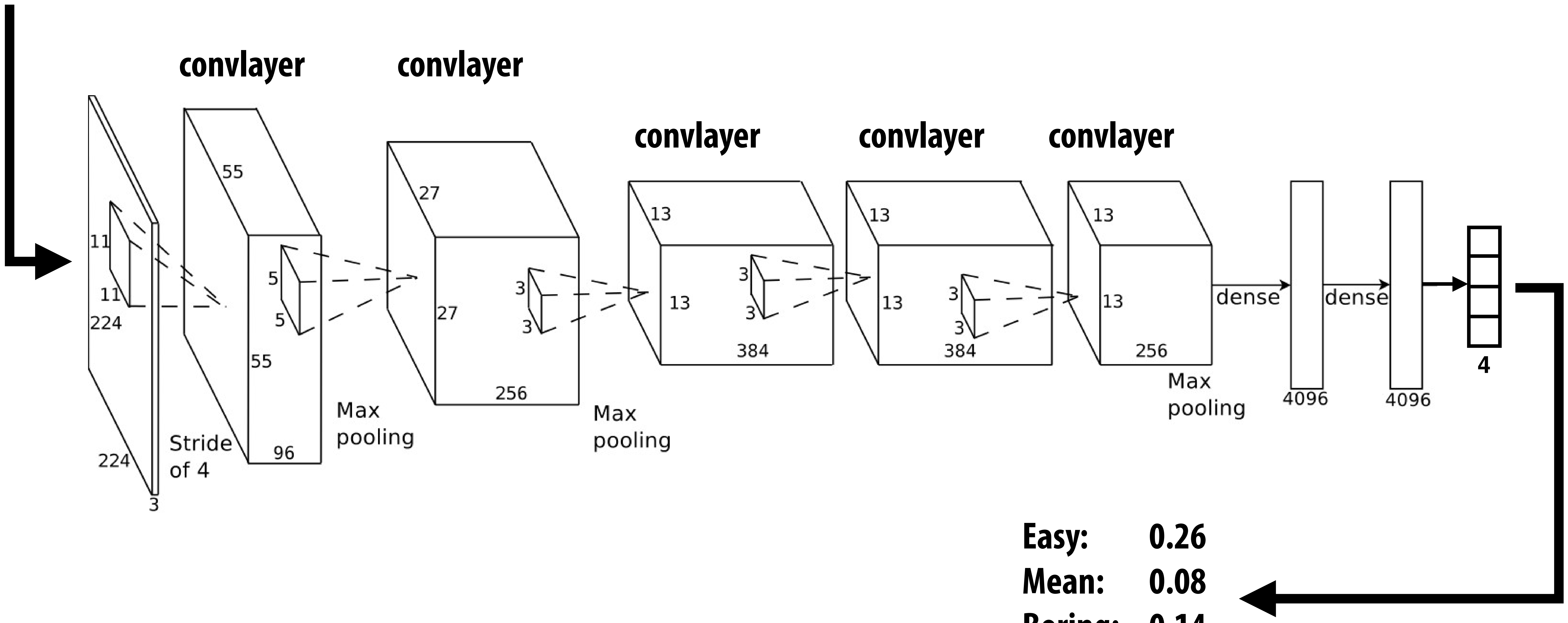
Classifies professors as easy, mean, boring, or nerdy based on their appearance.



Easy:	??
Mean:	??
Boring:	??
Nerdy:	??

Recall: large networks may have 10's-100's of millions of parameters

Professor classification network



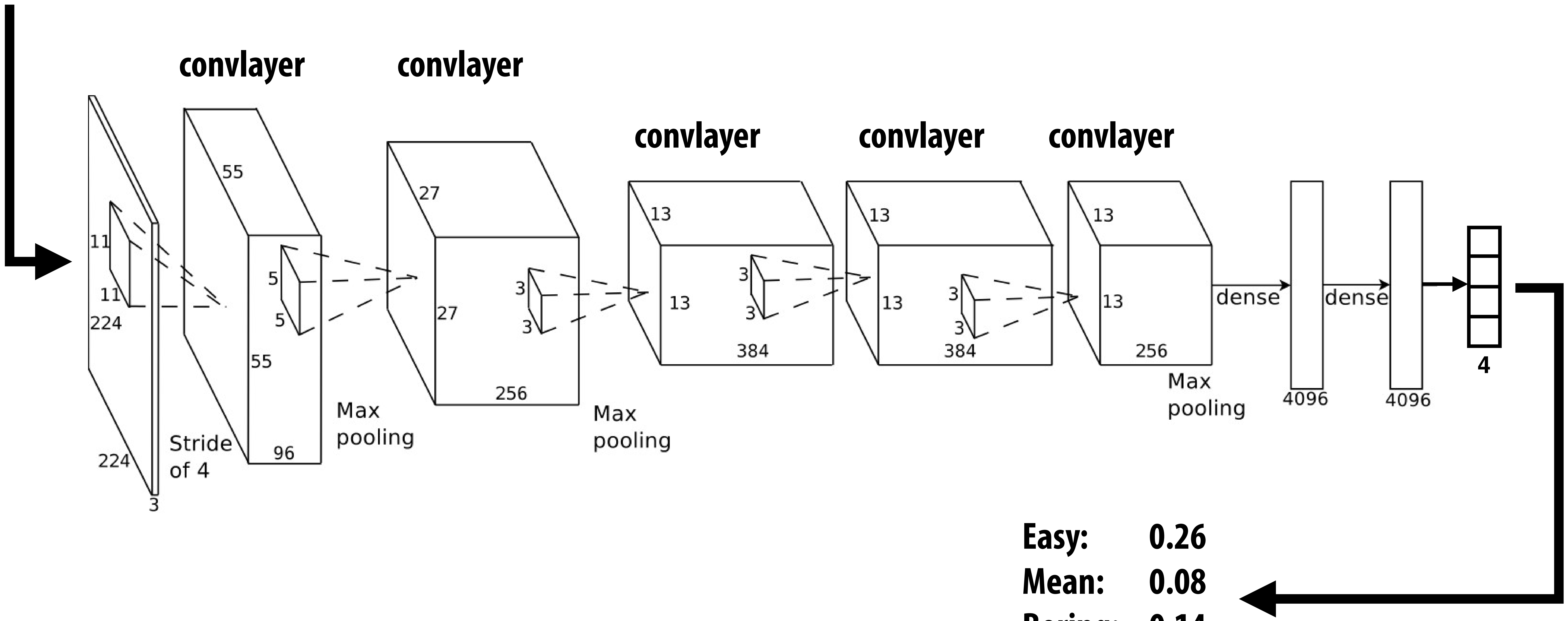
Easy:	**0.26**
Mean:	**0.08**
Boring:	**0.14**
Nerdy:	**0.52**

Network output

Professor classification network

Ground truth
(what the answer should be)

Easy:	0.0
Mean:	0.0
Boring:	0.0
Nerdy:	1.0



Easy:	0.26
Mean:	0.08
Boring:	0.14
Nerdy:	0.52

Network output

Error (loss)

Ground truth:
(what the answer should be)

Easy:	0.0
Mean:	0.0
Boring:	0.0
Nerdy:	1.0

Network output: *

Easy:	0.26
Mean:	0.08
Boring:	0.14
Nerdy:	0.52

Common example: softmax loss:

Output of network for correct category

$$L = -log\left(\frac{e^{f_c}}{\sum_j e^{f_j}}\right)$$

Output of network for all categories

* In practice a network using a softmax classifier outputs unnormalized, log probabilities (f_j), but I'm showing a probability distribution above for clarity

DNN training

Goal of training: learning good values of network parameters so that the network outputs the correct classification result for any input image

Idea: minimize loss for all the training examples (for which the correct answer is known)

$$L = \sum_i L_i$$

(total loss for entire training set is sum of losses *L_i* for each training example *x_i*)

Intuition: if the network gets the answer correct for a wide range of training examples, then hopefully it has learned parameter values that yield the correct answer for future images as well.

Gradient descent

An idea as old as the hills:

Intuition: gradient descent

Say you had a function *f* that contained hidden parameters *p₁ and p₂*: $f(x_i)$

And for some input *xᵢ*, your training data says the function should output 0.

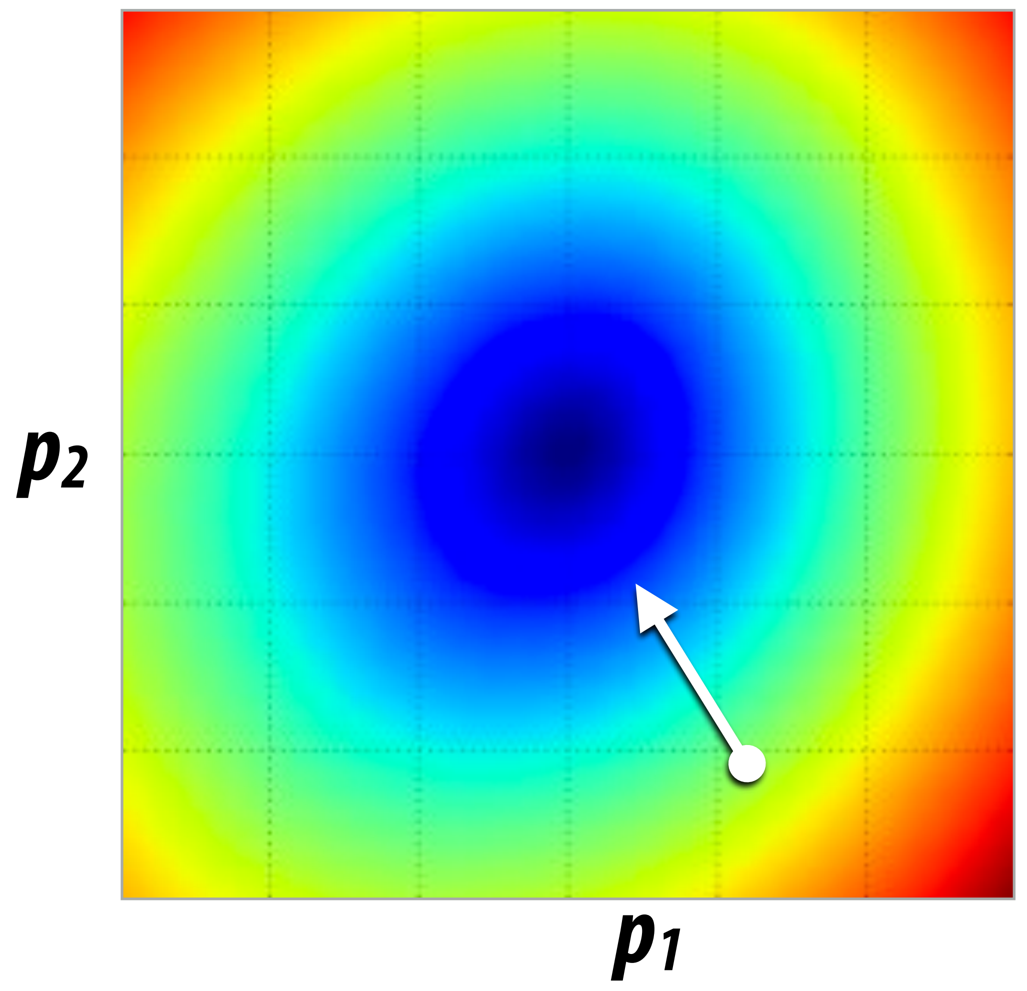
But for the current values of *p₁ and p₂*, it currently outputs 10.

$$f(x_i, p_1, p_2) = 10$$

And say I also gave you expressions for the derivative of *f* with respect to *p₁ and p₂* so you could compute their value at *xᵢ*.

$$\frac{df}{dp_1} = 2 \quad \frac{df}{dp_2} = -5 \qquad \nabla f = [2, -5]$$

red = high values of f
blue = low values



How might you adjust the values *p₁ and p₂* to reduce the error for this training example?

Basic gradient descent

```
while (loss too high):
  for each epoch: // a pass through the training dataset
     for each item x_i in training set:
        grad = evaluate_loss_gradient(f, params, loss_func, x_i)
        params += -grad * learning_rate;
```

Mini-batch stochastic gradient descent (mini-batch SGD):
choose a random (small) subset of the training examples to use to compute the gradient in each iteration of the while loop

```
while (loss too high):
  for each epoch: // a pass through the training dataset
     for all mini batches in training set:
       grad = 0;
       for each item x_i in minibatch:
          grad += evaluate_loss_gradient(f, params, loss_func, x_i)
       params += -grad * learning_rate;
```

How do we compute dLoss/dp for a deep neural network with millions of parameters?

SGD workload

```
while (loss too high):

    for each item x_i in mini-batch:
       grad += evaluate_loss_gradient(f, loss_func, params, x_i)

    params += -grad * step_size;
```

At first glance, this loop is sequential (each step of "walking downhill" depends on previous)

Parallel across images

sum reduction

large computation with its own parallelism (but working set may not fit on single machine)

trivial data-parallel over parameters

DNN training workload

- **Large computational expense**
  - Must evaluate the network (forward and backward) for millions of training images
  - Must iterate for many iterations of gradient descent (100's of thousands)
  - Training modern networks on big datasets takes days

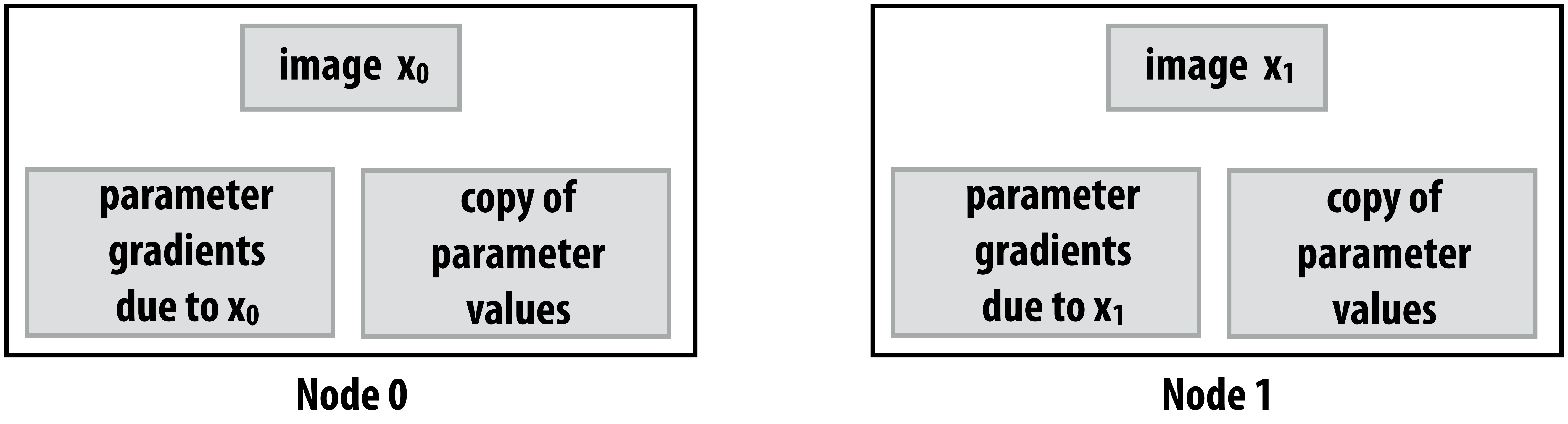
- **Large memory footprint**
  - Must maintain network layer outputs from forward pass
  - Additional memory to store gradients/gradient velocity for each parameter
  - Scaling to larger networks requires partitioning DNN across nodes to keep DNN + intermediates in memory

- **Dependencies /synchronization (not embarrassingly parallel)**
  - Each parameter update step depends on previous
  - Many units contribute to same parameter gradients (fine-scale reduction)
  - Different images in mini batch contribute to same parameter gradients

Synchronous data-parallel training (across images)

```
for each item x_i in mini-batch:
   grad += evaluate_loss_gradient(f, loss_func, params, x_i)
params += -grad * learning_rate;
```

Consider parallelization of the outer for loop across machines in a cluster



```
partition dataset across nodes
for each item x_i in mini-batch assigned to local node:
   // just like single node training
   grad += evaluate_loss_gradient(f, loss_func, params, x_i)
barrier();
sum reduce gradients, communicate results to all nodes
barrier();
update copy of parameter values
```

Synchronous training

- All nodes cooperate to compute gradients for a mini-batch *
- Gradients are summed (across the entire machine)
  - All-to-all communication
  - Good implementations will sum gradients for layer *i* when computing backprop for *i*+1 (overlap communication and computation).
- Update model parameters
  - Typically done without wide parallelism (e.g. each machine computes its own update)
- All nodes proceed to work on next mini-batch given new model parameters

* If curious about batch norm in a parallel training setting. In practice each of *k* nodes works on a set of *n* images, with batch norm statistics computed independently for each set of n (mini-batch size is *kn*).

Challenges of scaling out (many nodes)

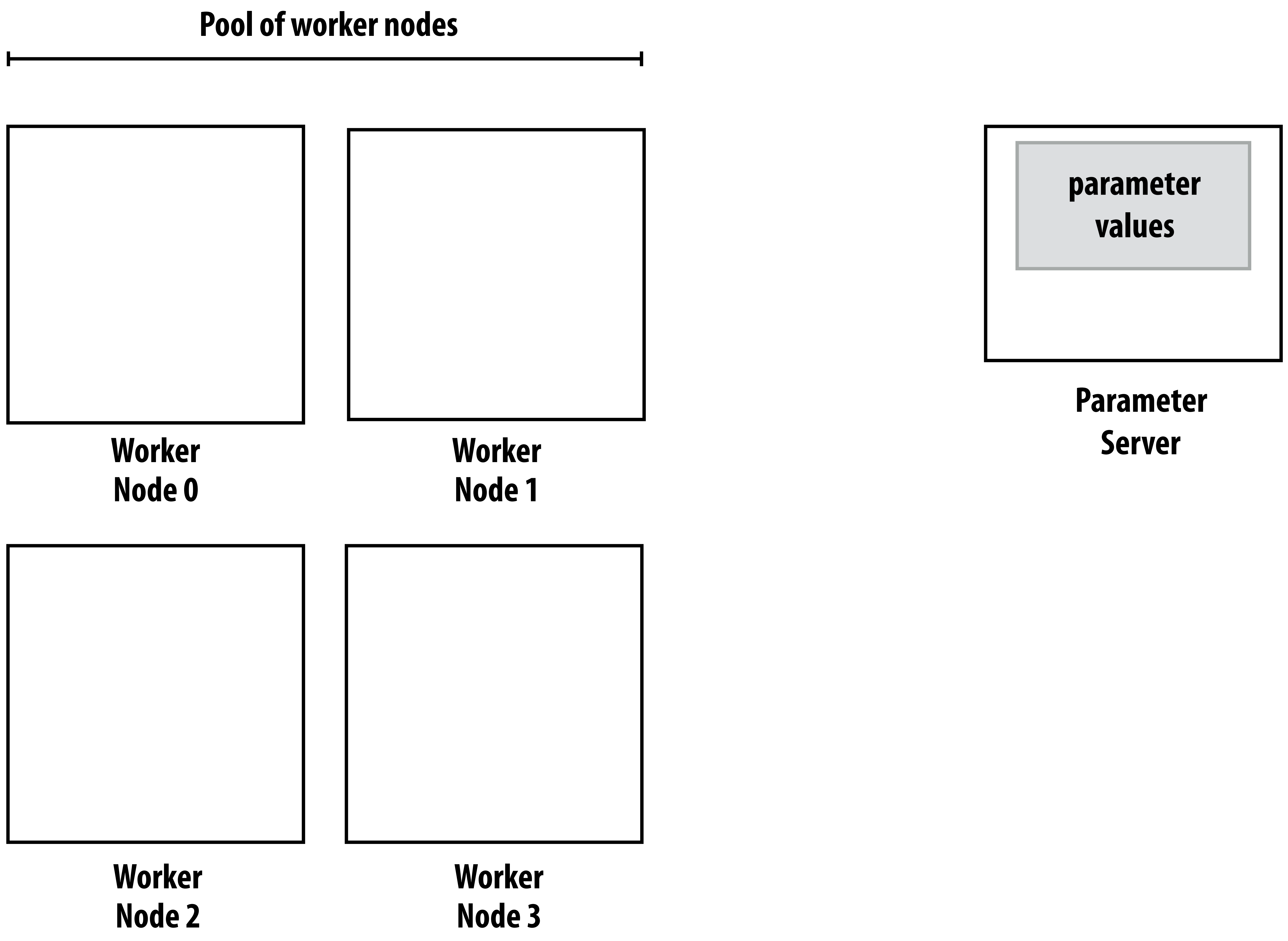
- **Slow communication between nodes**
  - Commodity clusters do not feature high-performance interconnects (e.g., infiniband) typical of supercomputers
  - Synchronous SGD involves all to all communication after each minibatch

- **Nodes with different performance (even if machines are the same)**
  - Workload imbalance at barriers (sync points between nodes)

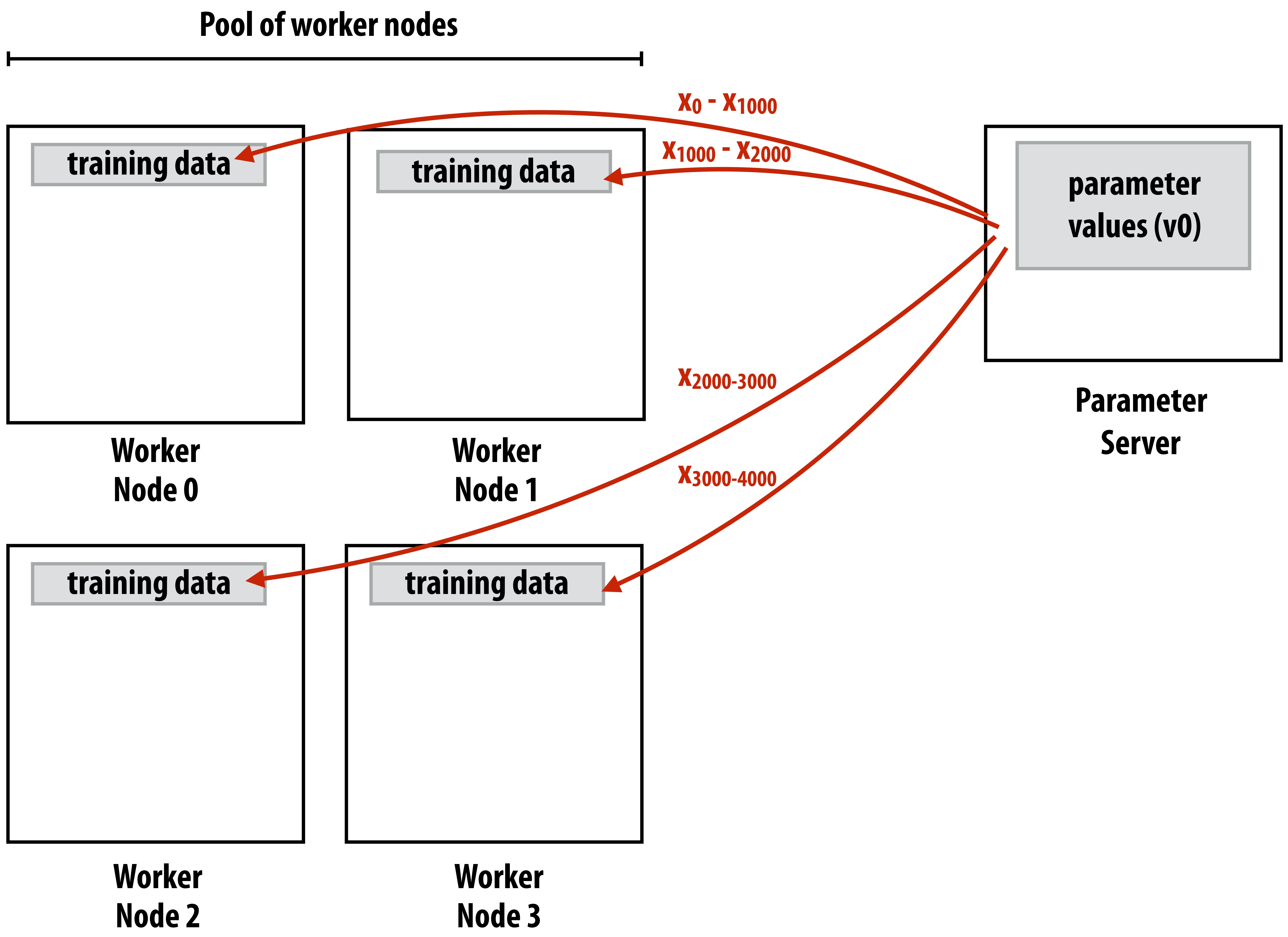
Alternative solution: exploit properties of SGD by using asynchronous execution

Parameter server design

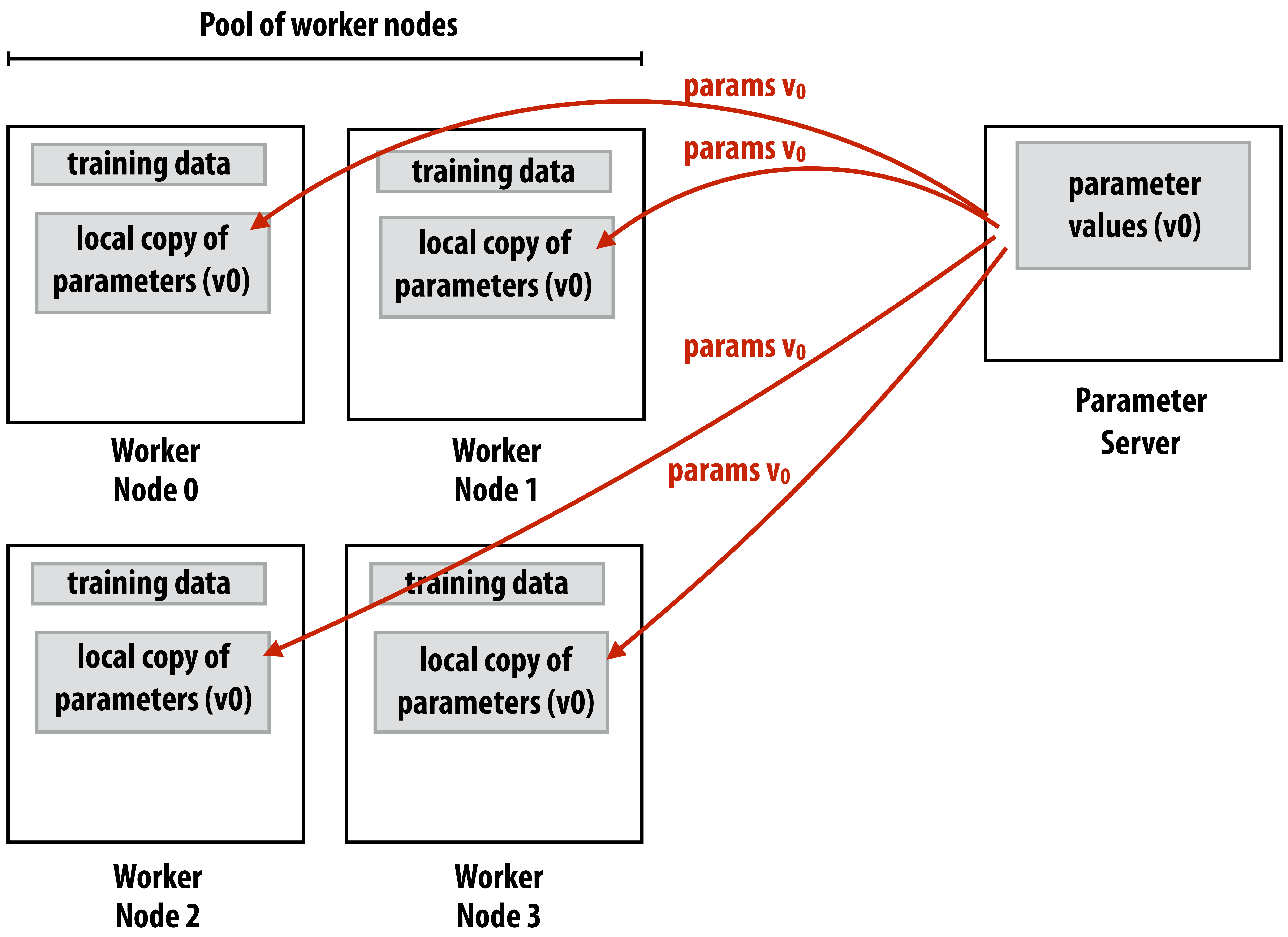
Google's DistBelief [Dean NIPS12]
Parameter Server [Li OSDI14]
Microsoft's Project Adam [Chilimbi OSDI14]



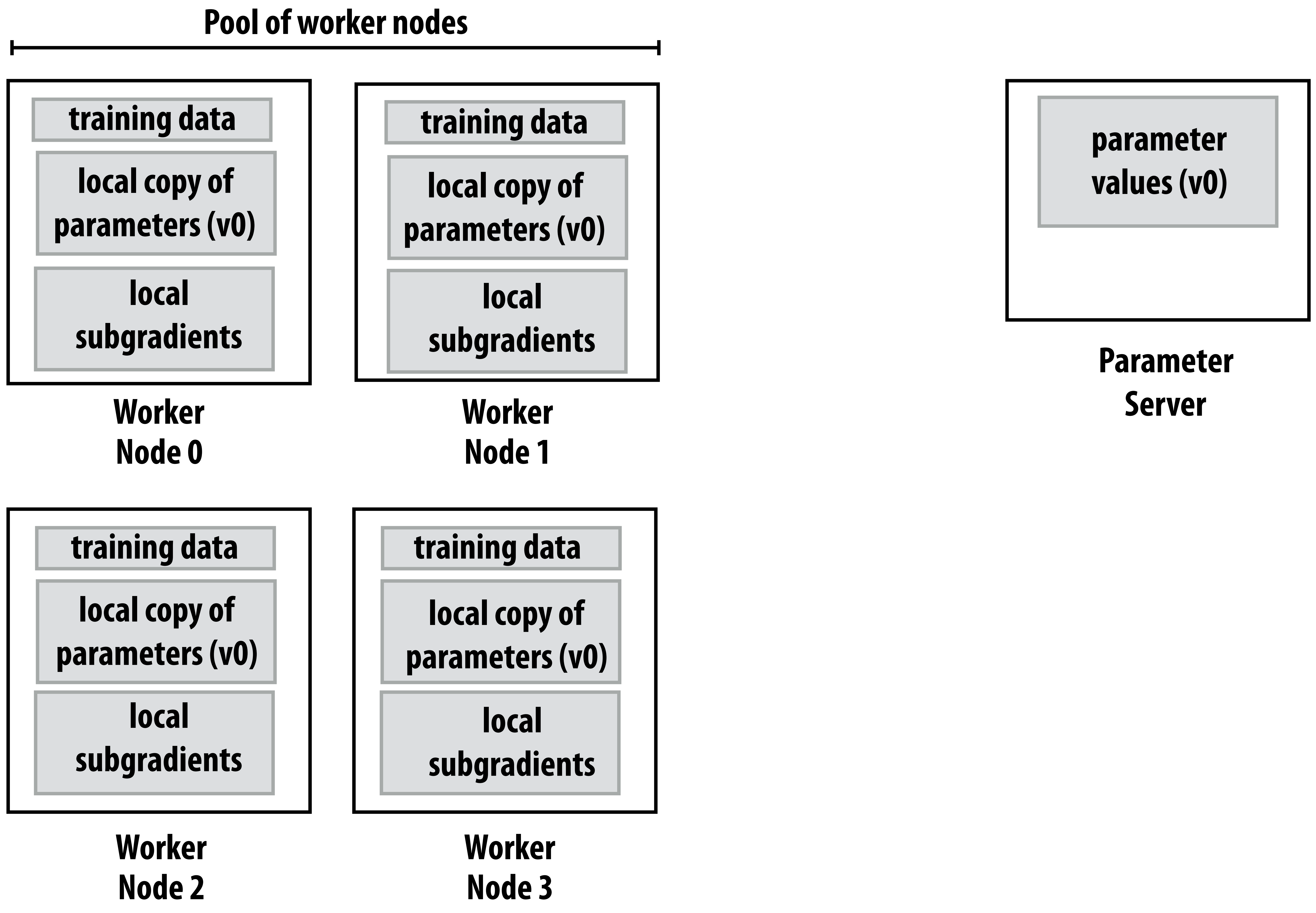
Training data partitioned among workers



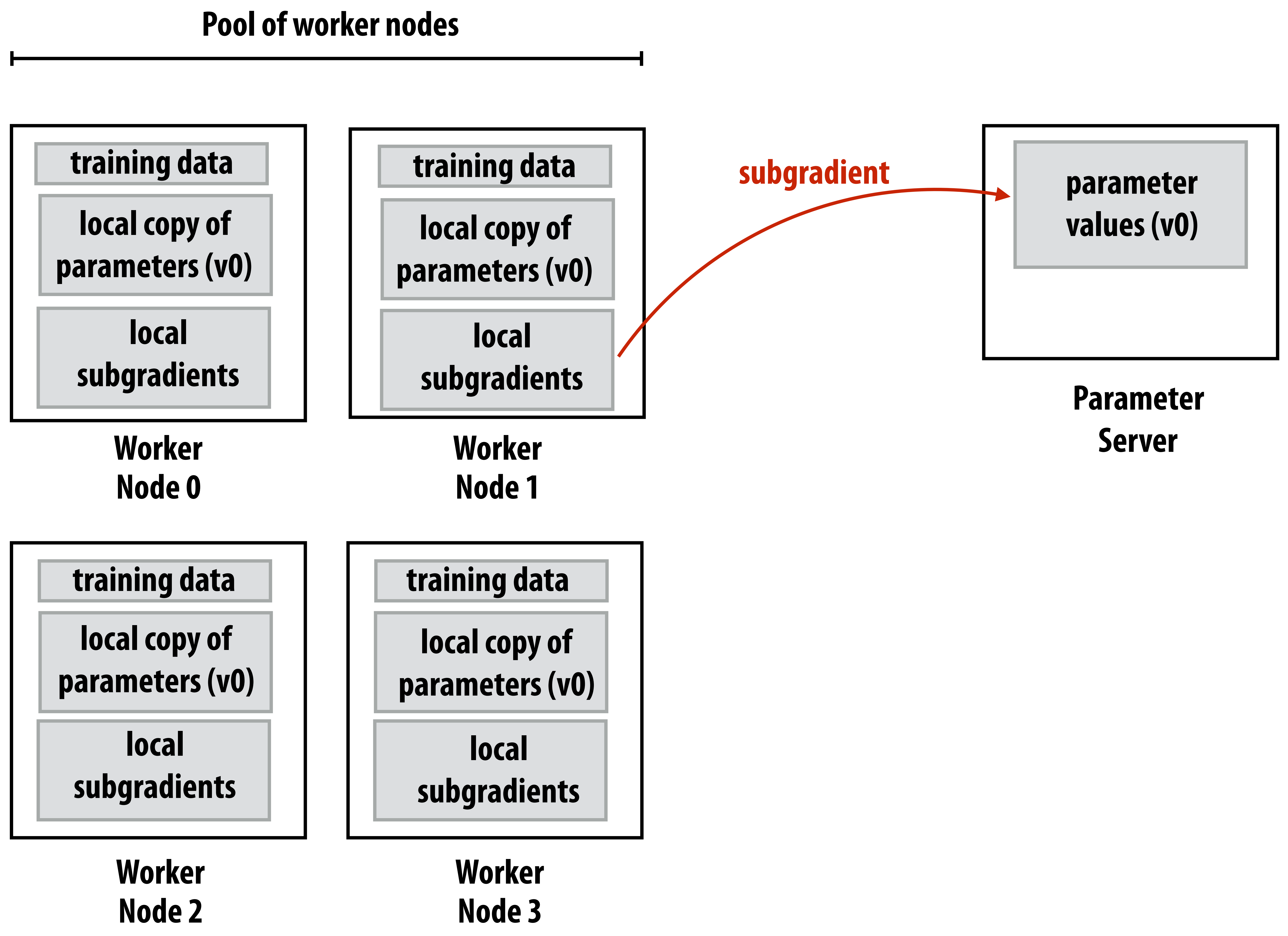
Copy of parameters sent to workers



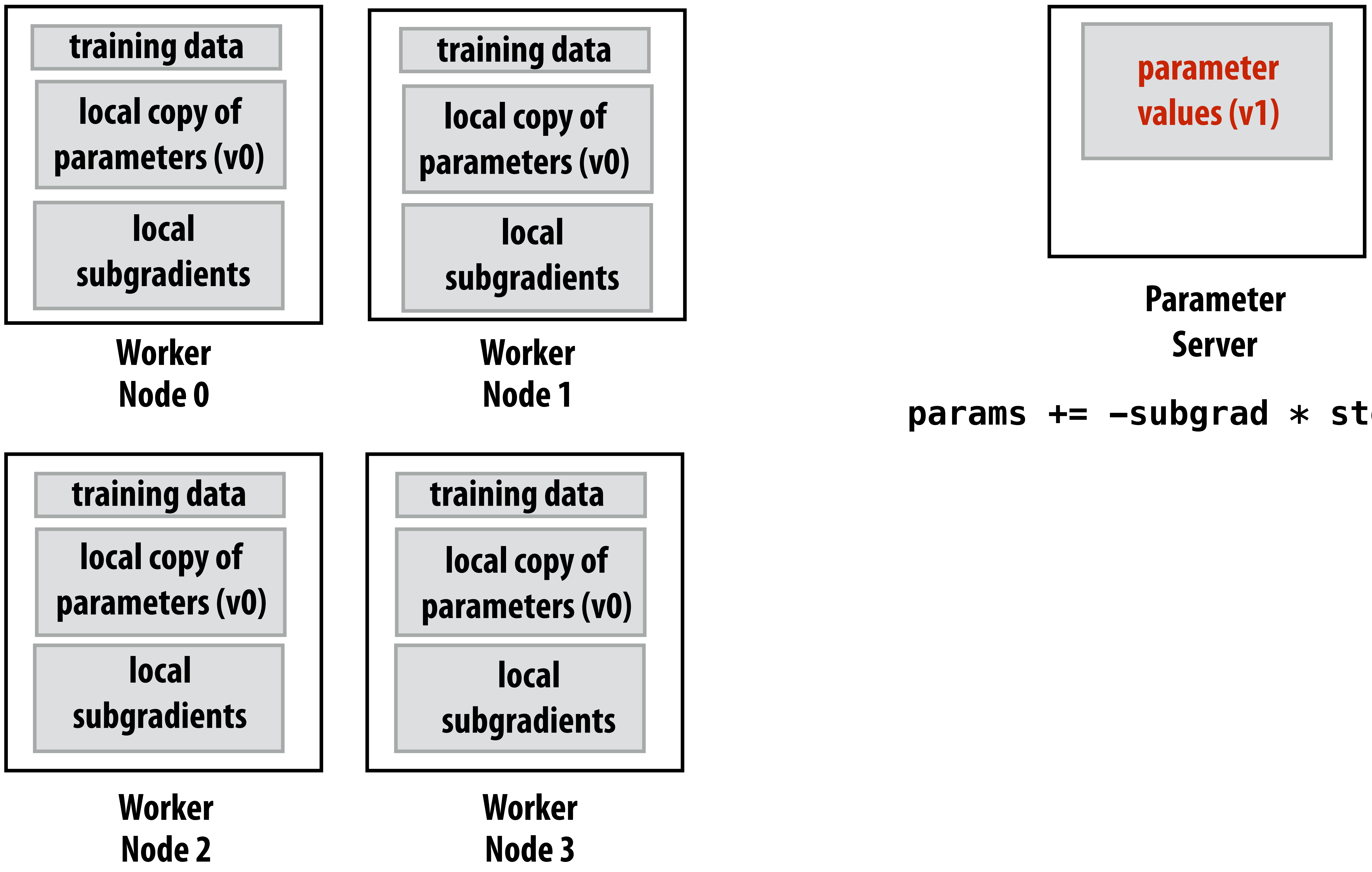
Data parallelism: workers independently compute local "subgradients" on different pieces of data



Worker sends subgradient to parameter server



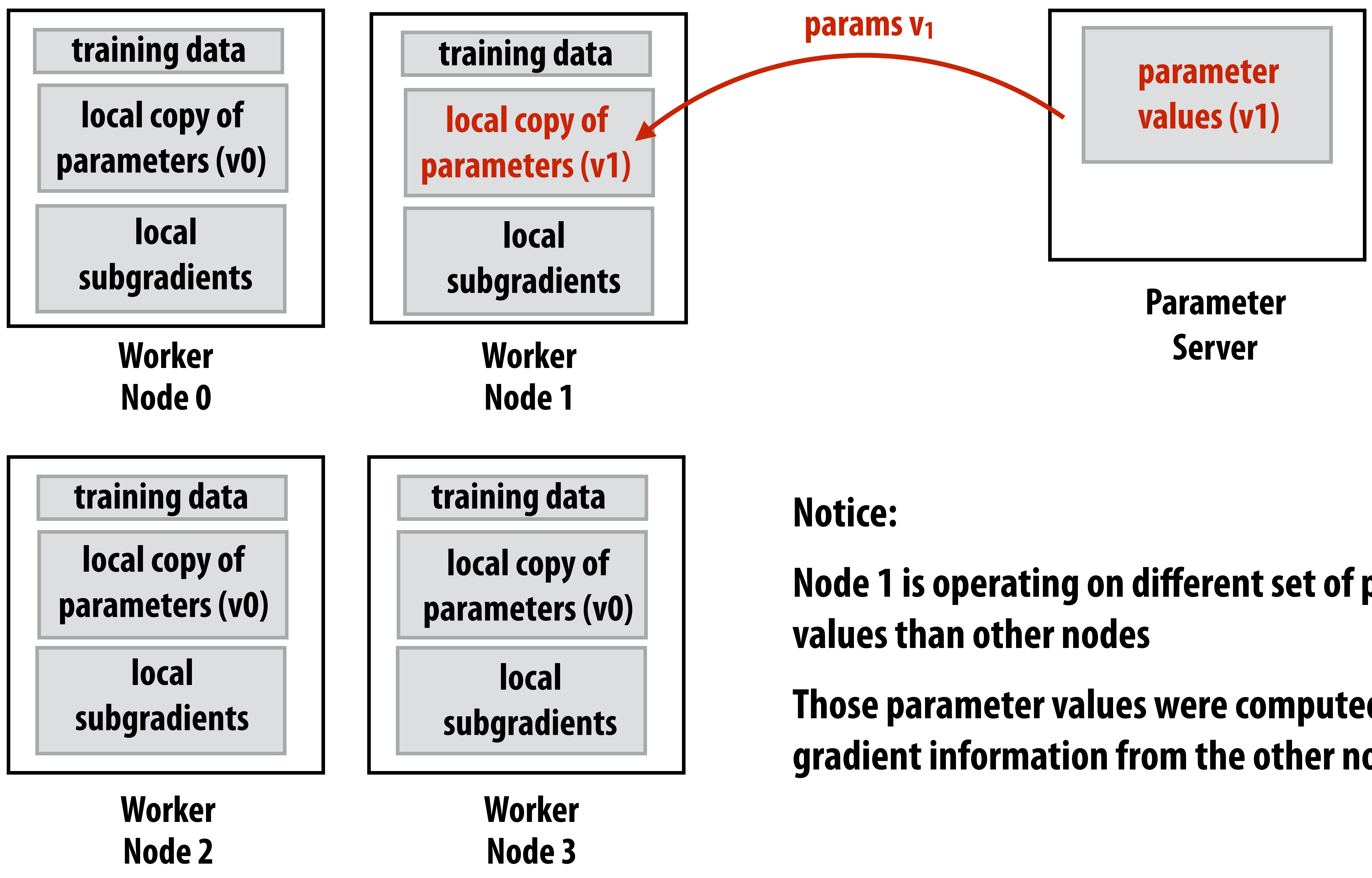
Server updates global parameter values based on subgradient



```
params += -subgrad * step_size;
```

Updated parameters sent to worker

Then worker proceeds with another gradient computation step

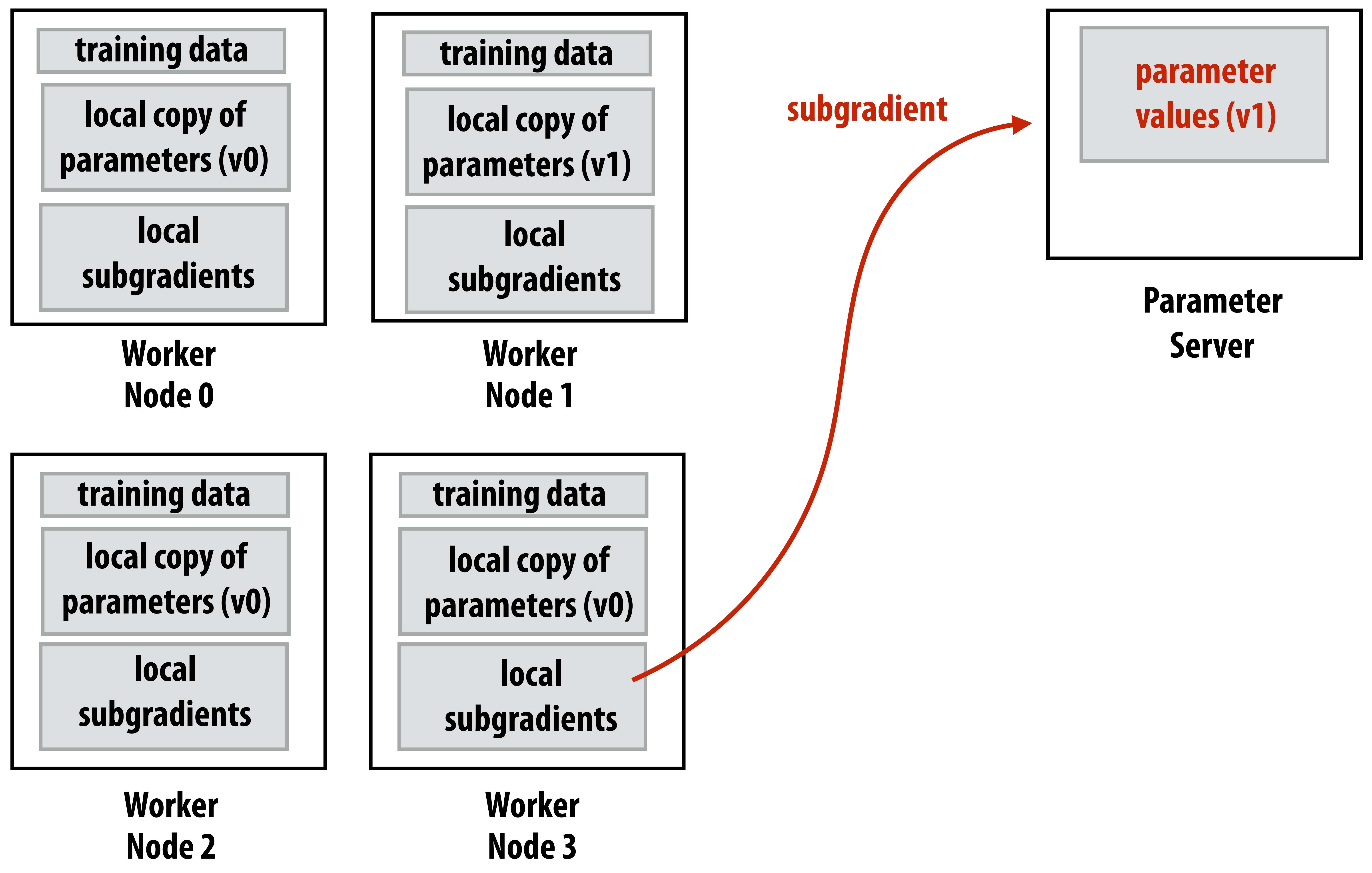


Notice:

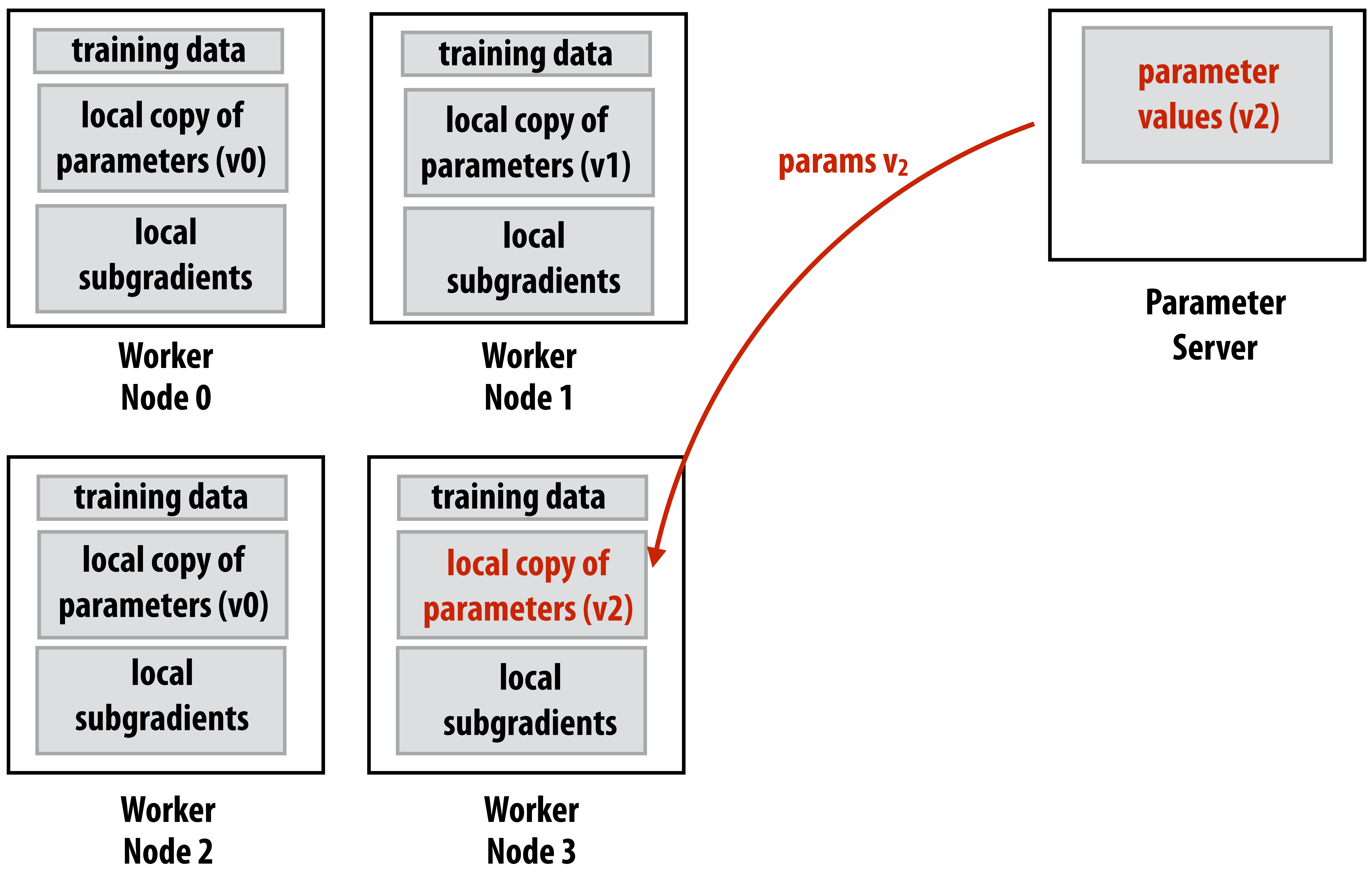
Node 1 is operating on different set of parameter values than other nodes

Those parameter values were computed without gradient information from the other nodes

Updated parameters sent to worker (again)



Worker continues with updated parameters

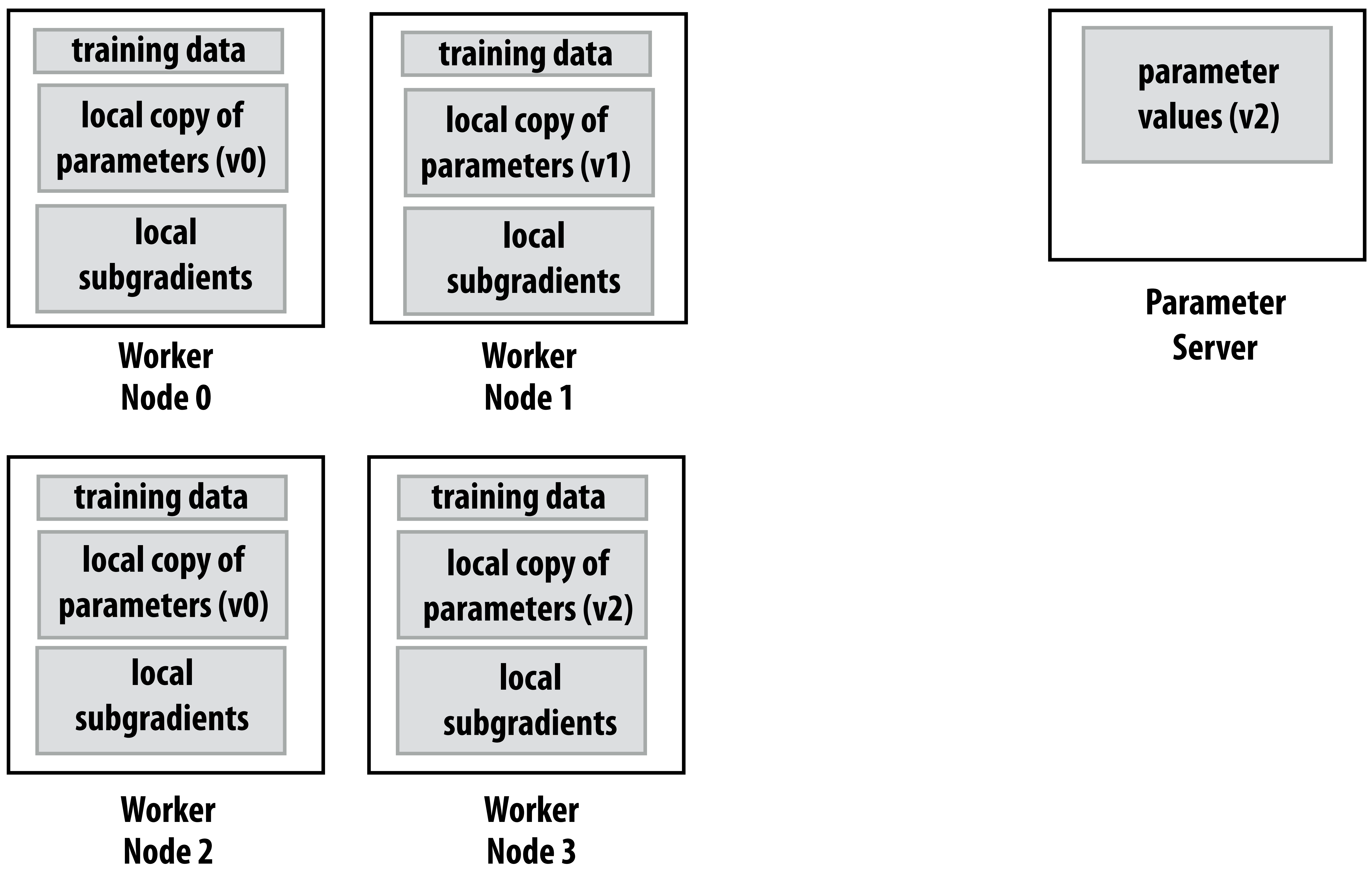


Summary: asynchronous parameter update

- **Idea: avoid global synchronization on all parameter updates between each SGD iteration**
  - Algorithm design reflects realities of cluster computing:
    - Slow interconnects
    - Unpredictable machine performance
- **Solution: asynchronous (and partial) subgradient updates**
- **Will impact convergence of SGD**
  - Node N working on iteration *i* may not have parameter values that result the results of the *i-1* prior SGD iterations

Bottleneck?

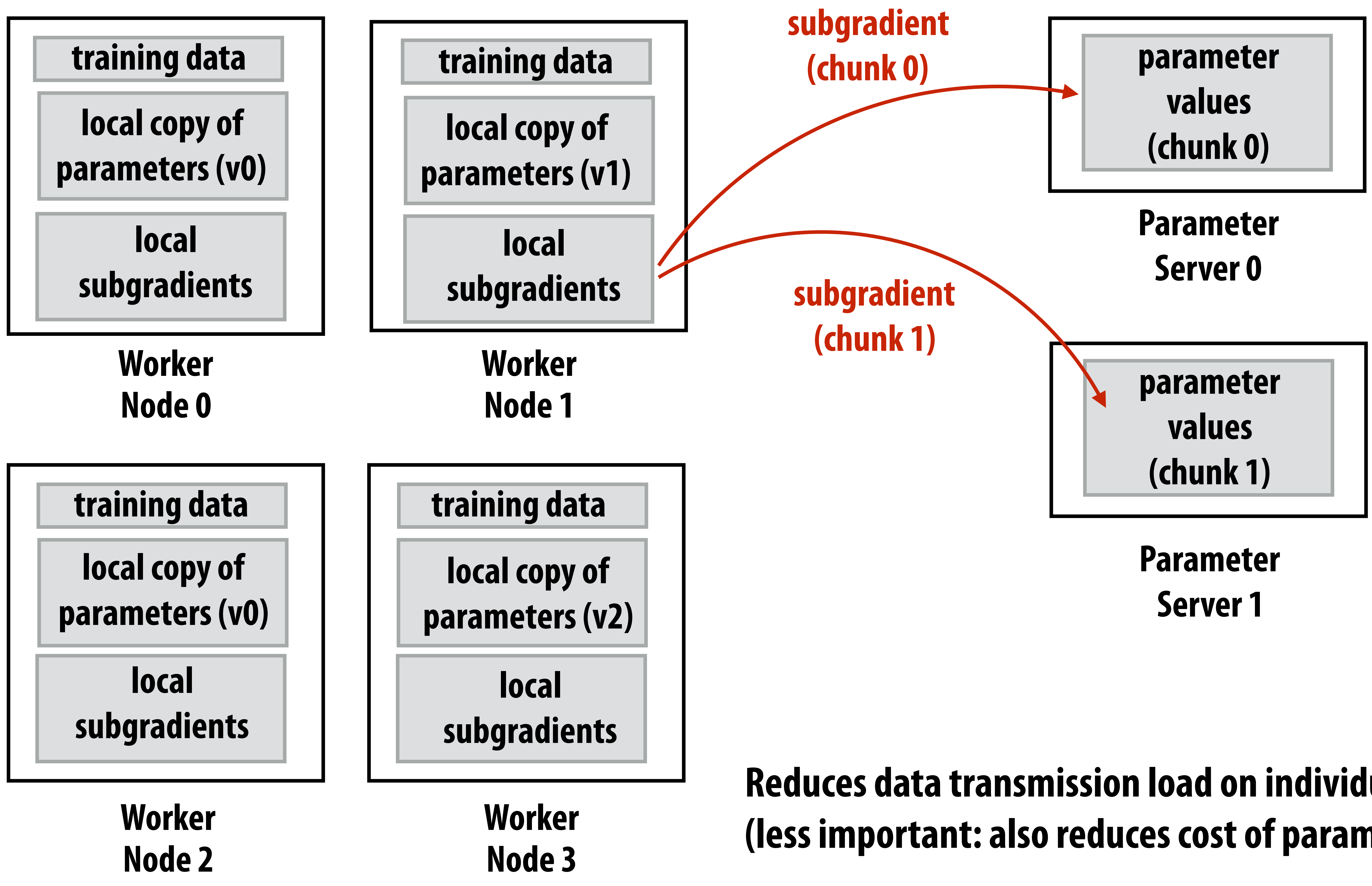
What if there is heavy contention for parameter server?



Shard the parameter server

Partition parameters across servers

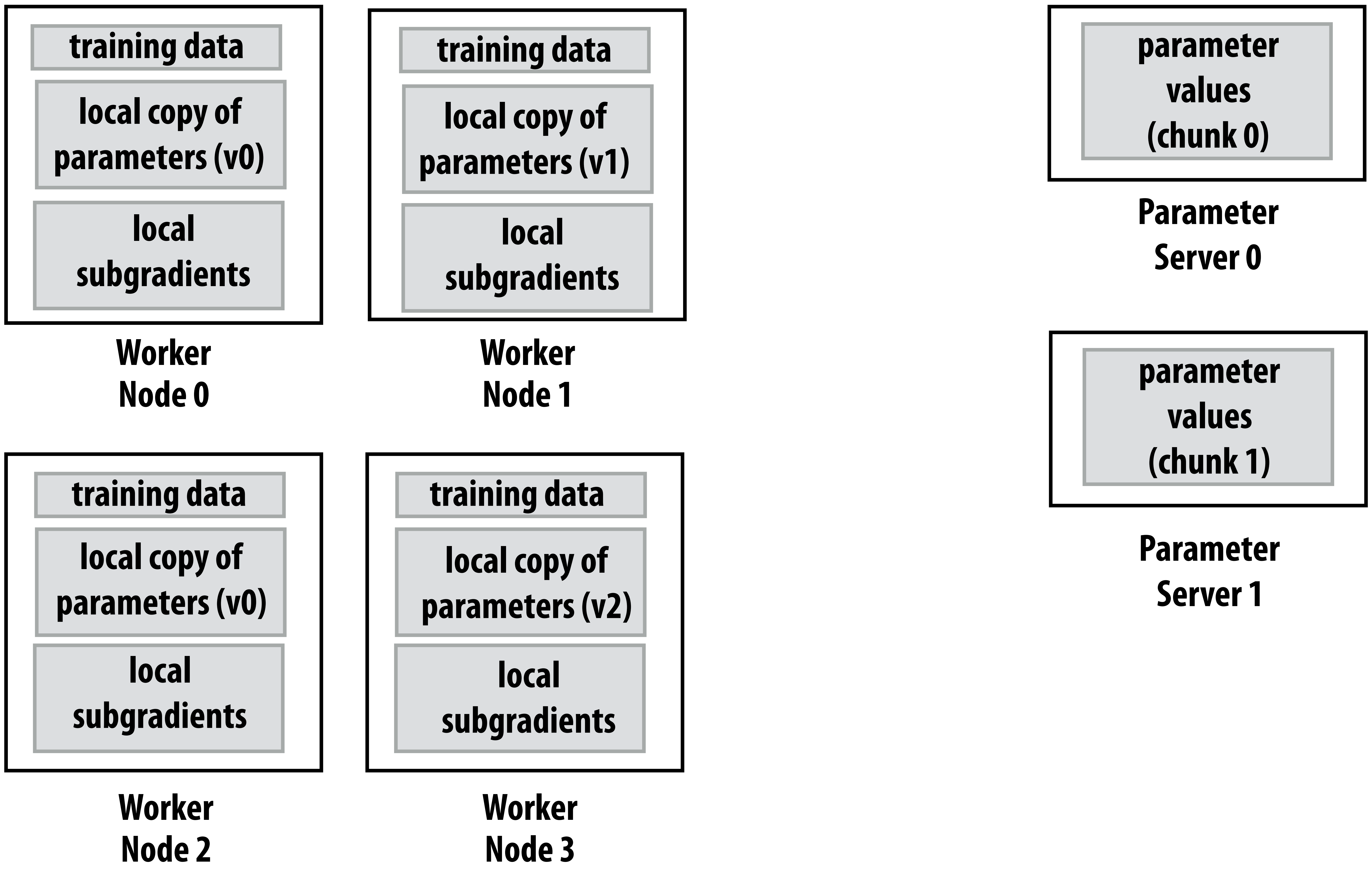
Worker sends chunk of subgradients to owning parameter server



Reduces data transmission load on individual servers
(less important: also reduces cost of parameter update)

What if model parameters do not fit on one worker?

Recall high footprint of training large networks (particularly with large mini-batch sizes)

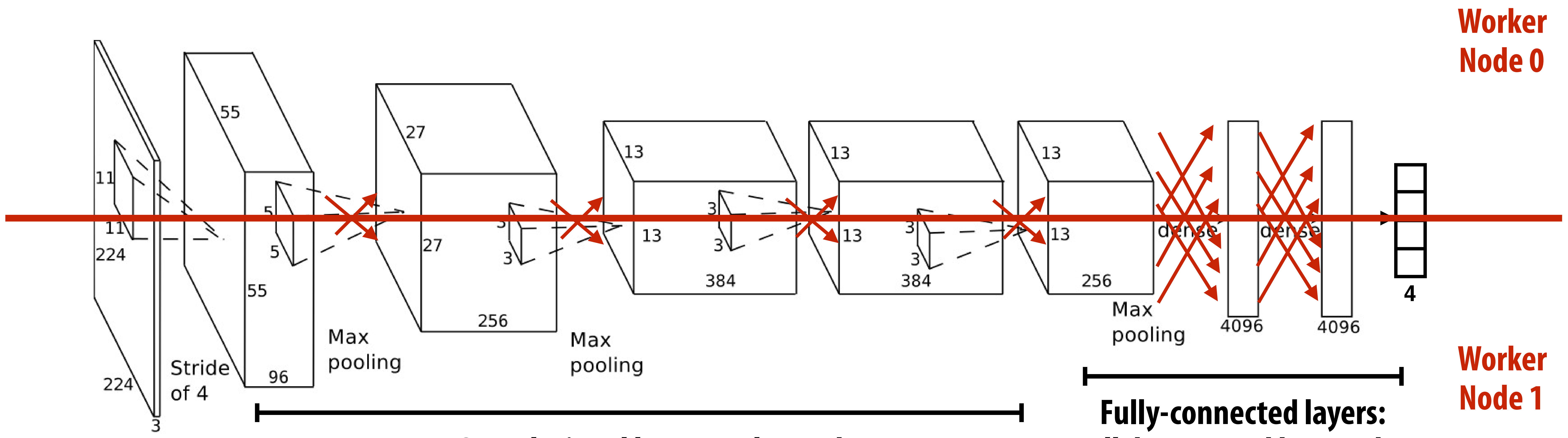


Model parallelism

Partition network parameters across nodes (spatial partitioning to reduce communication)

Reduce internode communication through network design:

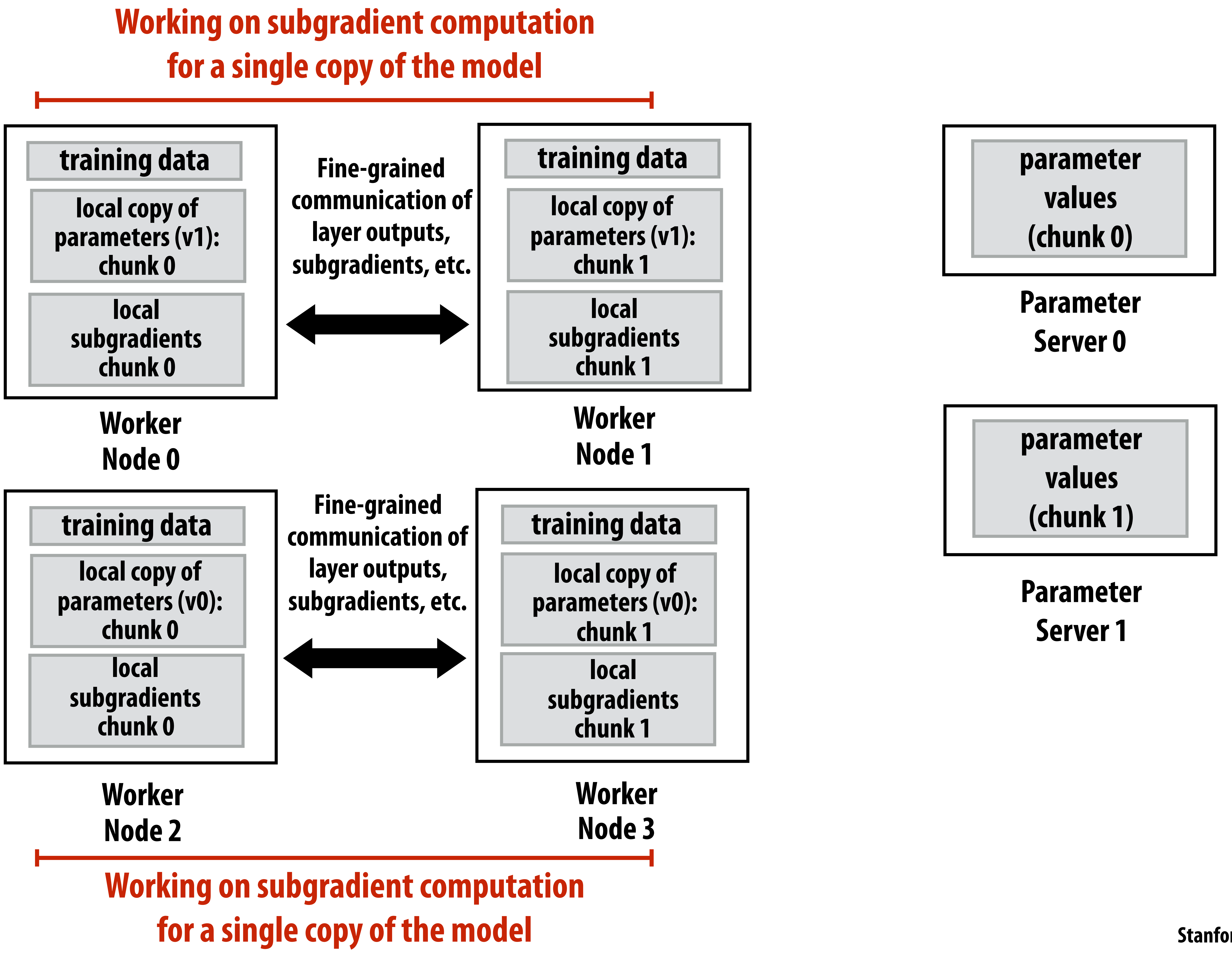
- Use small spatial convolutions (1x1 convolutions)
- Reduce/shrink fully-connected layers



Convolutional layers: only need to communicate outputs near spatial partition

Fully-connected layers: all data owned by a node must by communicated to other nodes

Data-parallel and model-parallel execution


Working on subgradient computation for a single copy of the model
training data
local copy of parameters (v1): chunk 0
local subgradients chunk 0
Worker Node 0
Fine-grained communication of layer outputs, subgradients, etc.
training data
local copy of parameters (v1): chunk 1
local subgradients chunk 1
Worker Node 1
parameter values (chunk 0)
Parameter Server 0
parameter values (chunk 1)
Parameter Server 1
training data
local copy of parameters (v0): chunk 0
local subgradients chunk 0
Worker Node 2
Fine-grained communication of layer outputs, subgradients, etc.
training data
local copy of parameters (v0): chunk 1
local subgradients chunk 1
Worker Node 3
Working on subgradient computation for a single copy of the model

Asynchronous vs. synchronous debate

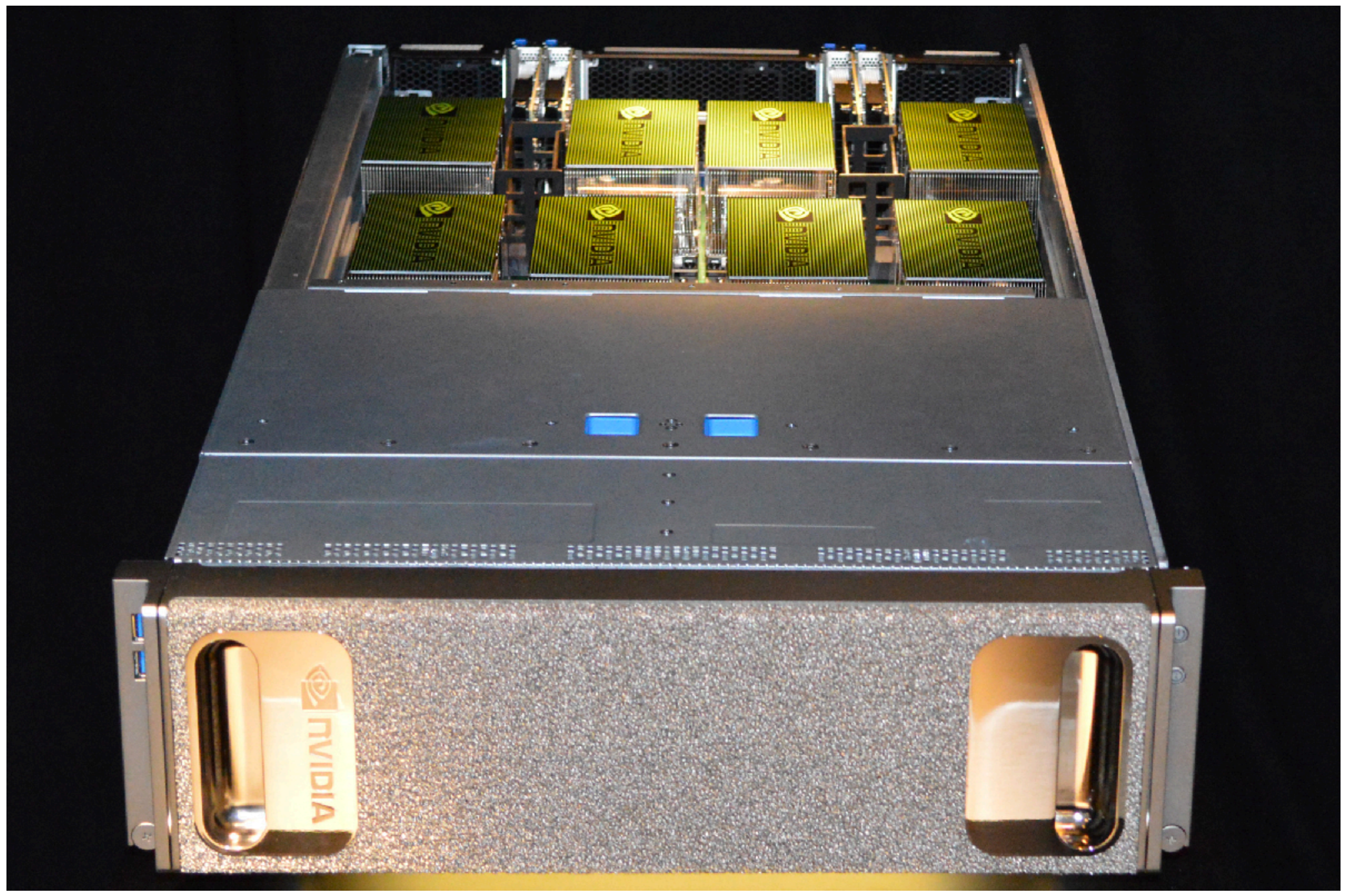
- **Asynchronous training: significant distributed system complexity incurred to combat bandwidth/latency constraints of modern cluster computing**

- **Interest in ways to improve scalability of synchronous training**
  - **Better hardware**
  - **Better algorithms for existing hardware**

Better hardware: using supercomputers for training

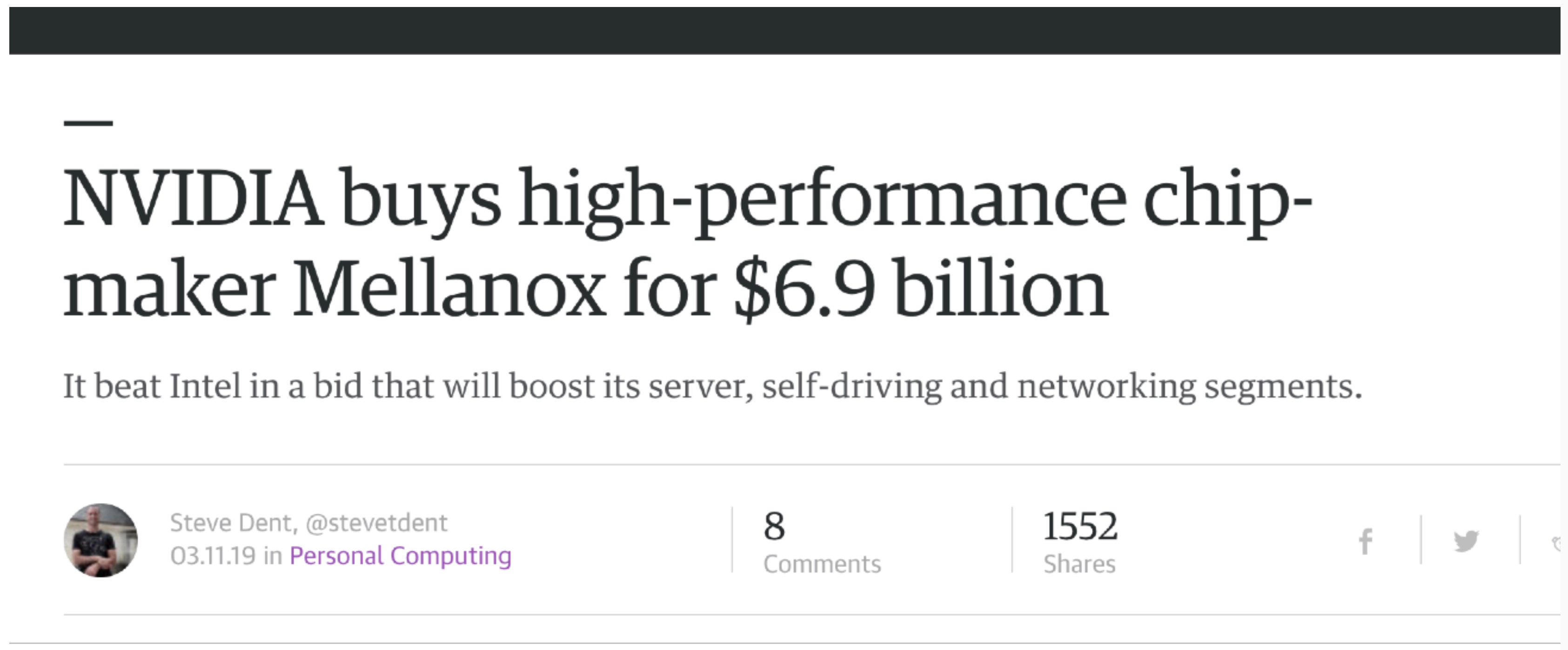
- Fast interconnects critical for model-parallel training
  - Fine-grained communication of outputs and gradients
- Fast interconnects diminish need for async training algorithms
  - Avoid randomness in training due to schedule of computation (there remains randomness due to stochastic part of SGD algorithm)

OakRidge Titan Supercomputer (low-latency interconnect used in a number of recent training papers)



NVIDIA DGX-1: 8 GPUs connected via high speed NV-Link interconnect ($150,000 in 2018)

Just the other day…



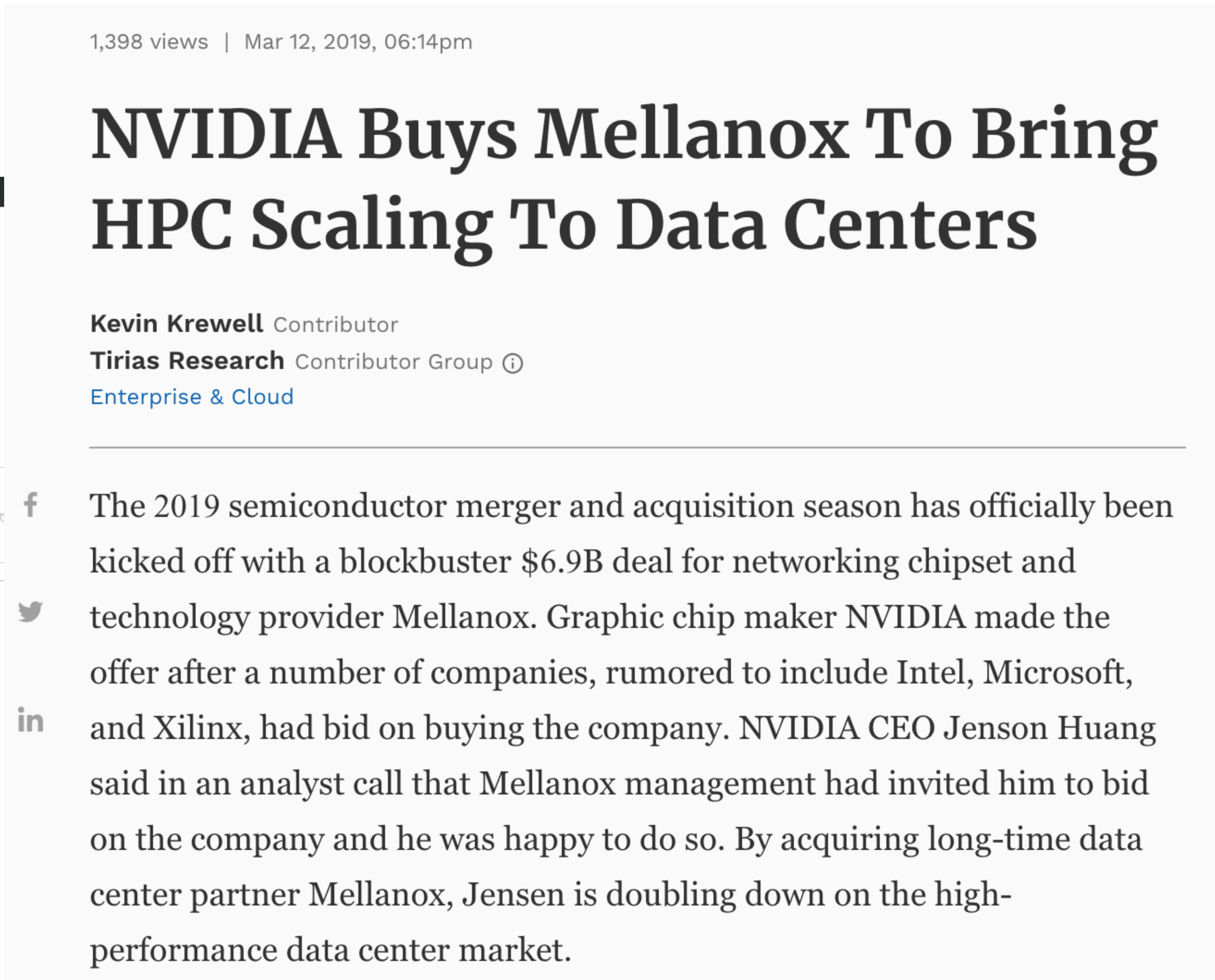
NVIDIA buys high-performance chip-maker Mellanox for $6.9 billion

It beat Intel in a bid that will boost its server, self-driving and networking segments.

Steve Dent, @stevetdent
03.11.19 in Personal Computing

8
Comments

1552
Shares

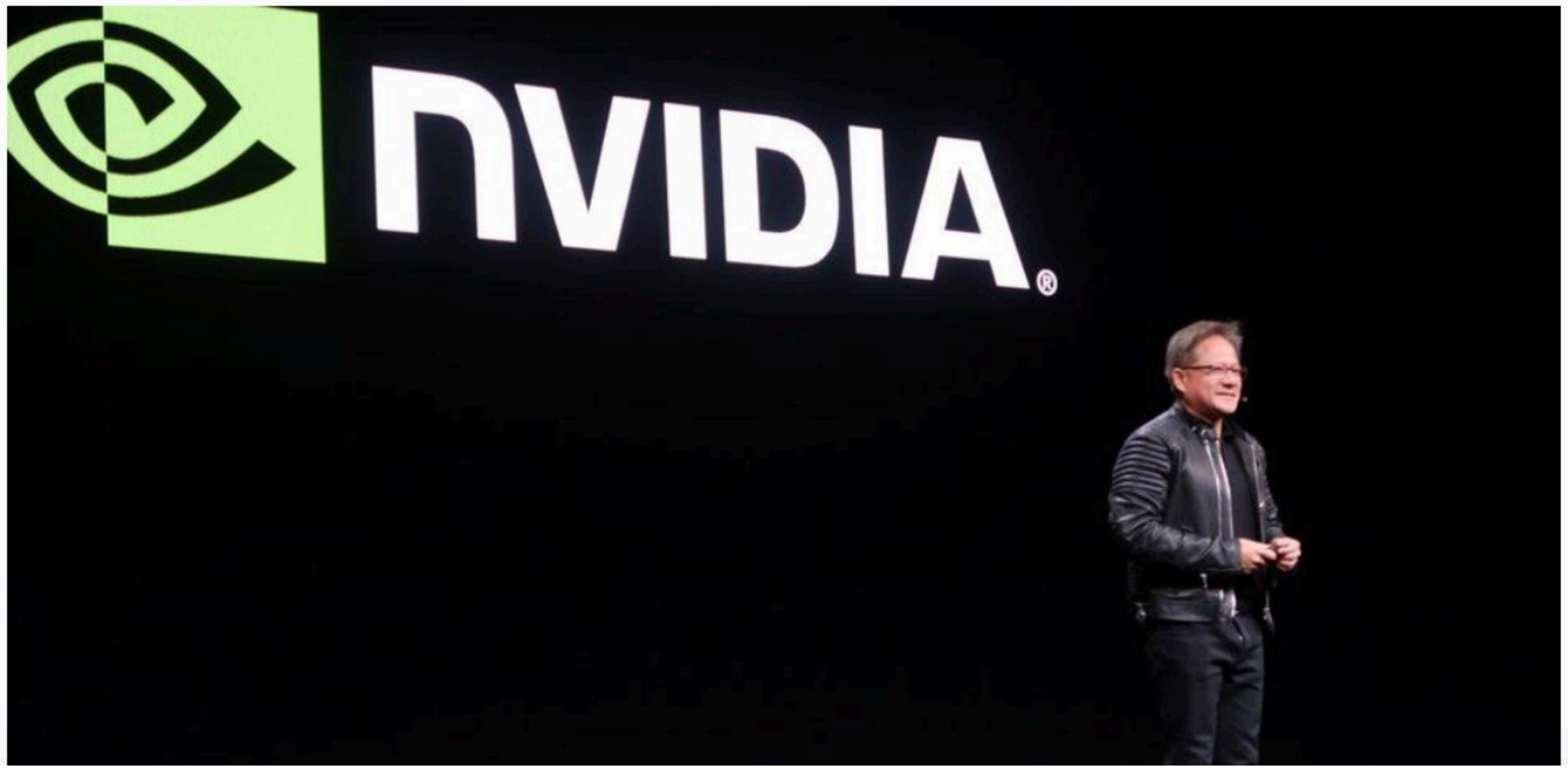


1,398 views | Mar 12, 2019, 06:14pm

NVIDIA Buys Mellanox To Bring HPC Scaling To Data Centers

Kevin Krewell Contributor
Tirias Research Contributor Group
Enterprise & Cloud

The 2019 semiconductor merger and acquisition season has officially been kicked off with a blockbuster $6.9B deal for networking chipset and technology provider Mellanox. Graphic chip maker NVIDIA made the offer after a number of companies, rumored to include Intel, Microsoft, and Xilinx, had bid on buying the company. NVIDIA CEO Jenson Huang said in an analyst call that Mellanox management had invited him to bid on the company and he was happy to do so. By acquiring long-time data center partner Mellanox, Jensen is doubling down on the high-performance data center market.



Modified algorithmic techniques (again): improving scalability of synchronous training…

- **Larger mini-batches increase computation-to-communication ratio: communicate gradients summed over B training inputs**

```
for each item x in mini-batch on this node:
   grad += evaluate_loss_gradient(f, loss_func, params, x)
barrier();
sum-reduce gradients across all nodes, communicate results to all nodes
barrier();
update copy of local parameter values
```

- **But large mini-batches (if used naively) reduce accuracy of model trained**

Accelerating data-parallel training

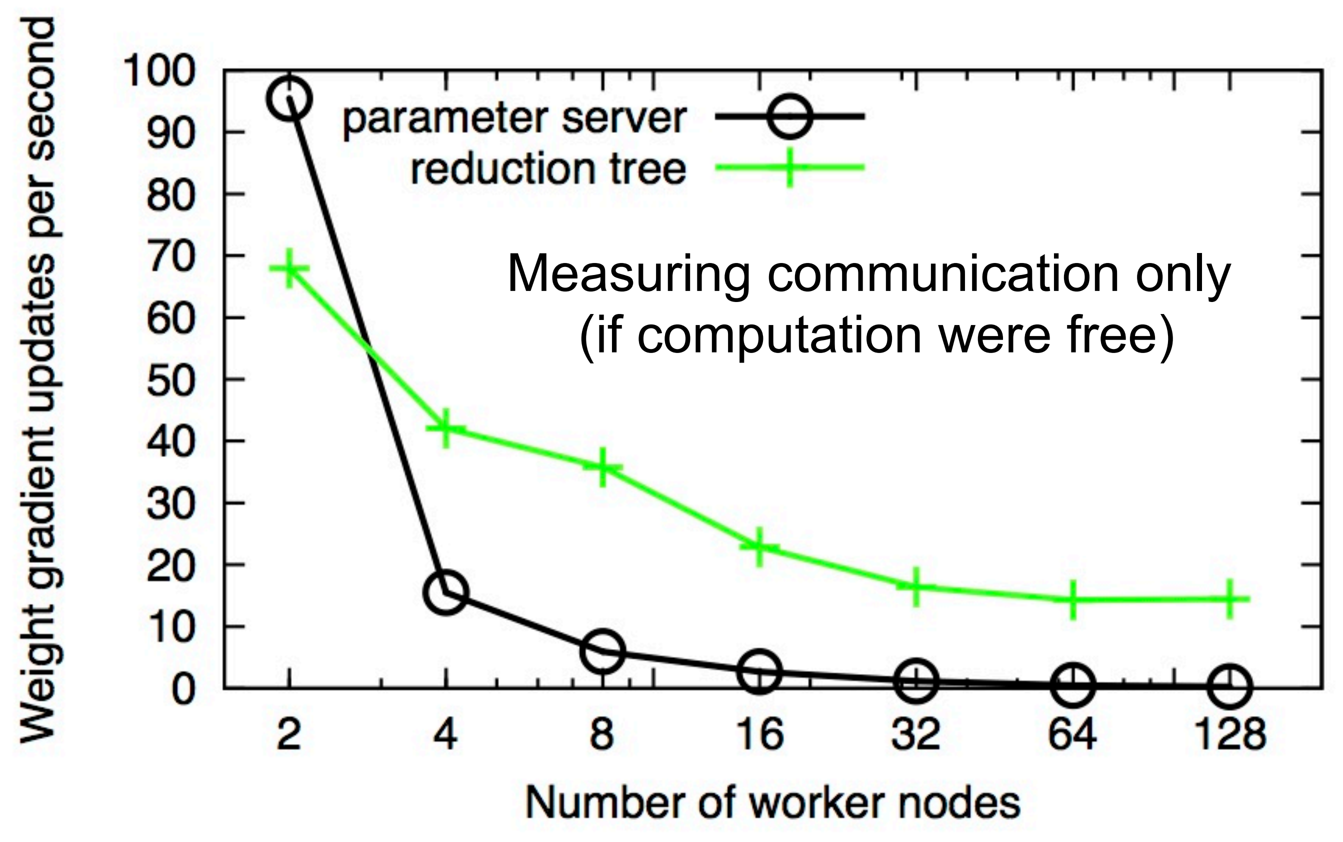
FireCaffe [Iandola 16]

- Use a high-performance Cray Gemini interconnect (Titan supercomputer)
- Use combining tree for accumulating gradients (rather than a single parameter server)
- Use larger batch size (to reduce frequency of communication) and offset by increasing learning rate

	Hardware	Net	Epochs	Batch size	Initial Learning Rate	Train time	Speedup	Top-1 Accuracy	Top-5 Accuracy
Caffe	1 NVIDIA K20	GoogLeNet [41]	64	32	0.01	21 days	1x	68.3%	88.7%
FireCaffe (ours)	32 NVIDIA K20s (Titan supercomputer)	GoogLeNet	72	1024	0.08	23.4 hours	20x	68.3%	88.7%
FireCaffe (ours)	128 NVIDIA K20s (Titan supercomputer)	GoogLeNet	72	1024	0.08	10.5 hours	**47x**	68.3%	88.7%

Dataset: ImageNet 1K

Result: reasonable scalability without asynchronous parameter update: for modern DNNs with fewer weights such as GoogLeNet (due to no fully connected layers)



[Goyal 2017]

Increasing learning rate with mini-batch size: linear scaling rule

size of mini batch = n
SGD learning rate = η

Recall: minibatch SGD parameter update

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

Consider processing of k minibatches (k steps of gradient descent)

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

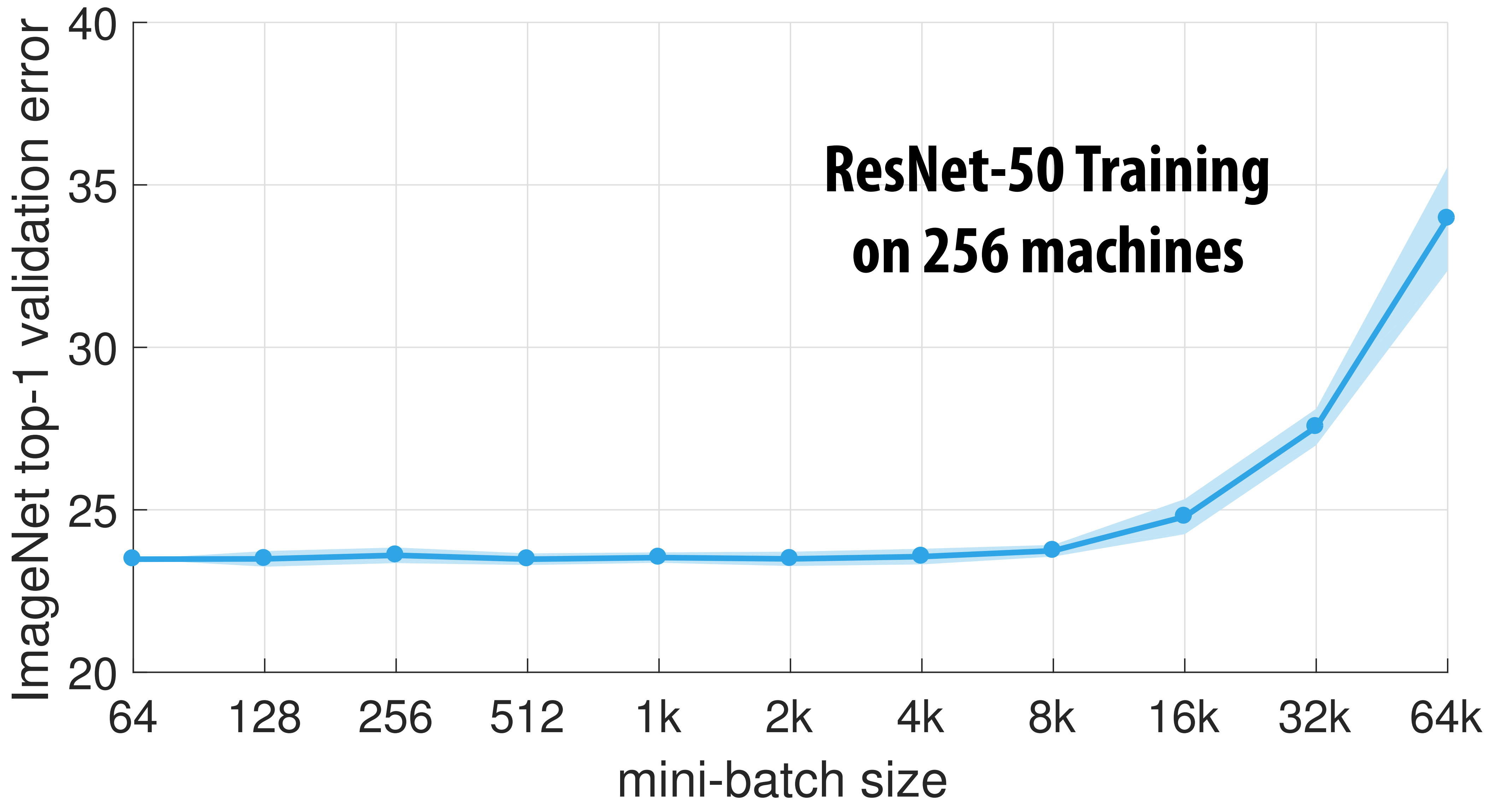
Consider processing one minibatch that is of size kn (one step of gradient descent)

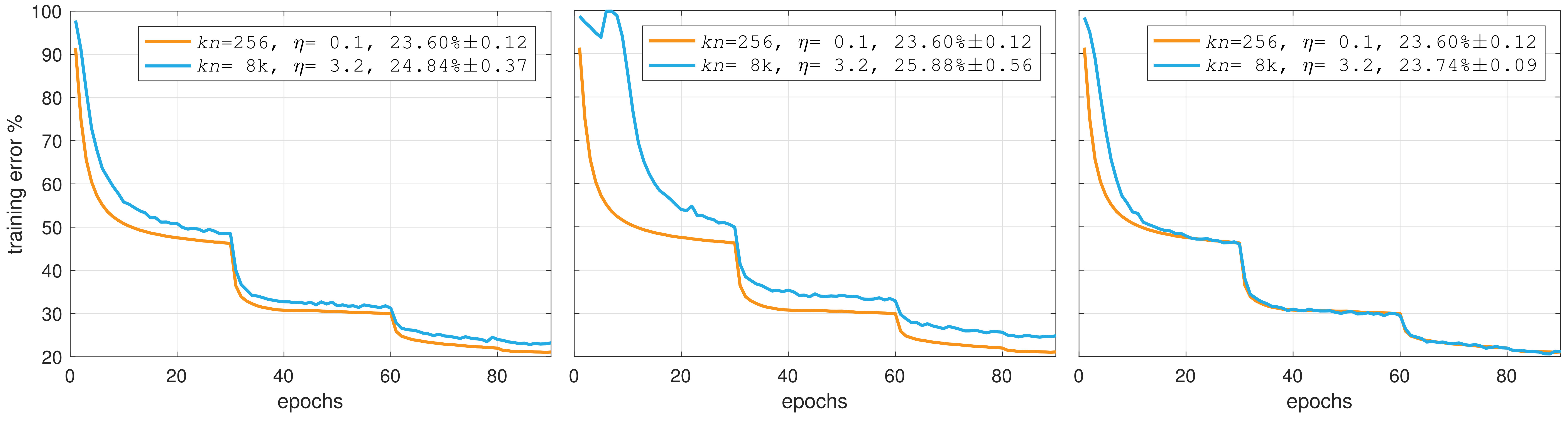
$$\hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_t)$$

Suggests that if $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ **for *j* < *k* then minibatch SGD with size *n* and learning rate** η **can be approximated by large mini batch SGD with size *kn* if the learning rate is also scaled to** $k\eta$

When does $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ not hold?

1. **At beginning of training**
   - **Suggests starting training with smaller learning rate (learning rate "warmup")**
2. **When minibatch size begins to get too large (there is a limit to scaling minibatch size)**





(a) no warmup (b) constant warmup (c) gradual warmup

Minibatch size = 256 (orange) vs. 8192 (blue)

[Figure credit: Goyal et al. 2017]

Many cool ideas popping up

- **Gradient compression**
  - Since the main source of communication is communicating gradients, compress the gradients (or reduce the frequency of gradient update)

- **Account of communication latency in SGD momentum calculations**
  - Asynchronous execution means SGD continues forward (with potentially stale gradients)
  - SGD with momentum has a similar effect (keep descending in the same direction, don't directly follow gradient)
  - Idea: reduce momentum proportionally to latency of gradient update

[Lin et al. ICLR 2018]

Example: "gradient compression”

- **Each node computes gradients for minibatch, but only sends gradients with magnitude above a threshold**

- **Locally accumulate gradients below threshold over multiple SGD steps (then send when exceed threshold)**

$$G_0^k = 0$$

for all iterations *t*:

$$G_t^k = G_{t-1}^k + \eta \frac{1}{Nb} \sum_{k=1}^{N} \sum_{x \in B_k}^{b} \nabla f(x; w_t)$$

Compress and send ONLY the elements of G_t^k greater than threshold.
(then locally zero out sent elements)

SGD update on each node only uses the sent weights.

Summary: training large networks in parallel

- **Data-parallel training with asynchronous update to efficiently use clusters of commodity machines with low speed interconnect**
  - Modification of SGD algorithm to meet constraints of modern parallel systems
  - Effects on convergence are problem dependent and not particularly well understood
  - Efficient use of fast interconnects may provide alternative to these methods (facilitate tightly orchestrated solutions much like supercomputing applications)

- **Modern DNN designs, large minibatch sizes, careful learning rate schedules enable scalability without asynchronous execution on commodity clusters**

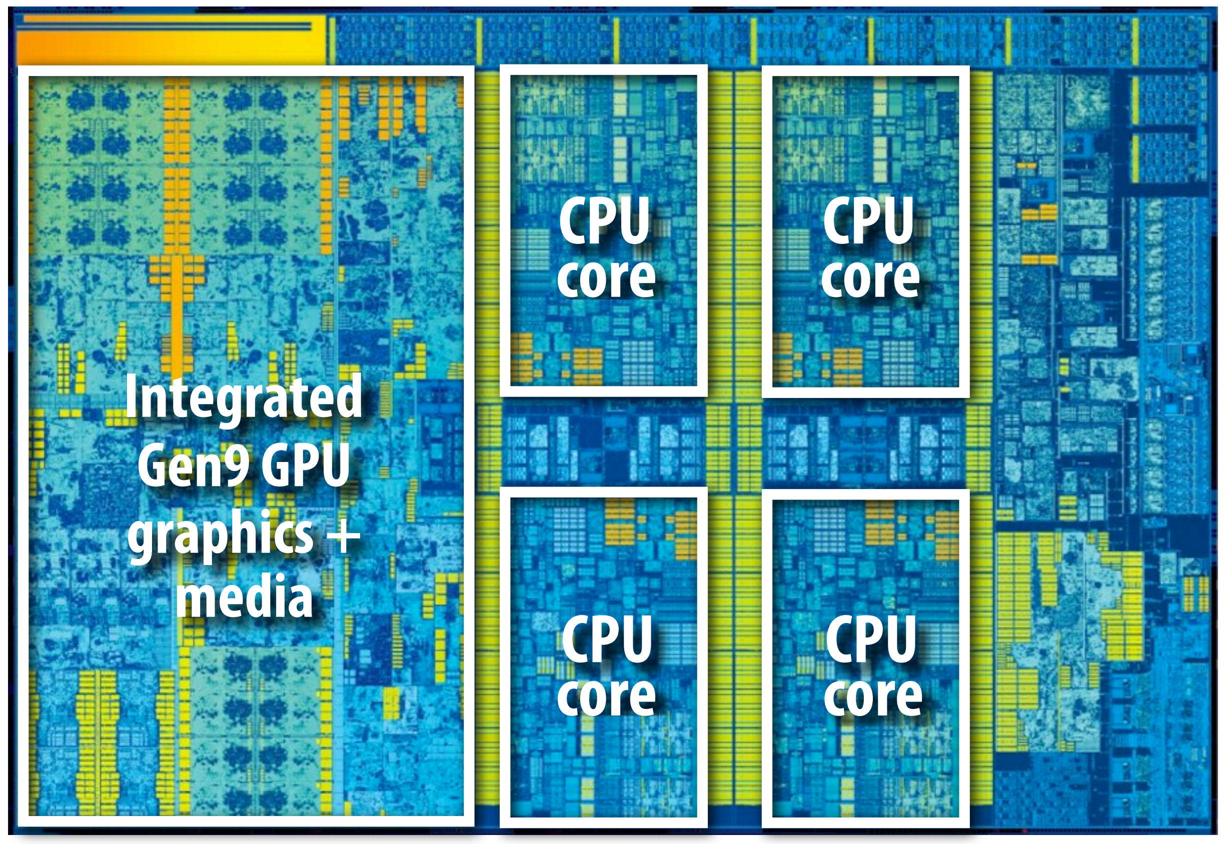
- **High-performance training of deep networks is an interesting example of constant iteration of algorithm design and parallelization strategy (a key theme of this course!)**

Course Wrap Up

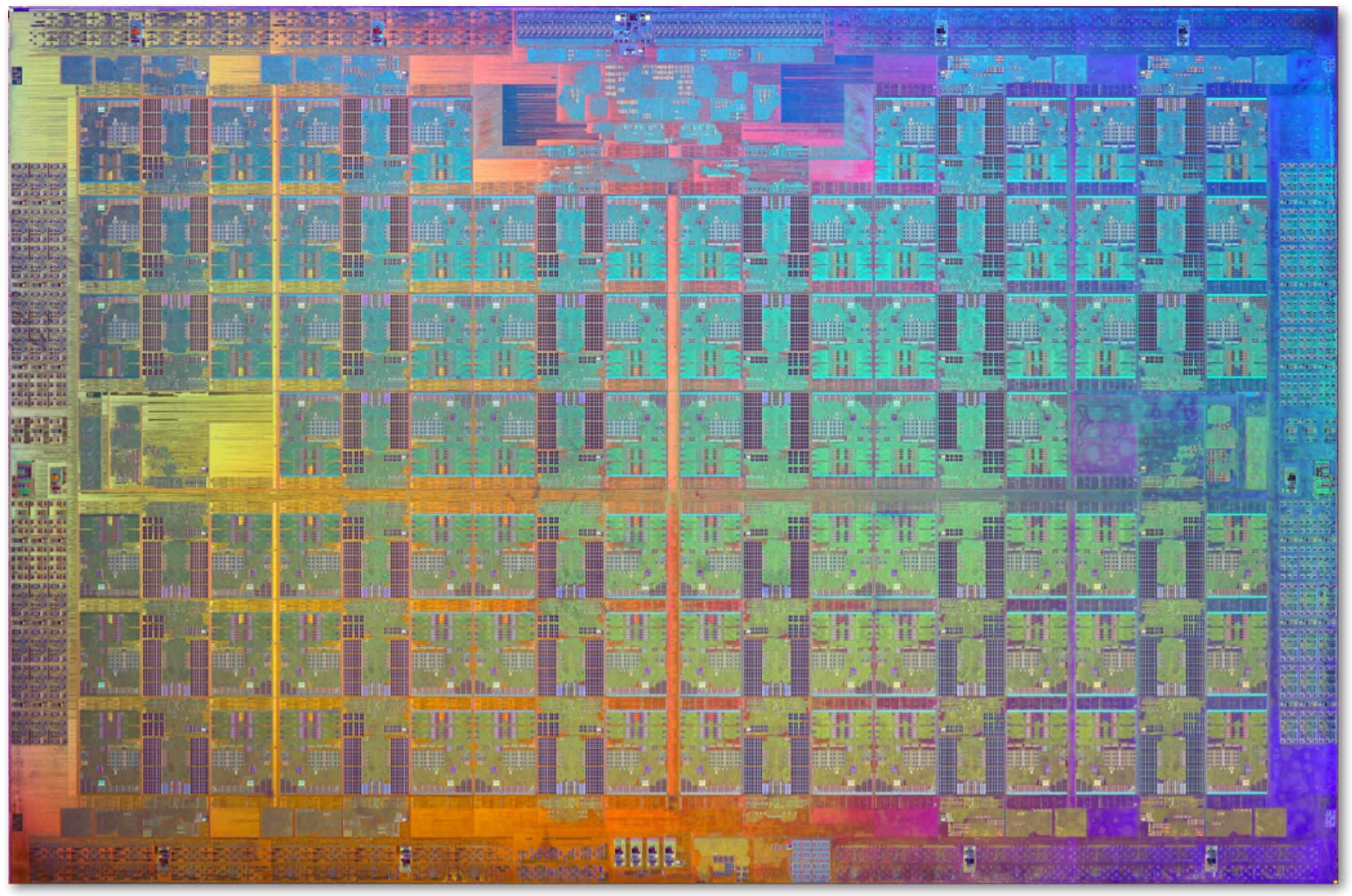
😆😎😂💃🕺

(Students)

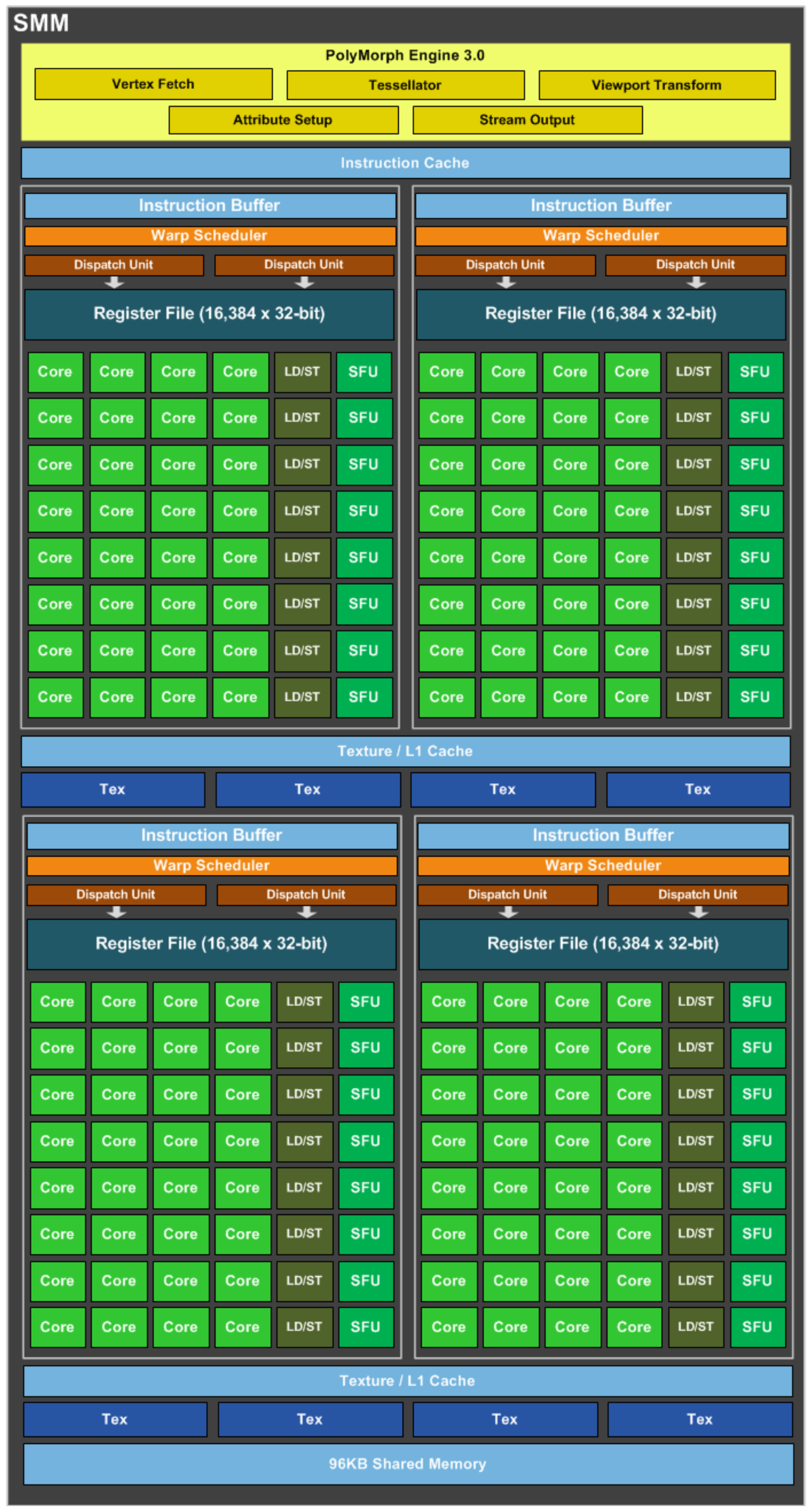
For the foreseeable future, the primary way to obtain higher performance computing hardware is through a combination of increased parallelism and hardware specialization.



Intel Core i7 CPU + integrated GPU and media

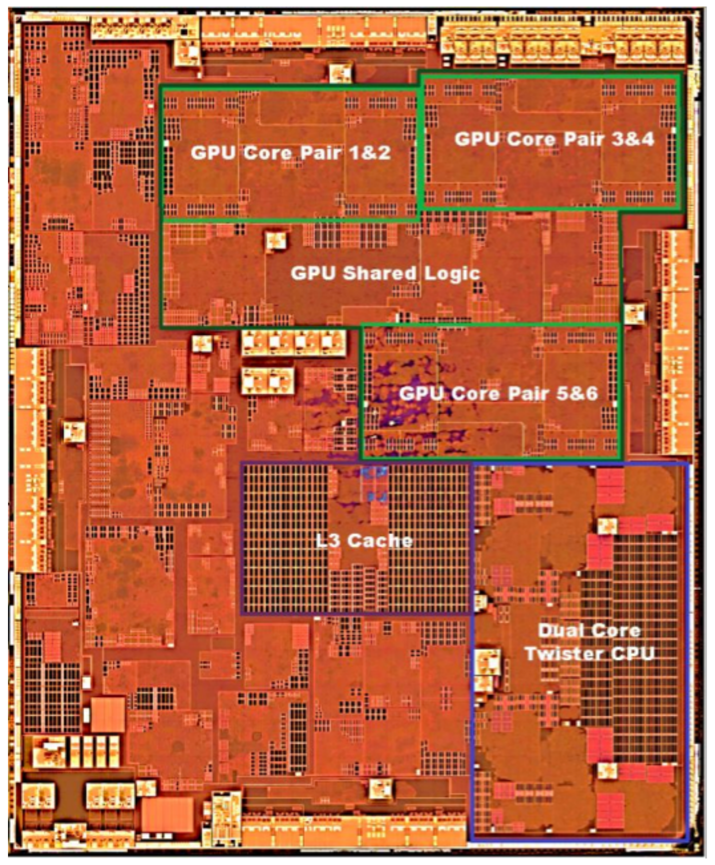


Intel Xeon Phi
72 cores, 16-wide SIMD, 4-way multi-threading



NVIDIA Maxwell GPU
(single SMM core)
32 wide SIMD
2048 CUDA/core threads per SMM

FPGA
(reconfigurable logic)



Apple A9
Heterogeneous SoC
multi-core CPU + multi-core GPU + media ASICs

Today's software is surprisingly inefficient compared to the capability of modern machines

A lot of performance is currently left on the table (increasingly so as machines get more complex, and parallel processing capability grows)

Extracting this performance stands to provide a notable impact on many compute-intensive fields (or, more importantly enable new applications of computing!)

Given current software programming systems and tools, understanding how a parallel machine works is important to achieving high performance.

A major challenge going forward is making it simpler for programmers to extract performance on these complex machines.

This is very important given how exciting (and efficiency-critical) the next generation of computing applications are likely to be.

Key issues we have addressed in this course

Identifying parallelism

(or conversely, identifying dependencies)

Efficiently scheduling parallelism

1. Achieving good workload balance

2. Overcoming communication constraints:

Bandwidth limits, dealing with latency, synchronization

Exploiting data/computation locality = <u>efficiently managing state</u>!

3. Scheduling under heterogeneity (using the right processor for the job)

We discussed these issues at many scales and in many contexts

Heterogeneous mobile SoC
Single chip, multi-core CPU
Multi-core GPU
CPU+GPU connected via bus
Clusters of machines
Large scale, multi-node supercomputers

Key issues we have addressed in this course

Abstractions for thinking about efficient code

Data parallel thinking

Functional parallelism

Transactions

Tasks

How throughput-oriented hardware works

Multiple cores, hardware-threads, SIMD

Specialization

After taking this course, you are ready to try undergraduate research in parallel computing!

Why research (or independent study)?

- You will learn <u>way more</u> about a topic than in any class.
- You think your undergrad friends are very smart? Come hang out with Stanford Ph.D. students! (you get to work side-by-side with them and with faculty). Imagine what level you might rise to.
- It's way more fun to be on the cutting edge. Industry might not even know about what you are working on. (imagine how much more valuable you are if you can teach them)
- It widens your mind as to what is possible.

Example: what my own Ph.D. students are working on these days…

- Generating efficient code from image processing or deep learning DSLs (Halide Autoscheduler), and compiling these applications directly to FGPAs
- Designing a new shading language for future real-time 3D graphics pipelines (collaboration with NVIDIA)
- Parallel computing platforms that make it simpler and more efficient to analyzing large video collections (Scanner project: "Spark for video")
- Designing programming models for querying video collections (e.g, find frames with "three people around a table" or where DNN1 disagrees with DNN2)
- Designing more efficient DNNs to accelerate image processing on video

Maybe you might like research and decide you want to go to grad school

Pragmatic comment: Without question, the number one way to get into a top grad school is to receive a strong letter of recommendation from faculty members. You get that letter only from being part of a research team for an extended period of time.

DWIC letter: ("did well in class" letter) What you get when you ask for a letter from a faculty member who you didn't do research with, but got an 'A' in their class. This letter is essentially thrown out by the Ph.D. admissions committee at good schools.

A very good reference

CMU Professor Mor Harchol-Balter's writeup:

"Applying to Ph.D. Programs in Computer Science"

http://www.cs.cmu.edu/~harchol/gradschooltalk.pdf

Research is just one option…

(Despite what many of us biased faculty tell you, there are many other good ones as well)

Why not start your own project?

Interested in applying computer science to a problem that excites you? Give it a shot!

Like a topic enough to be your own boss? Consider starting your own company.

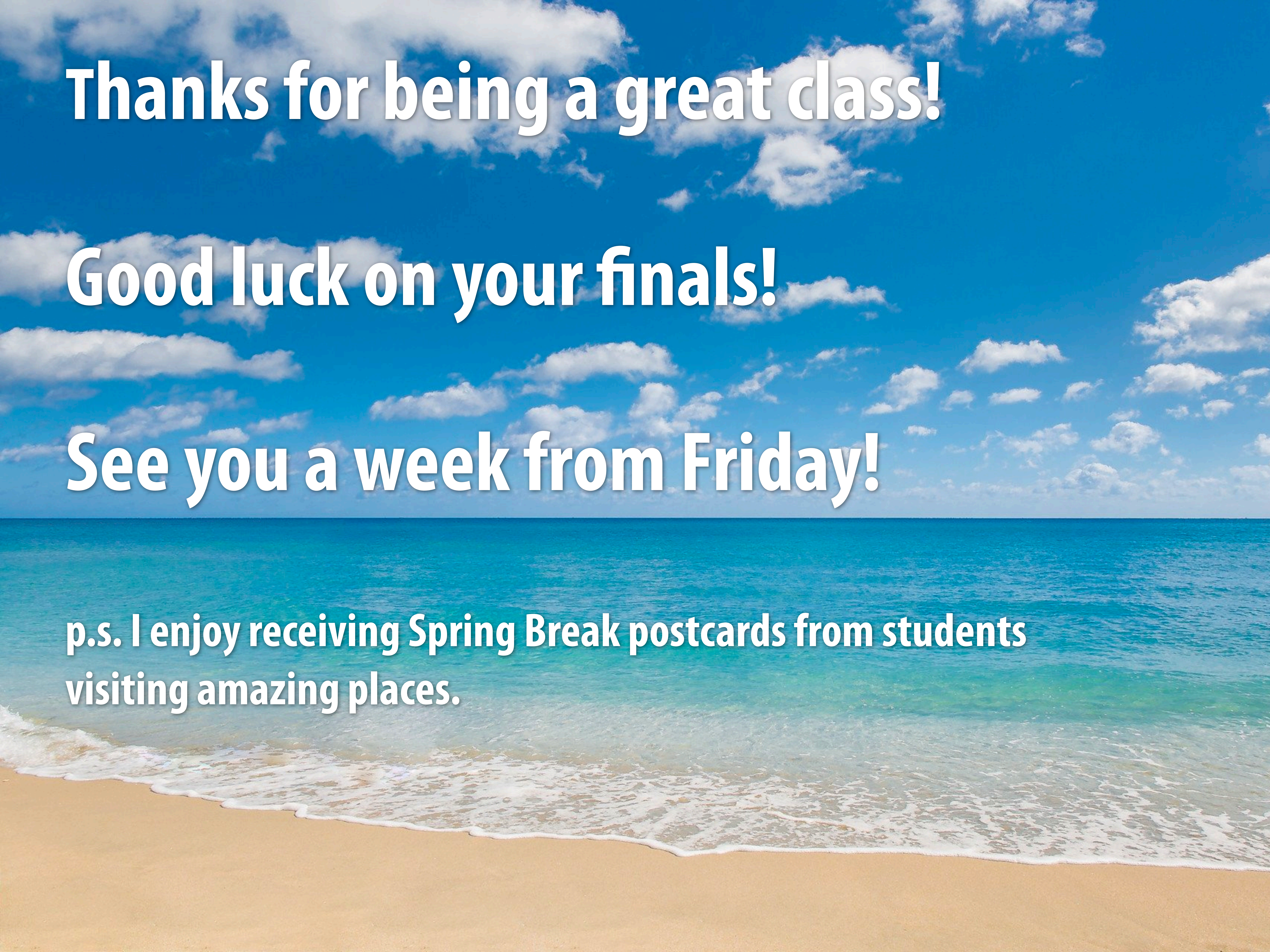
Why go work for Google or Facebook when you can start a company that beats them?

(yes, those are great jobs too!)

Your professors encourage you to be brave and take risks.

You are lucky because you are extremely talented. The cost of "messing up" for you is <u>actually much less</u> than for other students because your backup plan is very good.

Be ambitious while at Stanford with opportunities beyond just classes. If it doesn't work out, you'll try something else and you'll probably succeed… or end up getting the good job you would have gotten anyway.



Thanks for being a great class!

Good luck on your finals!

See you a week from Friday!

p.s. I enjoy receiving Spring Break postcards from students visiting amazing places.