Lecture 9:

Parallel Deep Network Training

Visual Computing Systems

Stanford CS348K, Fall 2018

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?

Quick review of back-propagation

Derivatives using the chain rule

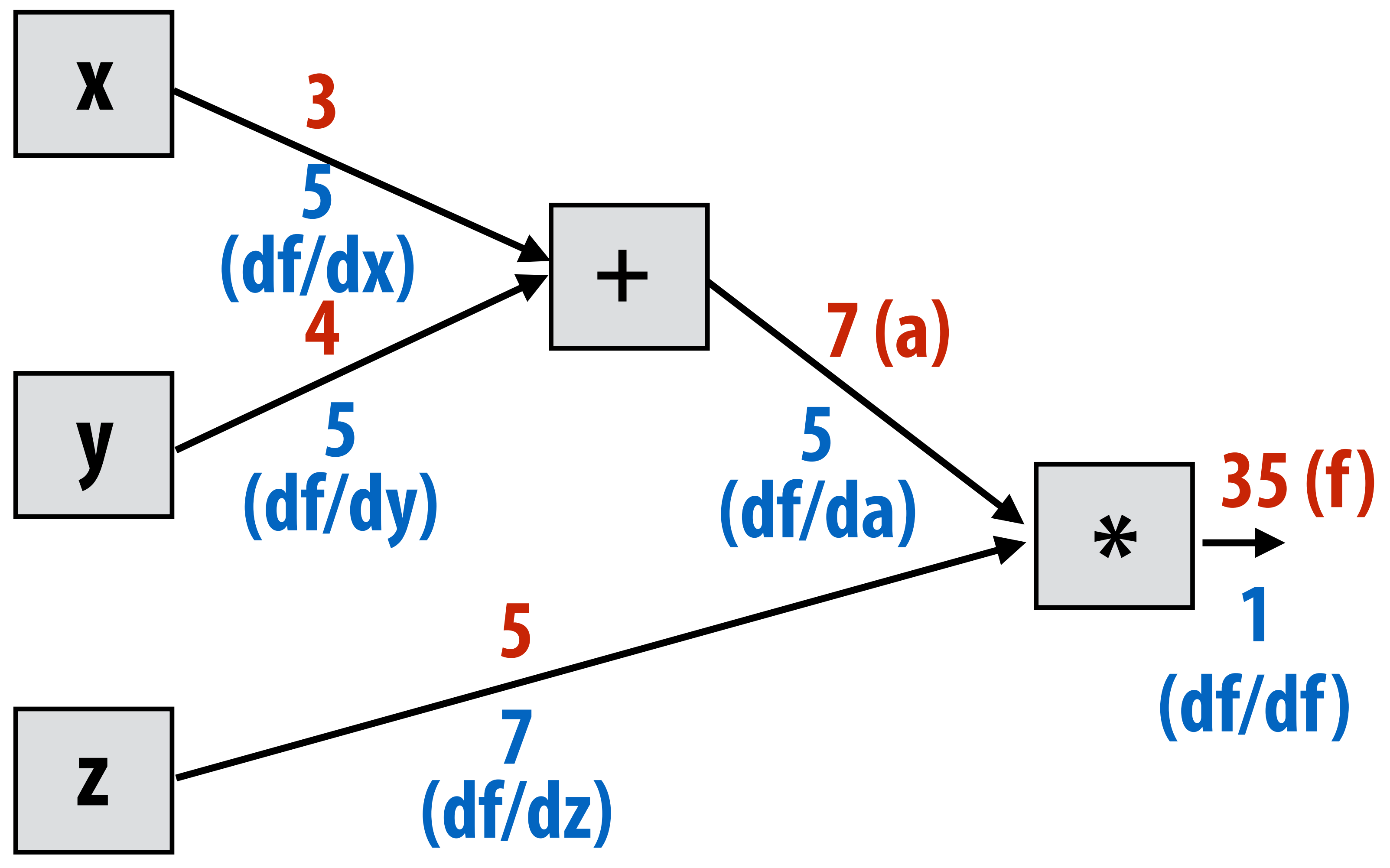
$$f(x, y, z) = (x + y)z = az$$

Where: $a = x + y$

$$\frac{df}{da} = z \qquad \frac{da}{dx} = 1 \qquad \frac{da}{dy} = 1$$

So, by the derivative chain rule:

$$\frac{df}{dx} = \frac{df}{da}\frac{da}{dx} = z$$

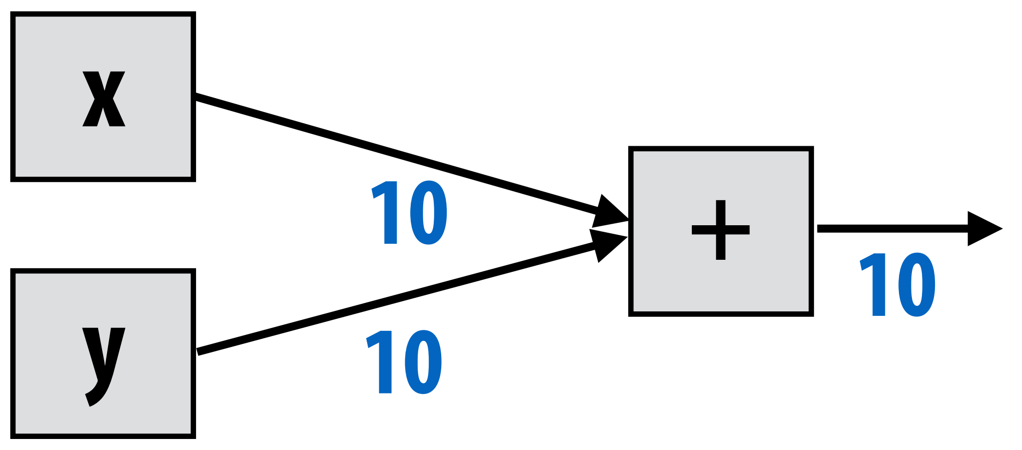


Red = output of node
Blue = df/dnode

Backpropagation

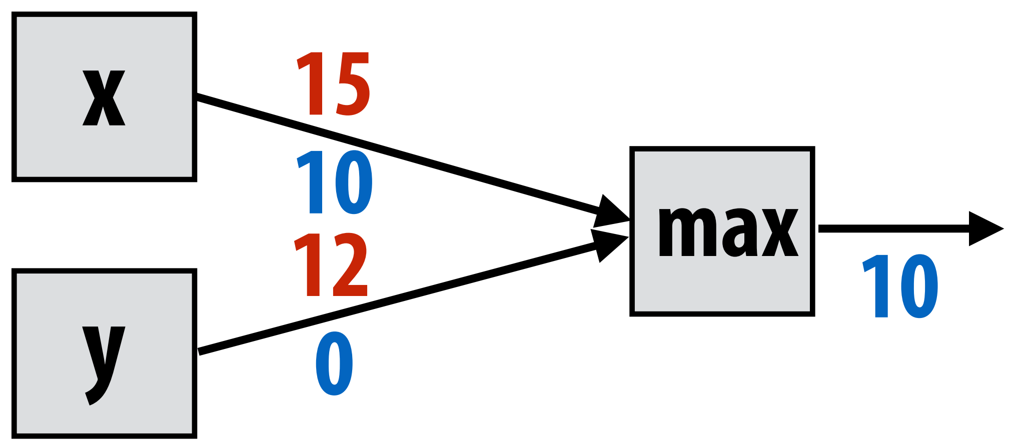
Red = output of node
Blue = df/dnode

Recall: $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$



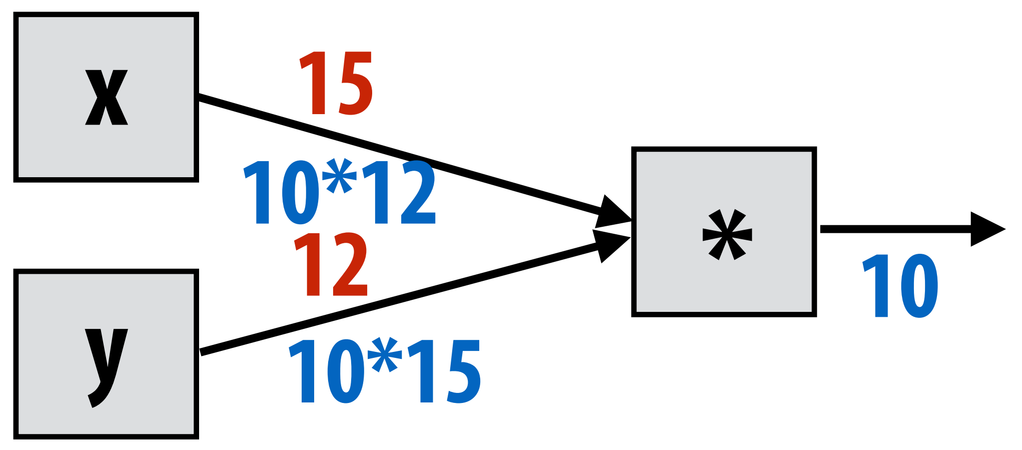
$g(x,y) = x + y$

$\frac{dg}{dx} = 1\,,\ \frac{dg}{dy} = 1$



$g(x,y) = \max(x,y)$

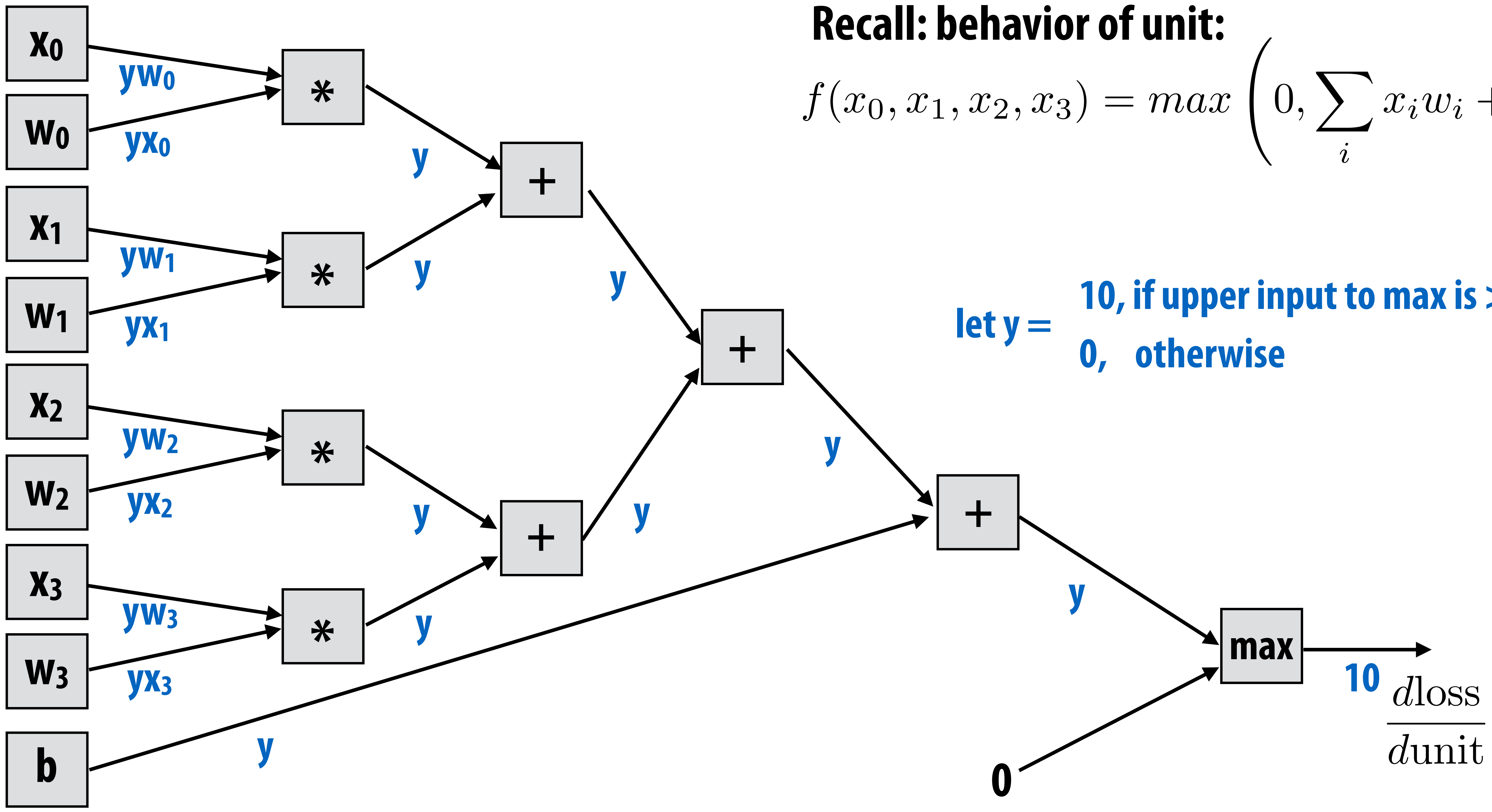
$\frac{dg}{dx} =$ **1, if x > y**
0, otherwise



$g(x,y) = xy$

$\frac{dg}{dx} = y\,,\ \frac{dg}{dy} = x$

Back-propagating through single unit



Recall: behavior of unit:

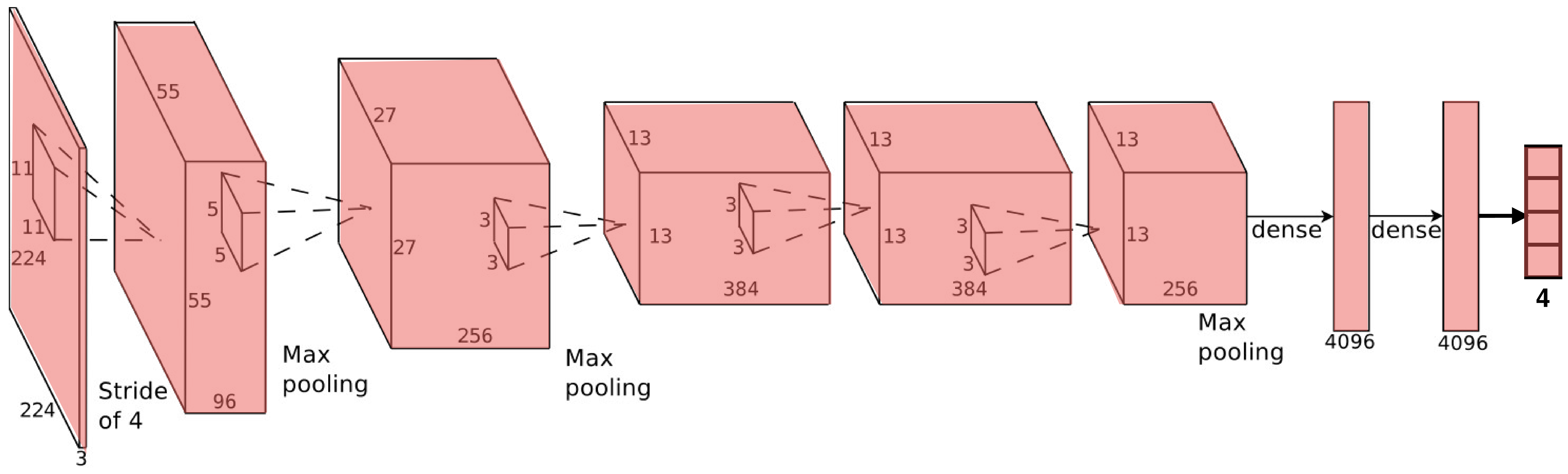
$$f(x_0, x_1, x_2, x_3) = max\left(0, \sum_i x_i w_i + b\right)$$

let y = 10, if upper input to max is > 0
0, otherwise

$$\frac{d\text{loss}}{d\text{unit}}$$

Observe: output of prior layer must be retained in order to compute weight gradients for this unit during backprop.

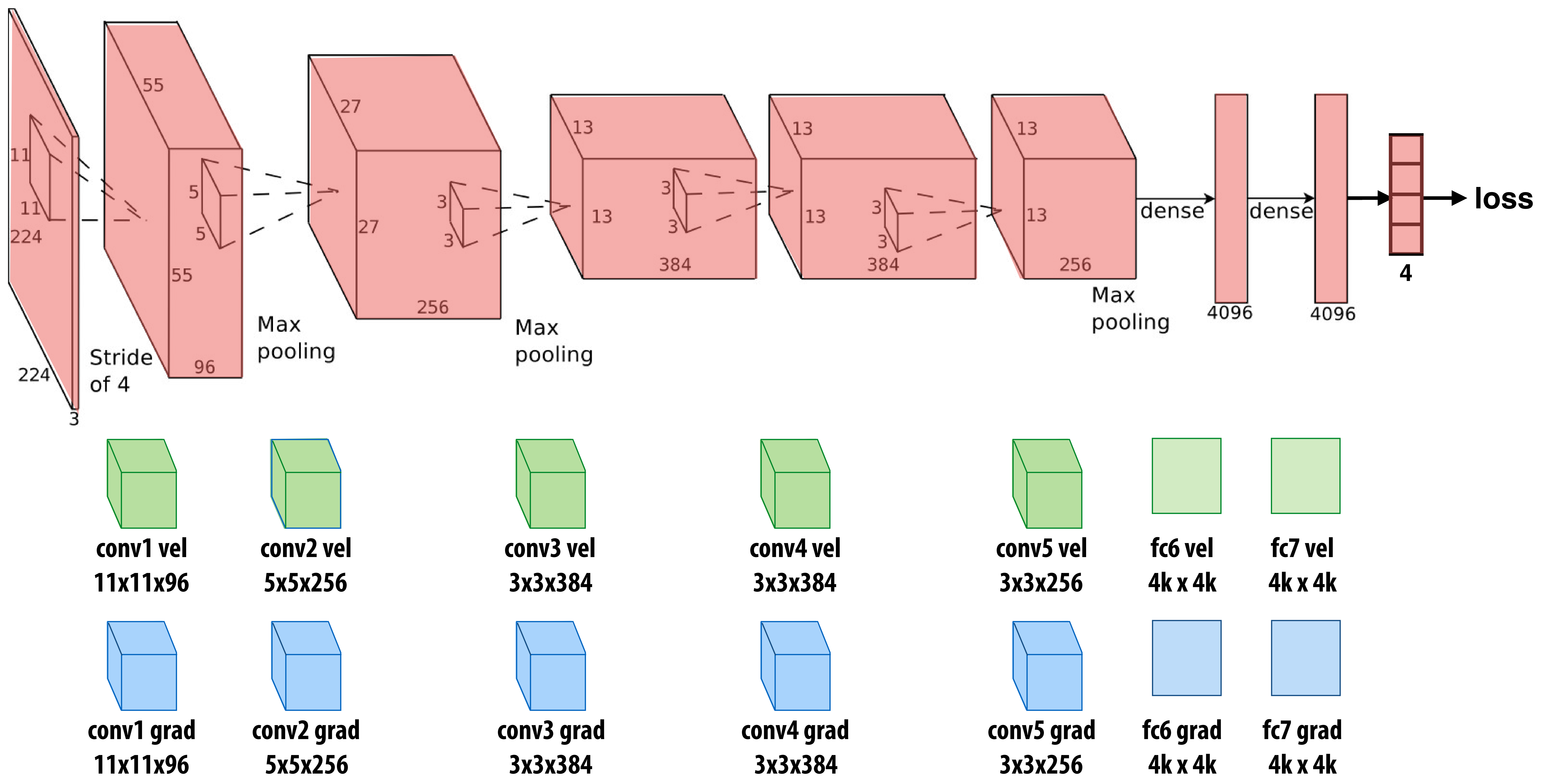
Data lifetimes during network evaluation



Weights (read-only) reside in memory

After evaluating layer i, can free outputs from layer i-1

Data lifetimes during training



- Must retain outputs for all layers because they are needed to compute gradients during back-prop
- Parallel back-prop will require storage for per-weight gradients (more about this in a second)
- In practice: may also store per-weight gradient velocity (if using SGD with "momentum") or step size cache in adaptive step size schemes like Adagrad

```
vel_new = mu * vel_old - step_size * grad
w_new = w_old + vel_new
```

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

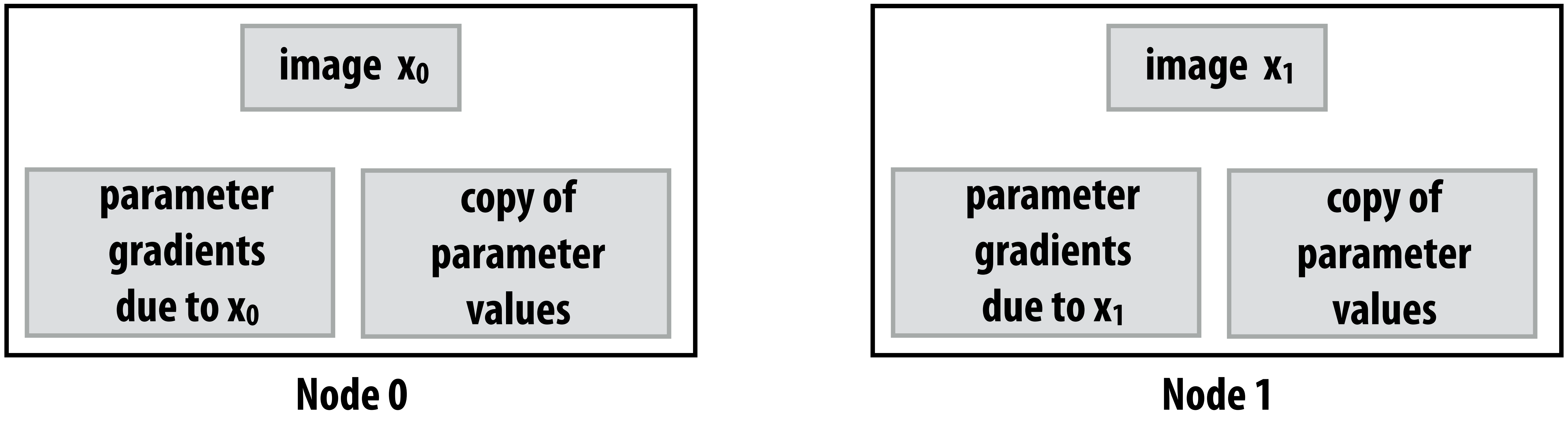
- **Large memory footprint**
  - Must maintain network layer outputs from forward pass
  - Additional memory to store gradients/gradient velocity for each parameter
  - Recall parameters for popular VGG-16 network require ~500 MB of memory (training requires GBs of memory for academic networks)
  - Scaling 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*).

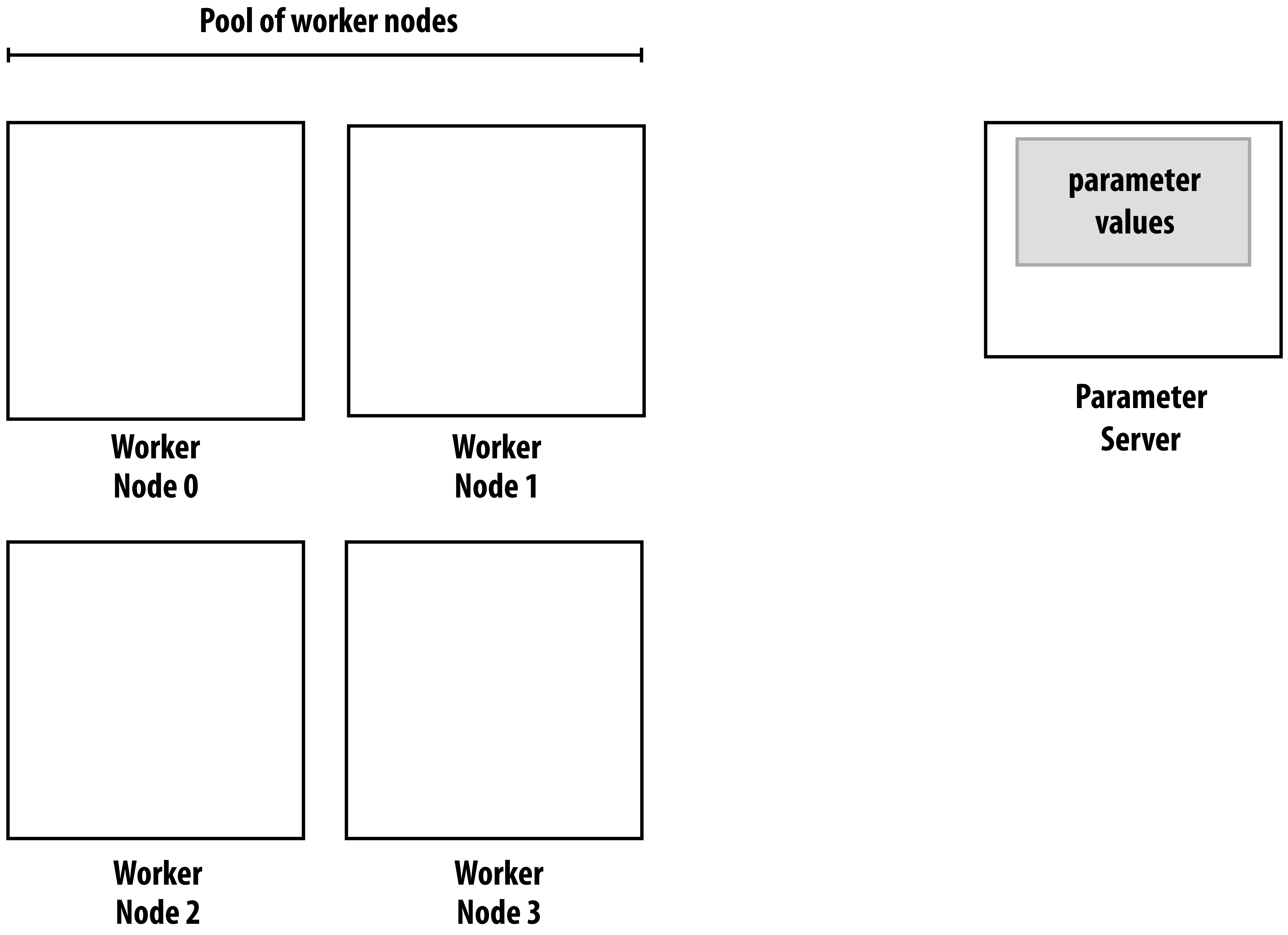
Challenges of scaling out (many nodes)

- 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)

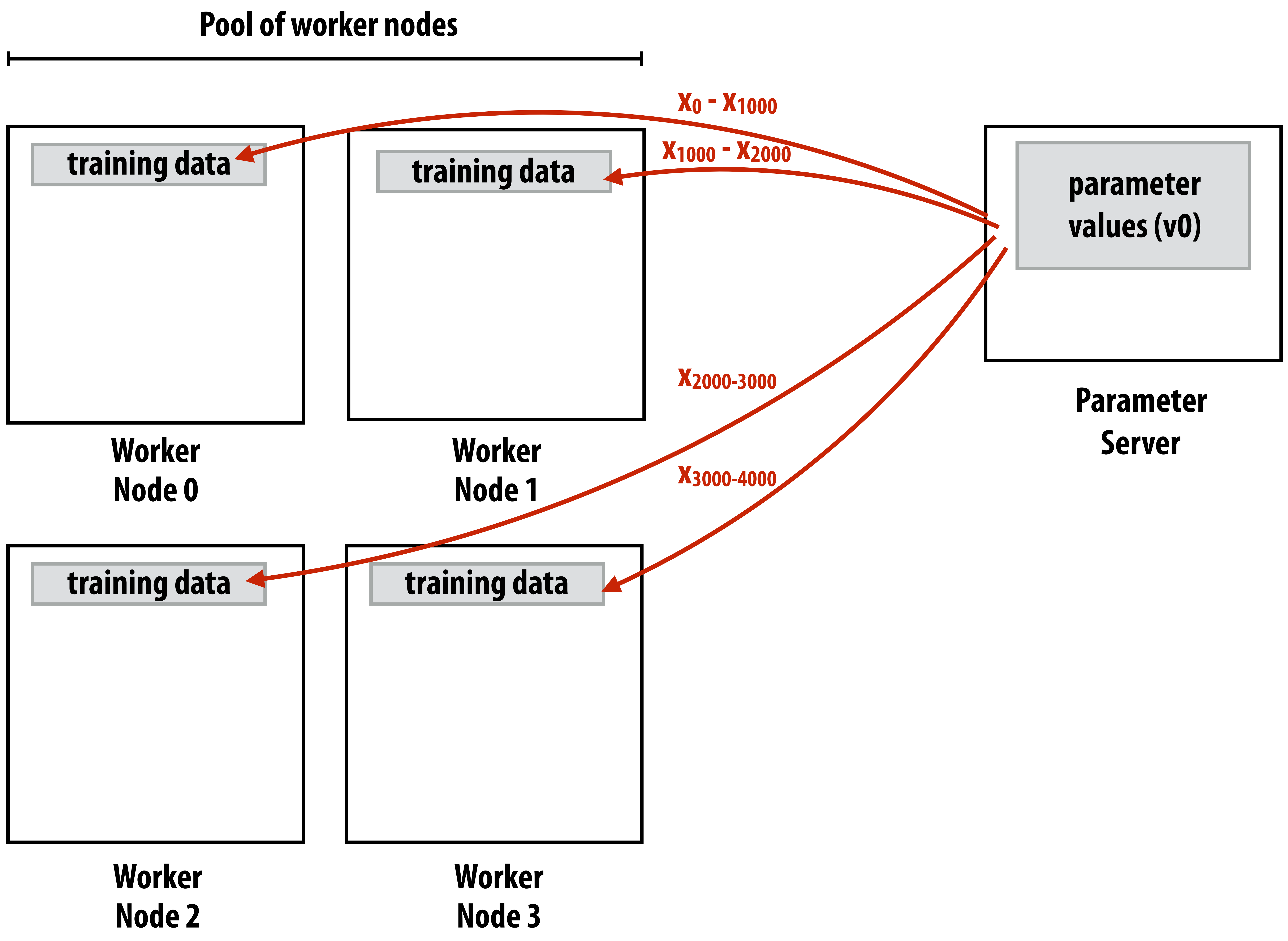
Alternative solution: exploit properties of SGD by using asynchronous execution

Parameter server design

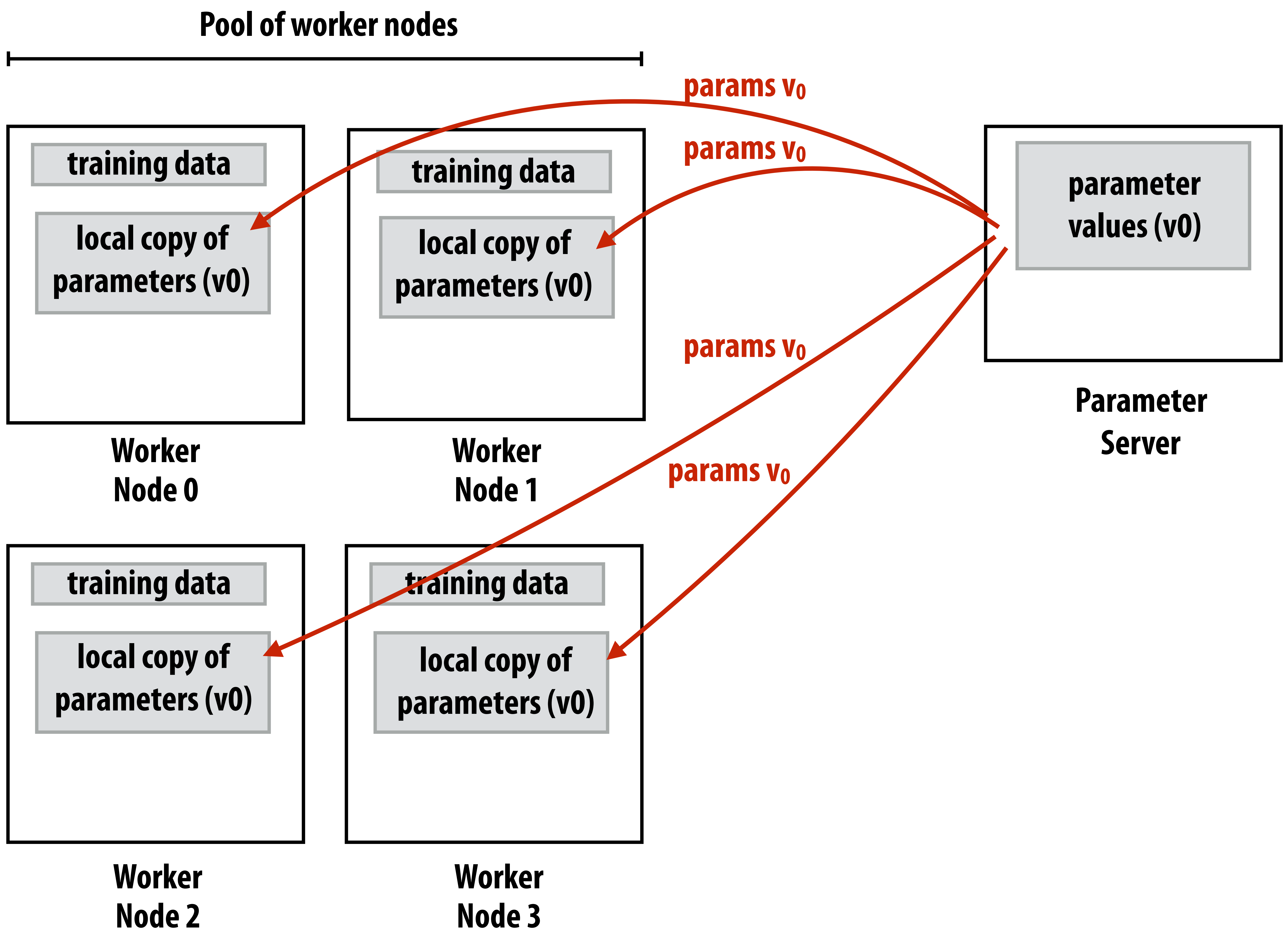
Google's DistBelief [Dean NIPS12]
Parameter Server [Li OSDI14]
Microsoft's Project Adam [Chilimbi OSDI14]



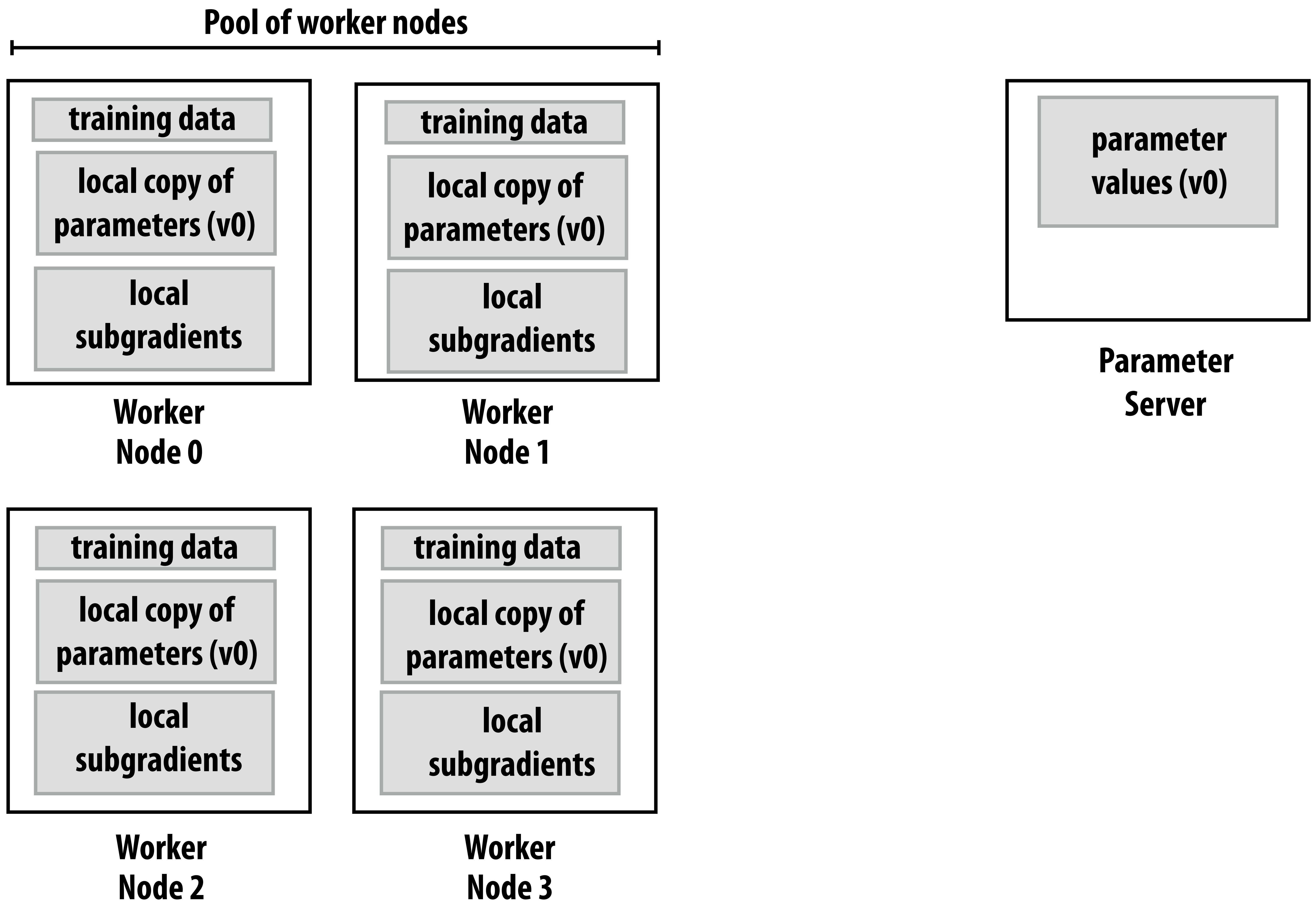
Training data partitioned among workers



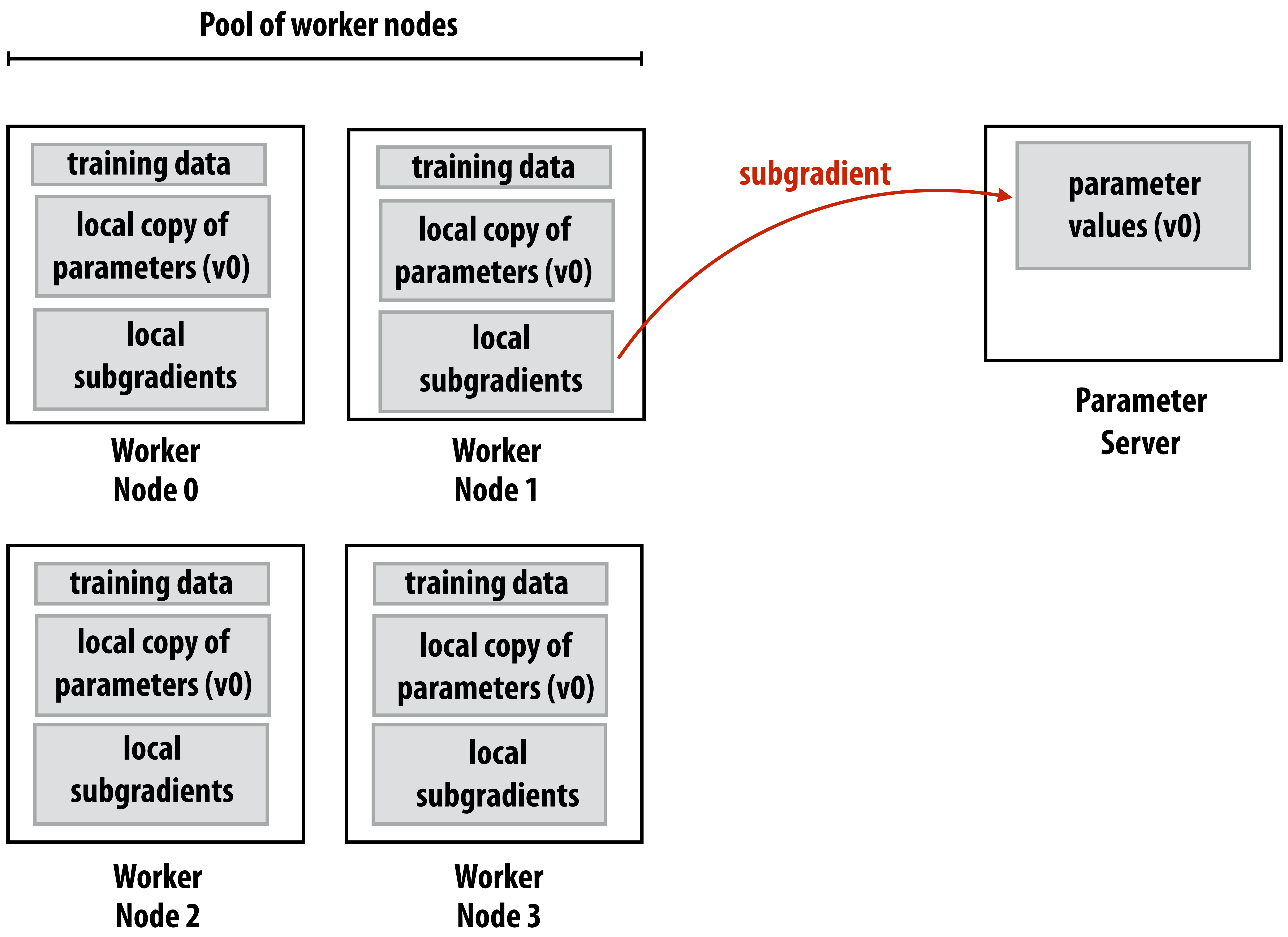
Copy of parameters sent to workers



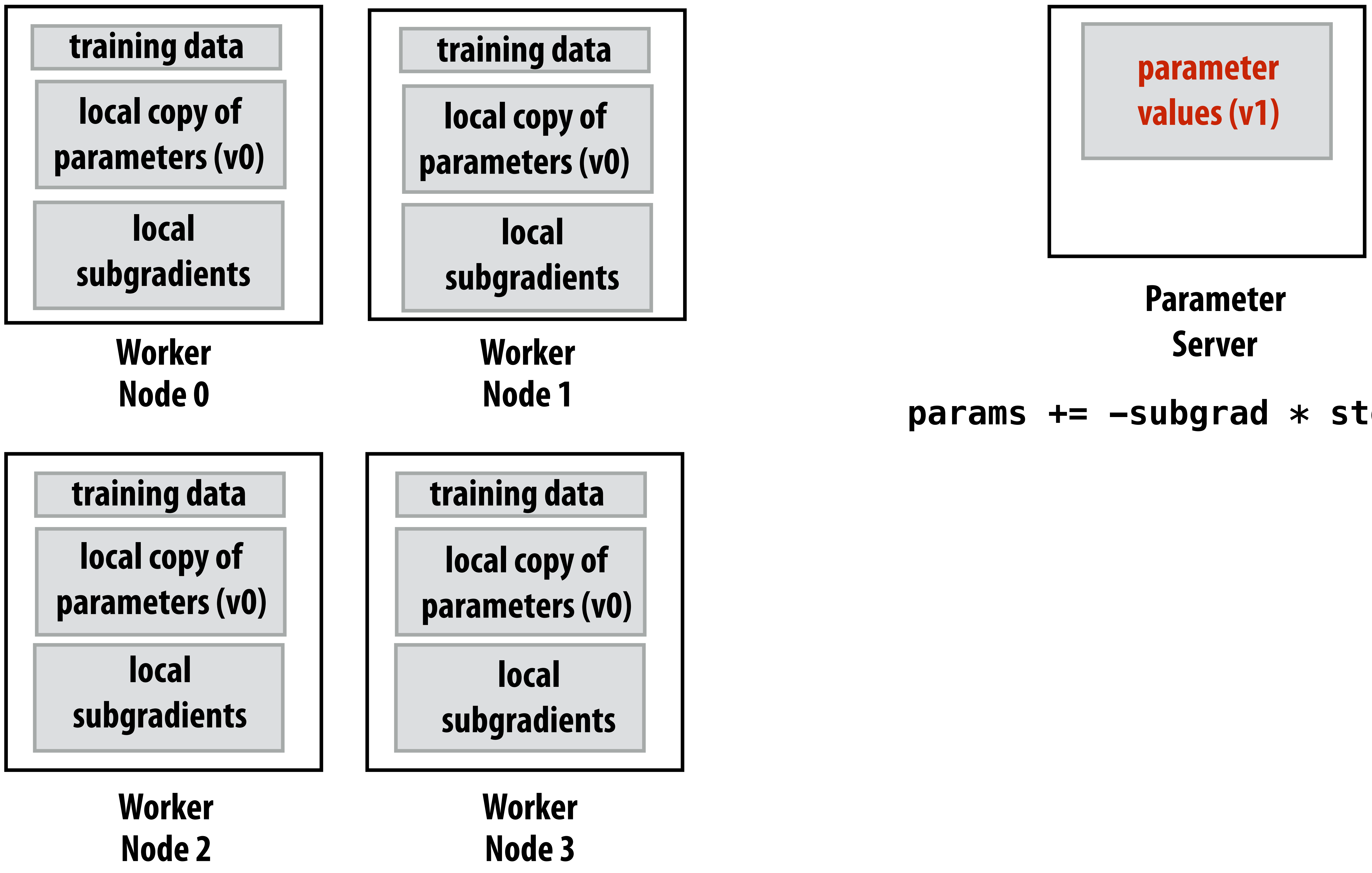
Data parallelism: workers independently compute local "subgradients" on different pieces of data



Worker sends subgradient to parameter server



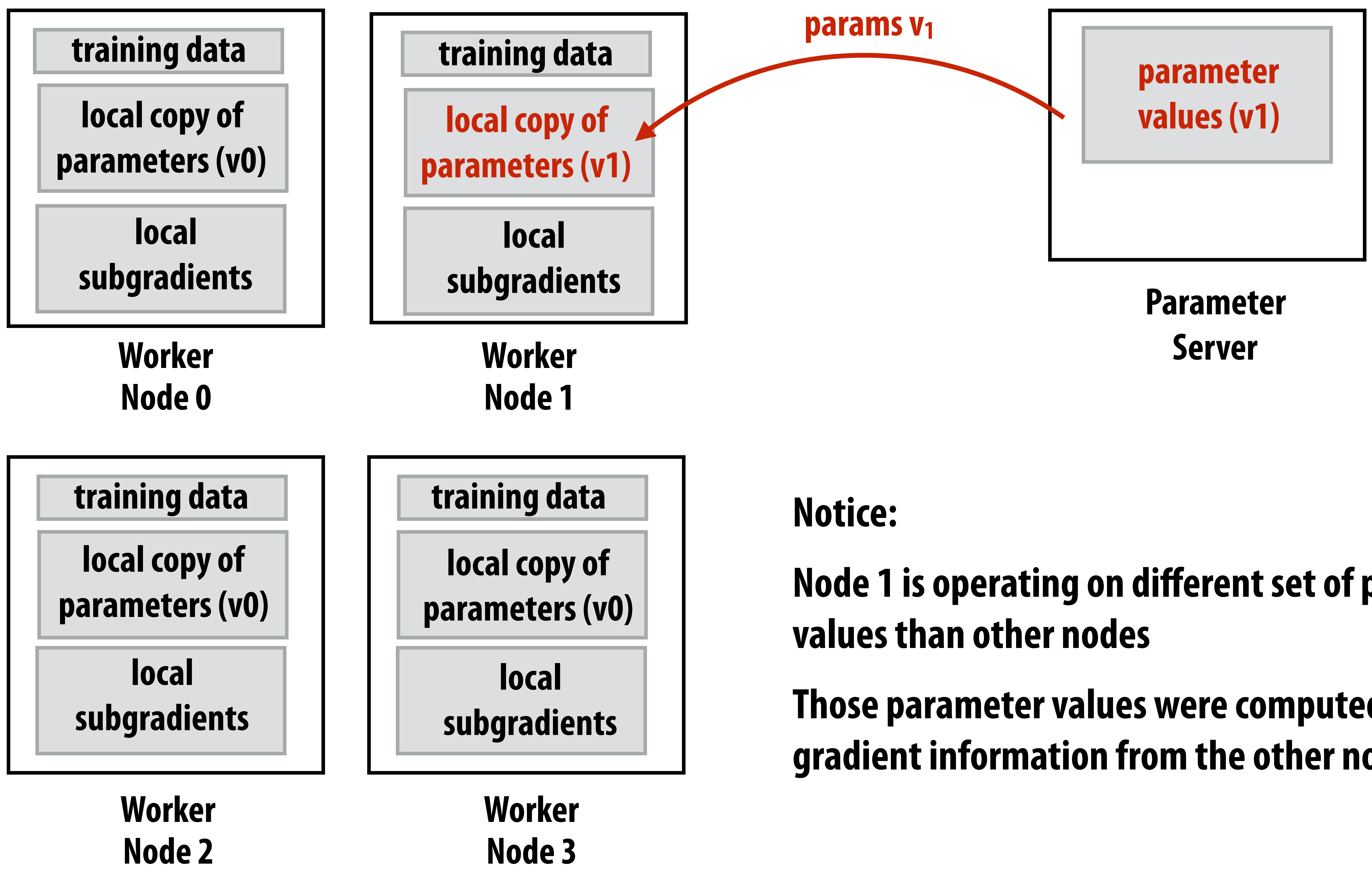
Server updates global parameter values based on subgradient



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

Updated parameters sent to worker

Then worker proceeds with another gradient computation step

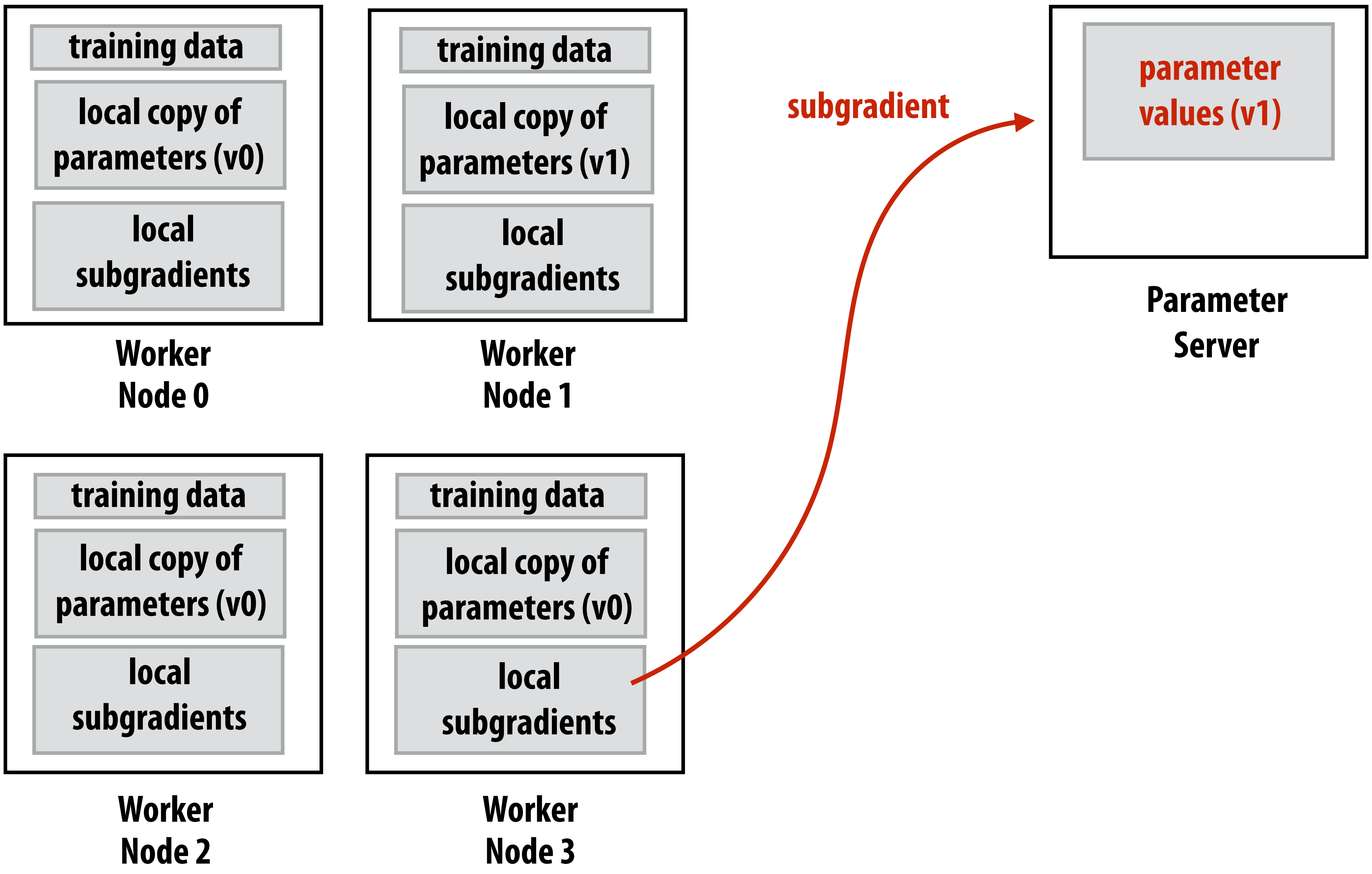


Notice:

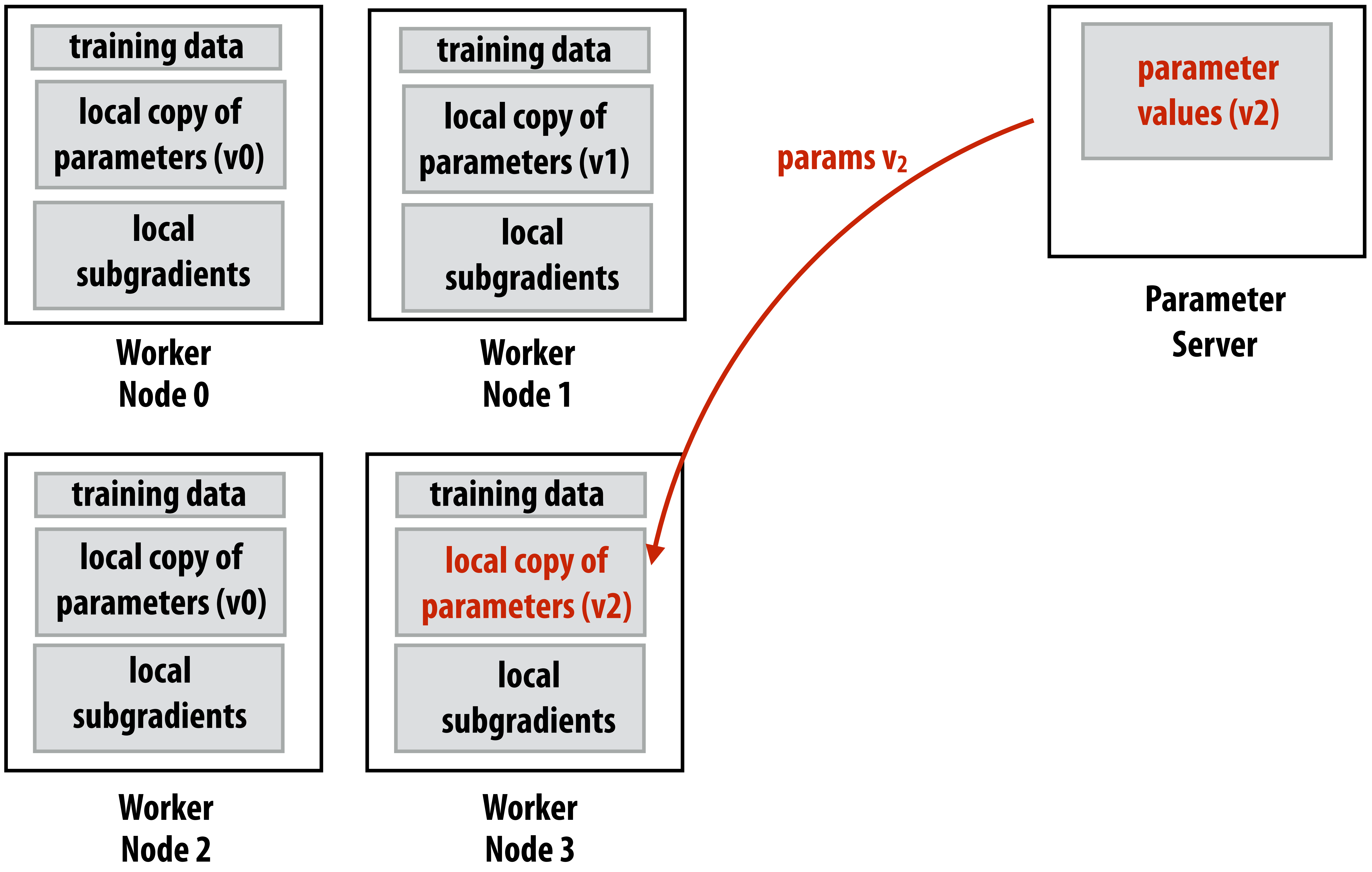
Node 1 is operating on different set of parameter values than other nodes

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

Updated parameters sent to worker (again)



Worker continues with updated parameters

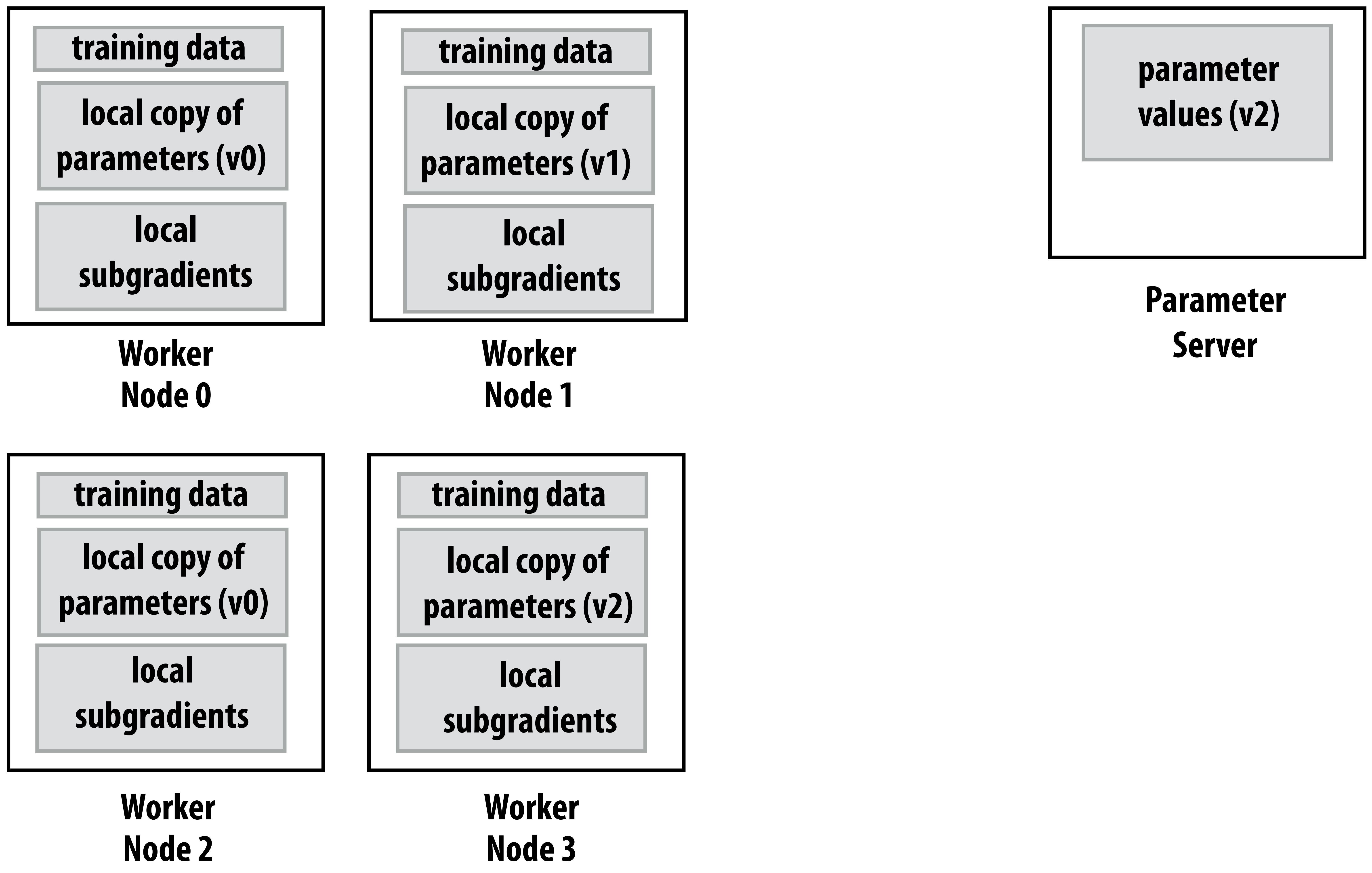


Summary: asynchronous parameter update

- **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?

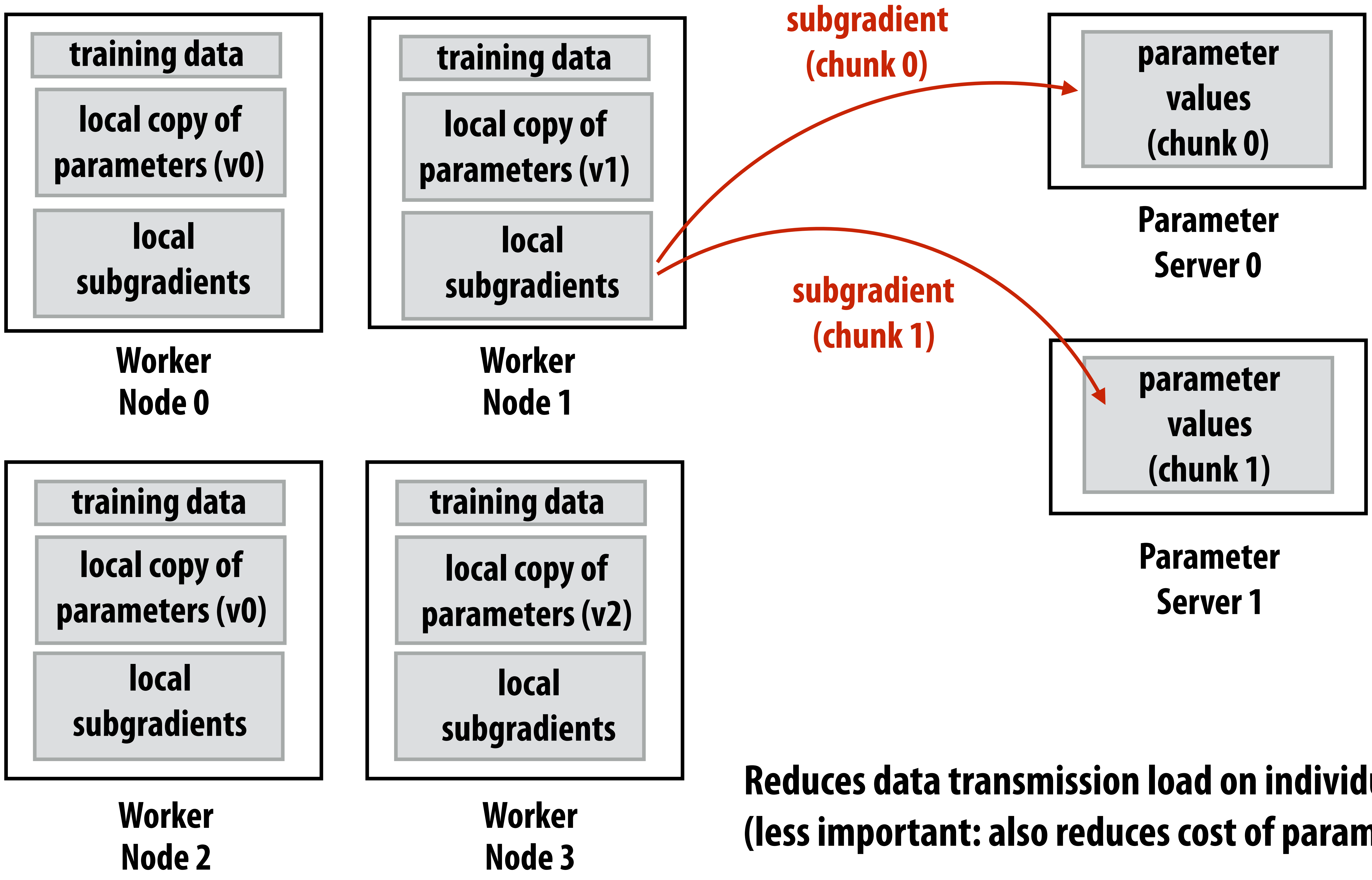
What if there is heavy contention for parameter server?



Shard the parameter server

Partition parameters across servers

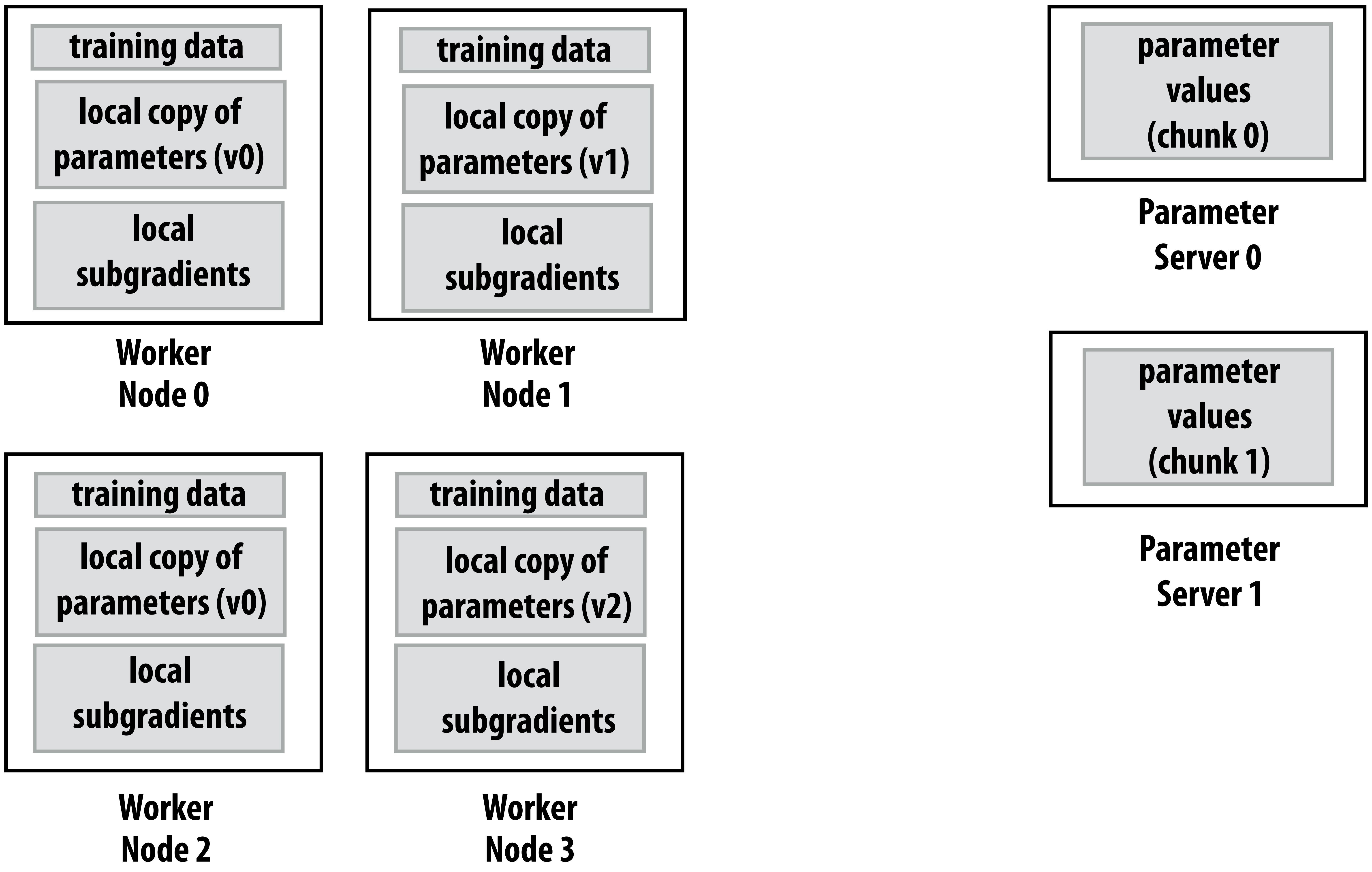
Worker sends chunk of subgradients to owning parameter server



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)

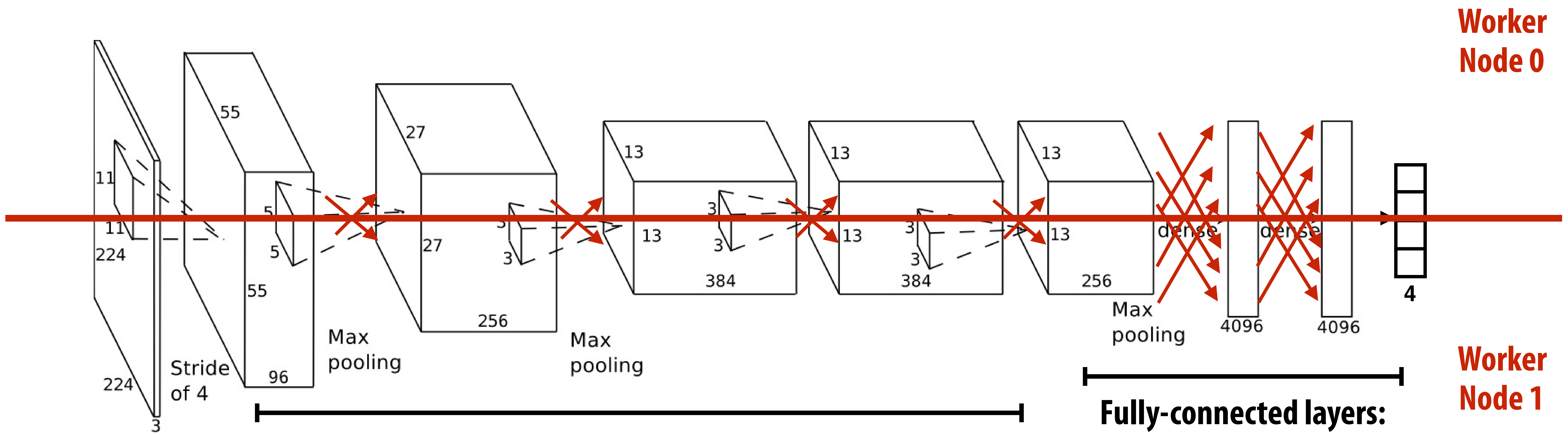


Model parallelism

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

Reduce internode communication through network design:

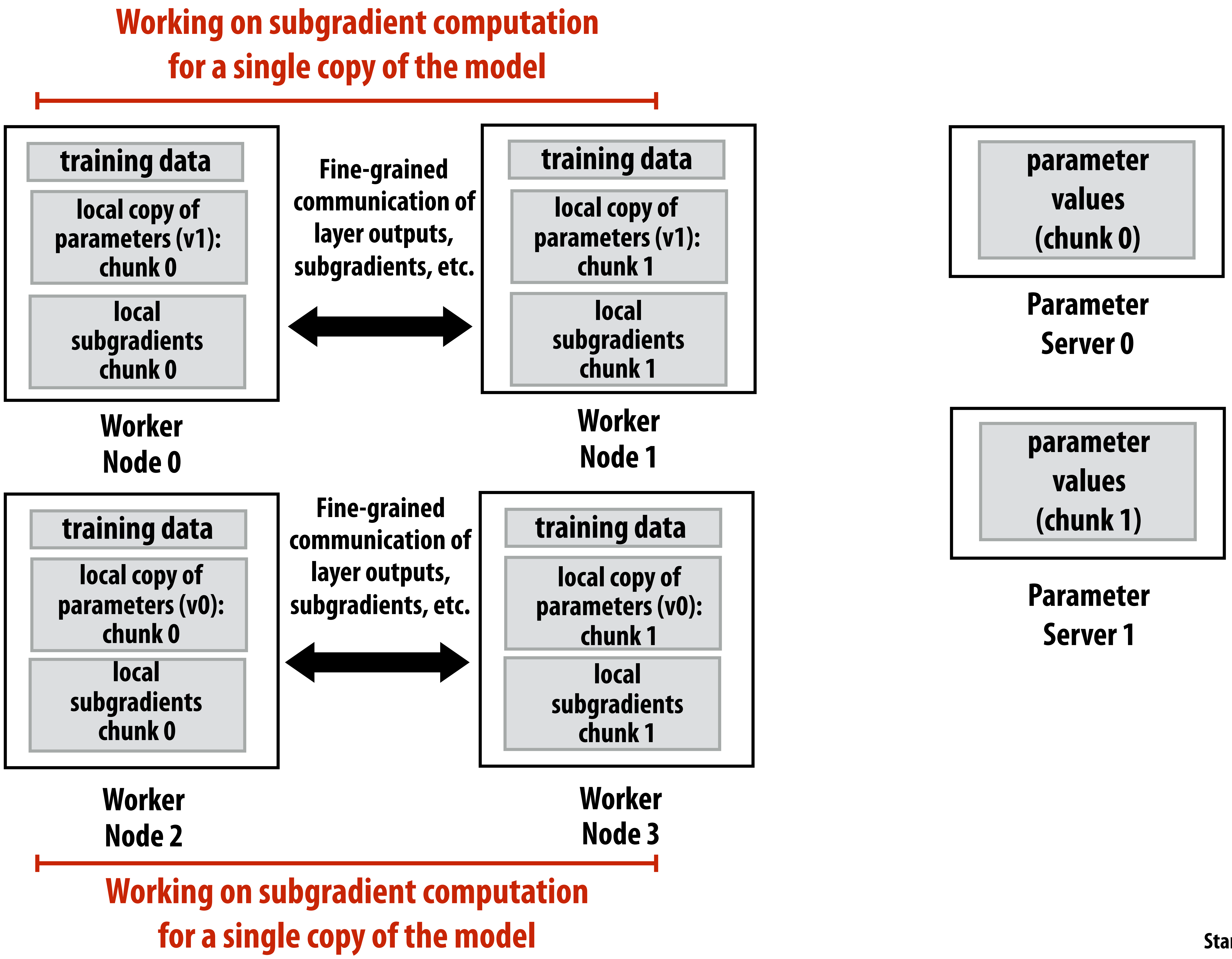
- Use small spatial convolutions (1x1 convolutions)
- Reduce/shrink fully-connected layers



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

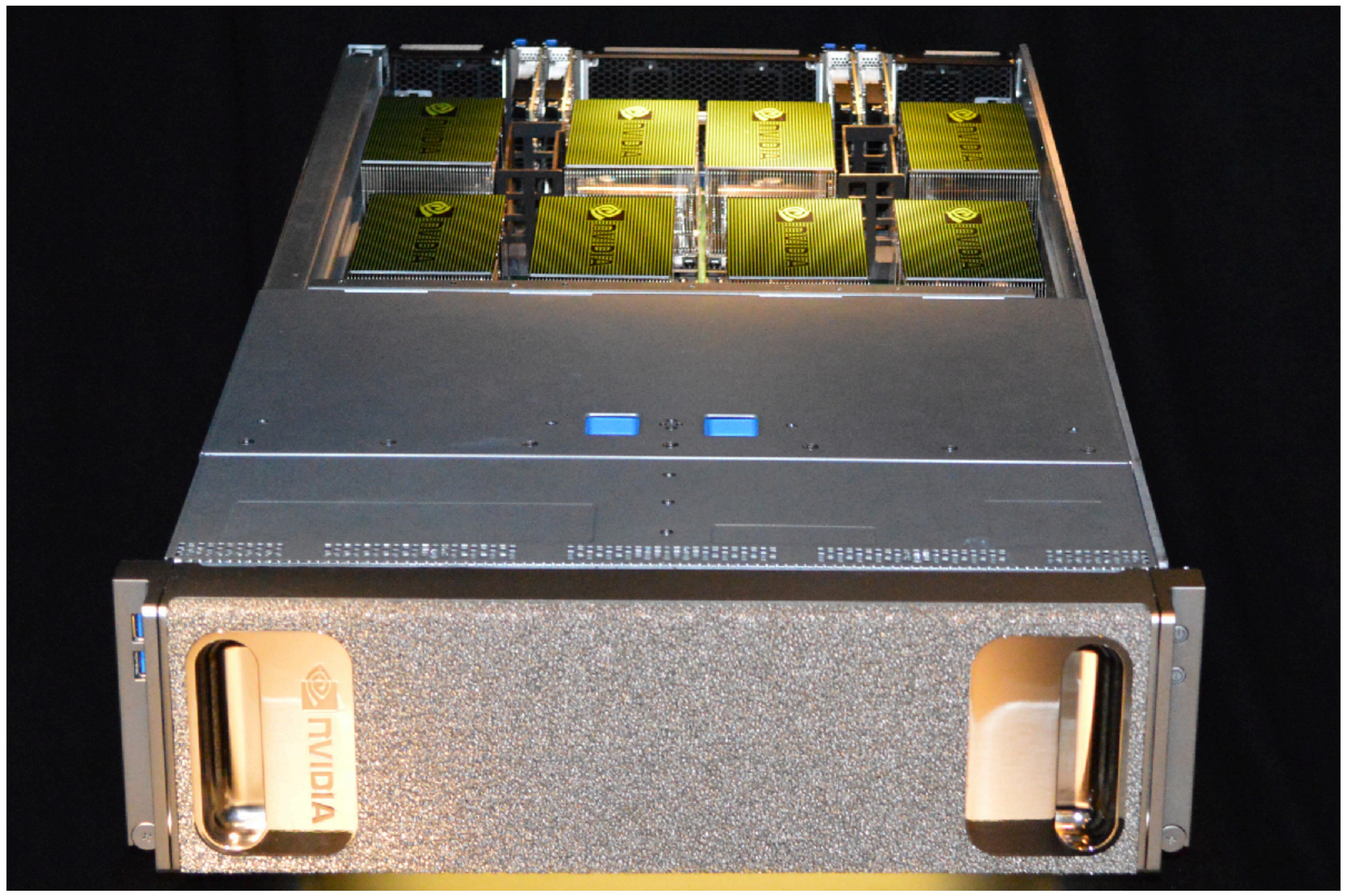
- Asynchronous training: significant distributed system complexity incurred to combat bandwidth/latency constraints of modern cluster computing

- 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)



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

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

- Larger mini-batches increase compute-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

[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)

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

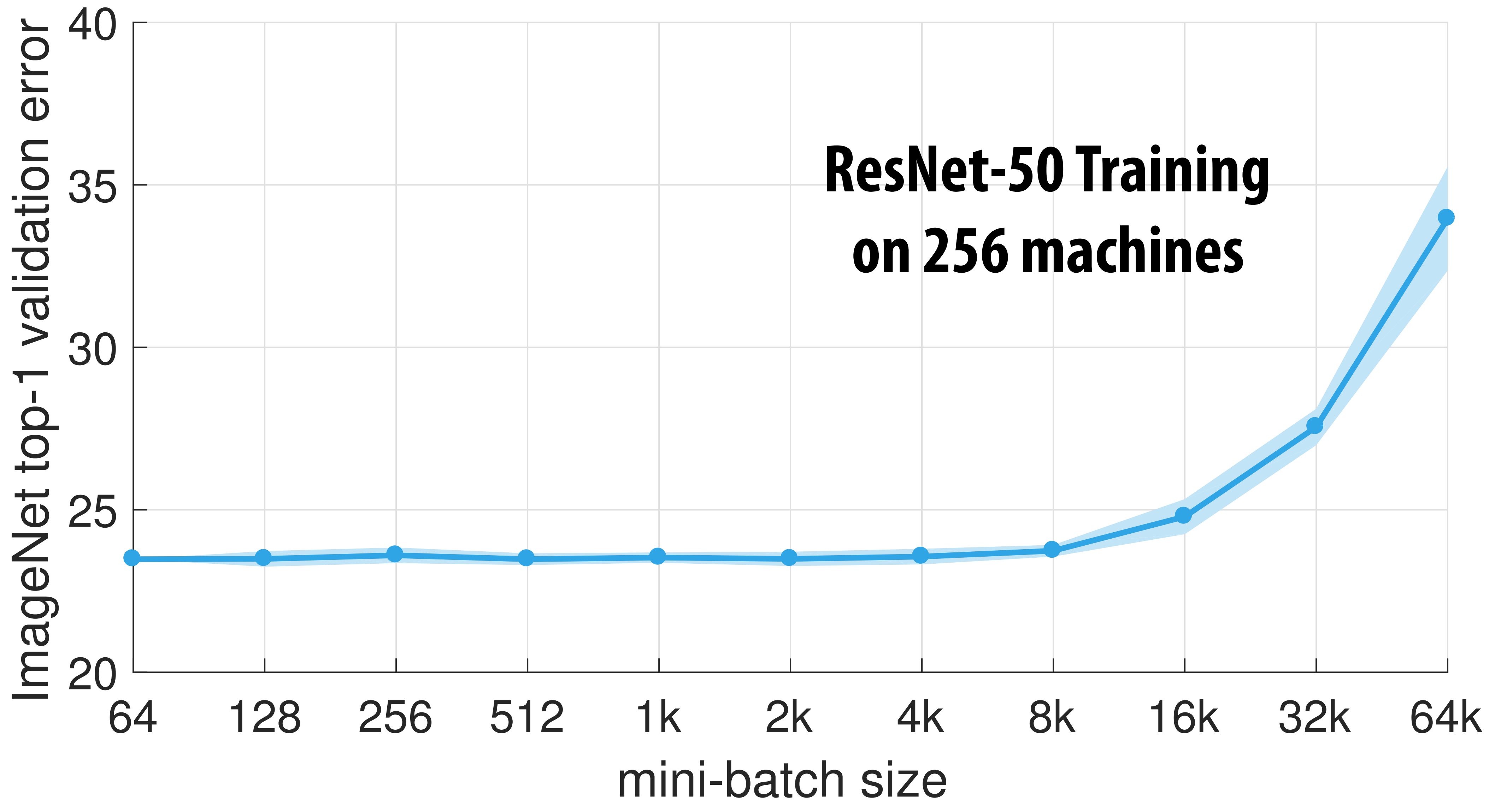
Consider processing one minibatch that is of size kn (one step of gradient descent)

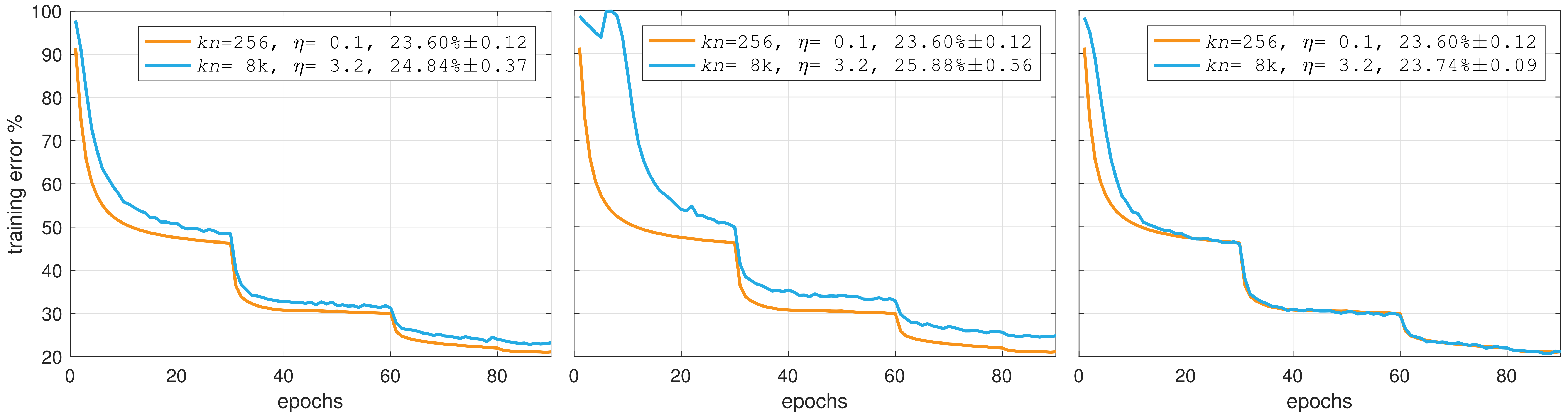
$$\hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_t)$$

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]

[Lin et al. ICLR 2018]

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 note only uses the sent weights.

Handling momentum

Consider basic momentum in SGD:

$$u_t = m u_{t-1} + \sum_{k=1}^{N} \left(\nabla_{k,t} \right), \quad w_{t+1} = w_t - \eta u_t$$

$$\boxed{\nabla_{k,t} = \tfrac{1}{Nb} \textstyle\sum_{x \in \mathcal{B}_{k,t}} \nabla f(x, w_t)}$$

Consider weight update with momentum after T iterations of SGD

$$w_{t+T}^{(i)} = w_t^{(i)} - \eta \left[\cdots + \left(\sum_{\tau=0}^{T-2} m^{\tau} \right) \nabla_{k,t+1}^{(i)} + \left(\sum_{\tau=0}^{T-1} m^{\tau} \right) \nabla_{k,t}^{(i)} \right]$$

Basic sparse update: (what's the problem?)

$$v_{k,t} = v_{k,t-1} + \nabla_{k,t}, \quad u_t = m u_{t-1} + \sum_{k=1}^{N} sparse\left(v_{k,t} \right), \quad w_{t+1} = w_t - \eta u_t$$

Problem: momentum discount not applied correctly after sparse update interval T (assume sparse gradients propagated after T iterations of SGD)

$$w_{t+T}^{(i)} = w_t^{(i)} - \eta \left(\cdots + \nabla_{k,t+1}^{(i)} + \nabla_{k,t}^{(i)} \right)$$

Fix: locally accumulate and communicate gradient velocities, not gradients:

$$u_{k,t} = m u_{k,t-1} + \nabla_{k,t}, \quad v_{k,t} = v_{k,t-1} + u_{k,t}, \quad w_{t+1} = w_t - \eta \sum_{k=1}^{N} sparse\left(v_{k,t} \right)$$

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!)