Large Scale Machine Learning

Part 2: Distributed Logistic Regression

Previously on Large Scale ML

- Definition of Large Scale ML
- Overview of Large Scale ML software and hardware paradigms
- Large Scale ML on your machine

This episode

- Distributed Logistic Regression
  - Synchronous Distributed Gradient Descent
  - Second order methods
- Other distribution strategies
  - Parameter Server
  - AllReduce

Distributed Computing

- Using multiple networked computation nodes (computers)
- Nodes communicate and coordinate their actions by passing data on the network
- Distributed ML
  - Designing algorithms that work efficiently on distributed systems

Logistic Regression

- We are going to use logistic regression as a show case for some of the techniques shown
  - Most of what we are going to say applied to most parametric models that can be optimized with Gradient Descent

- Logistic Regression is used everywhere
  - Most common classification algorithm, can be quite very flexible
  - Criteo trains thousands of LR models per day and uses them to do billions of predictions

Logistic Regression

- Let's start with a well known algorithm to show case the design ideas
- M data points $(x, y)_i \; - \; i \; = 1..M$
- Parametrized model function $f_\theta(x) = \hat{y}$
- Loss function
  - For each data point

$$l(y, \hat{y}) = l\big(y, f_\theta(x)\big) = -\,\mathrm{y}\,.\log(\hat{y}) - (1 \; - y).log(1 \; - \; \hat{y})$$

  - For the whole dataset

$$L(\theta) = \frac{1}{M}\sum_{i=1}^{M} l(y_i, \hat{y_i})$$

- Objective

$$argmin_\theta \; L(\theta)$$

Logistic Regression

- Gradient descent
  - Repeat until convergence

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L(\theta)$$

- Stochastic gradient descent
  - Repeat until convergence
    - Sample random i from $\{1..M\}$

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla_\theta l(y_i, \widehat{y_i})$$

Logistic Regression

- Mini-batch stochastic gradient descent
  - Repeat until convergence
    - Sample a batch of data of size b

$$L_b(\theta) = \frac{1}{b}\sum_{i=1}^{b} l(y_i, \hat{y}_i)$$

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla_\theta L_b(\theta)$$

- Stochastic and mini-batch gradient descent rely on

$$E\left(\nabla_\theta l(y_i, \hat{y}_i)\right) = E\left(\nabla_\theta L_b(\theta)\right) = \nabla_\theta L(\theta)$$

Distributed Gradient Descent 101

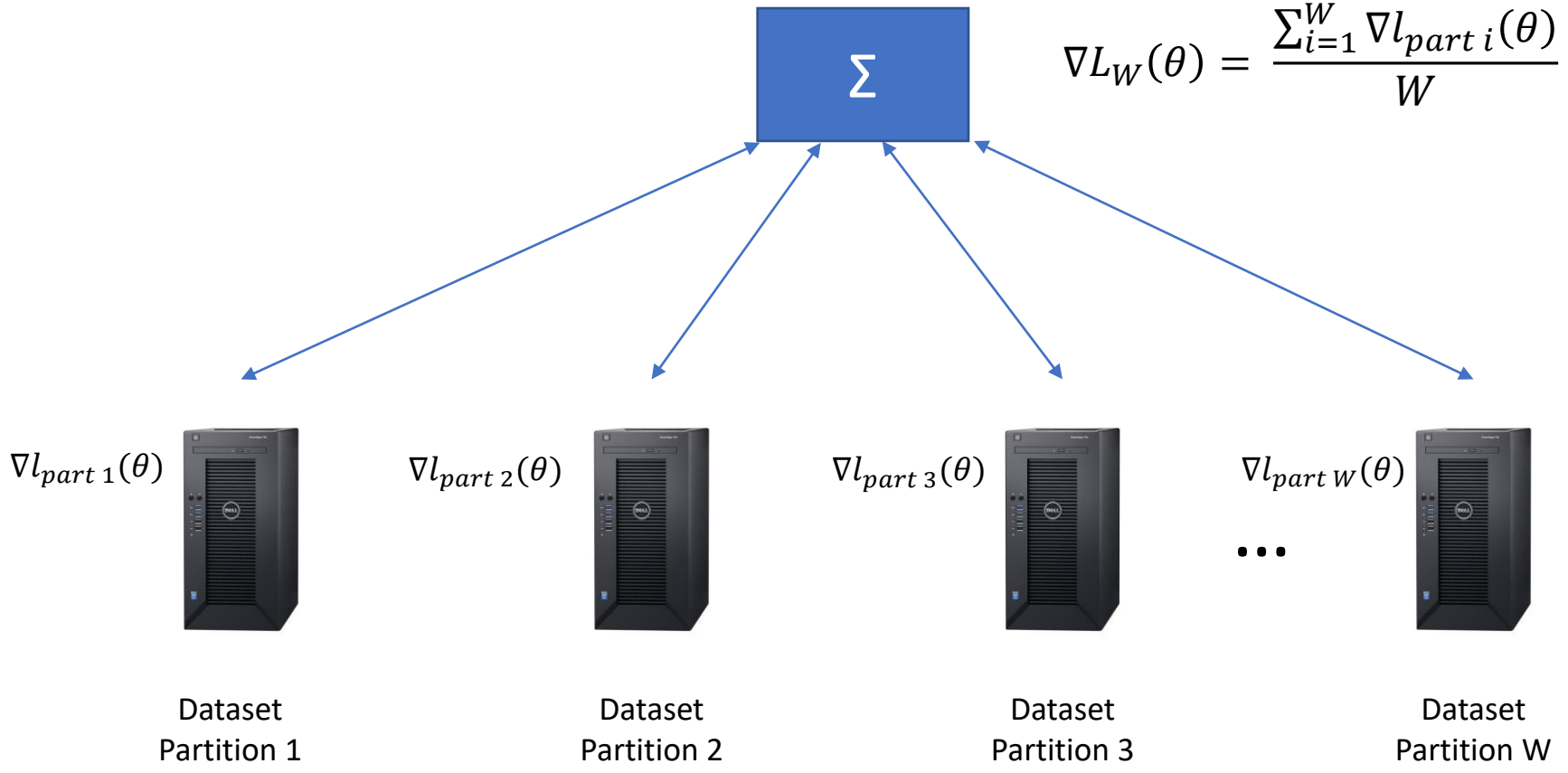
- W workers
- Each worker
  - Reads a partition of data
  - Computes a local gradient
  - Sends the gradient over to be aggregated
  - Model is updated and re-pushed to the workers
  - Rince and repeat

Distributed Gradient Descent 101

- In mini-batch SGD gradients will only be computed on a batch of data
  - Each worker will process a small batch of data

- In Full Gradient Descent it will computed on the whole dataset
  - Assuming n workers, each worker will process 1/n of the dataset

Distributed Gradient Descent 101

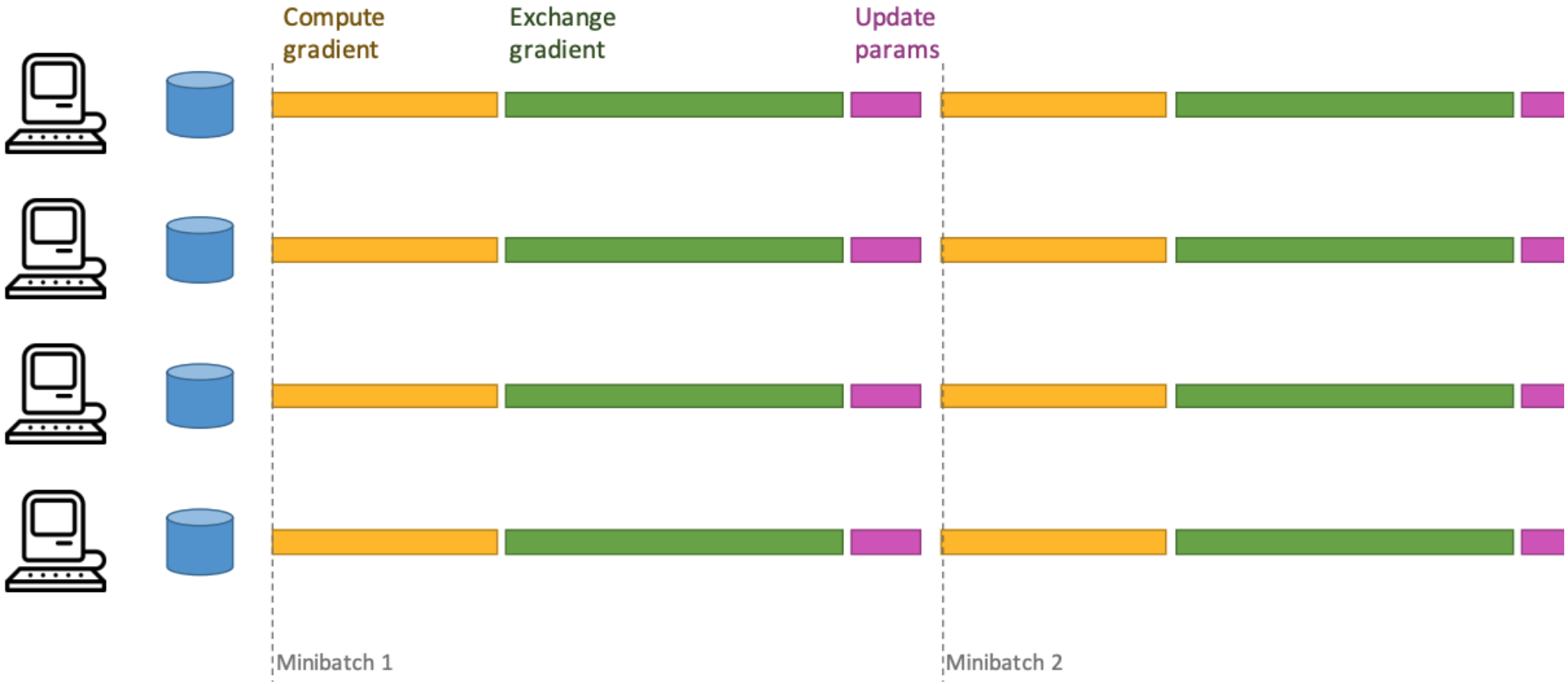


Distributed Gradient Descent 101

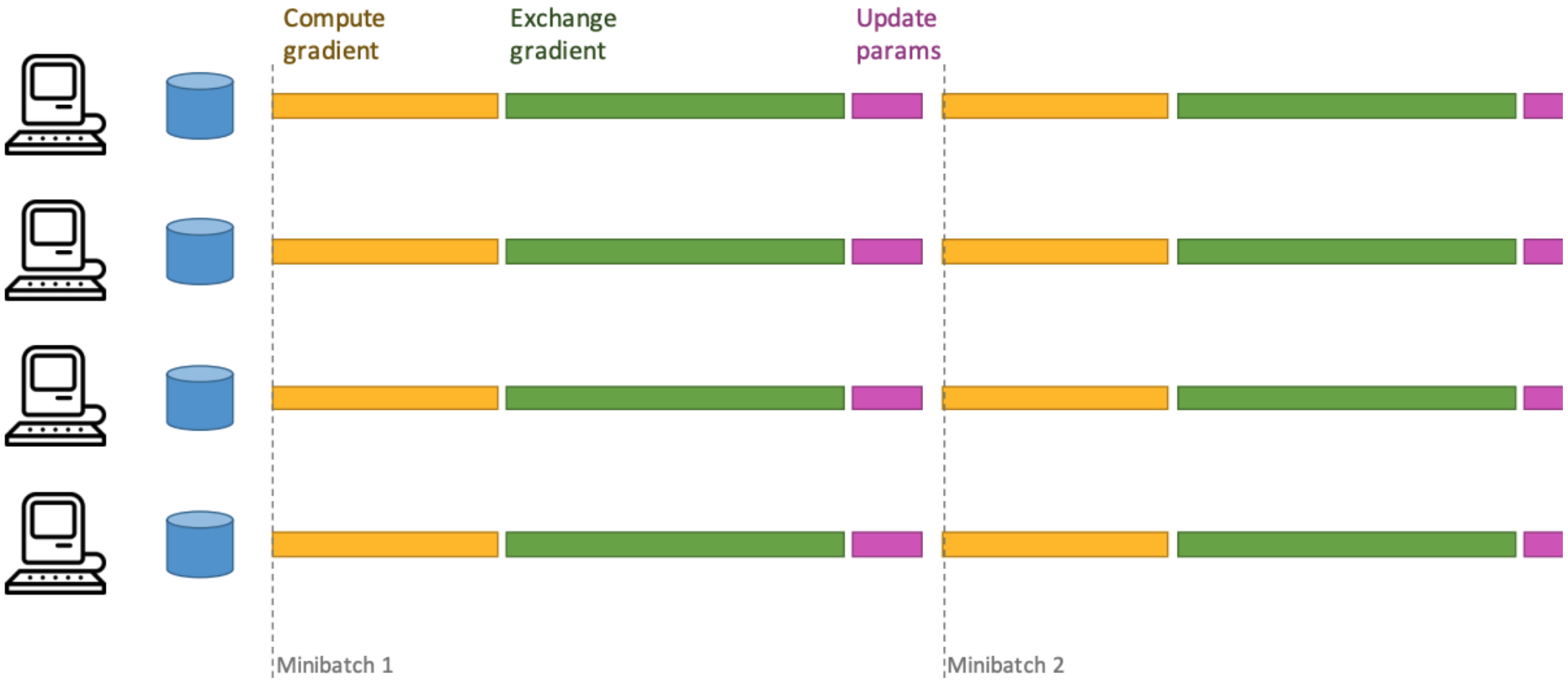
- Easy to implement a simple version in Spark

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

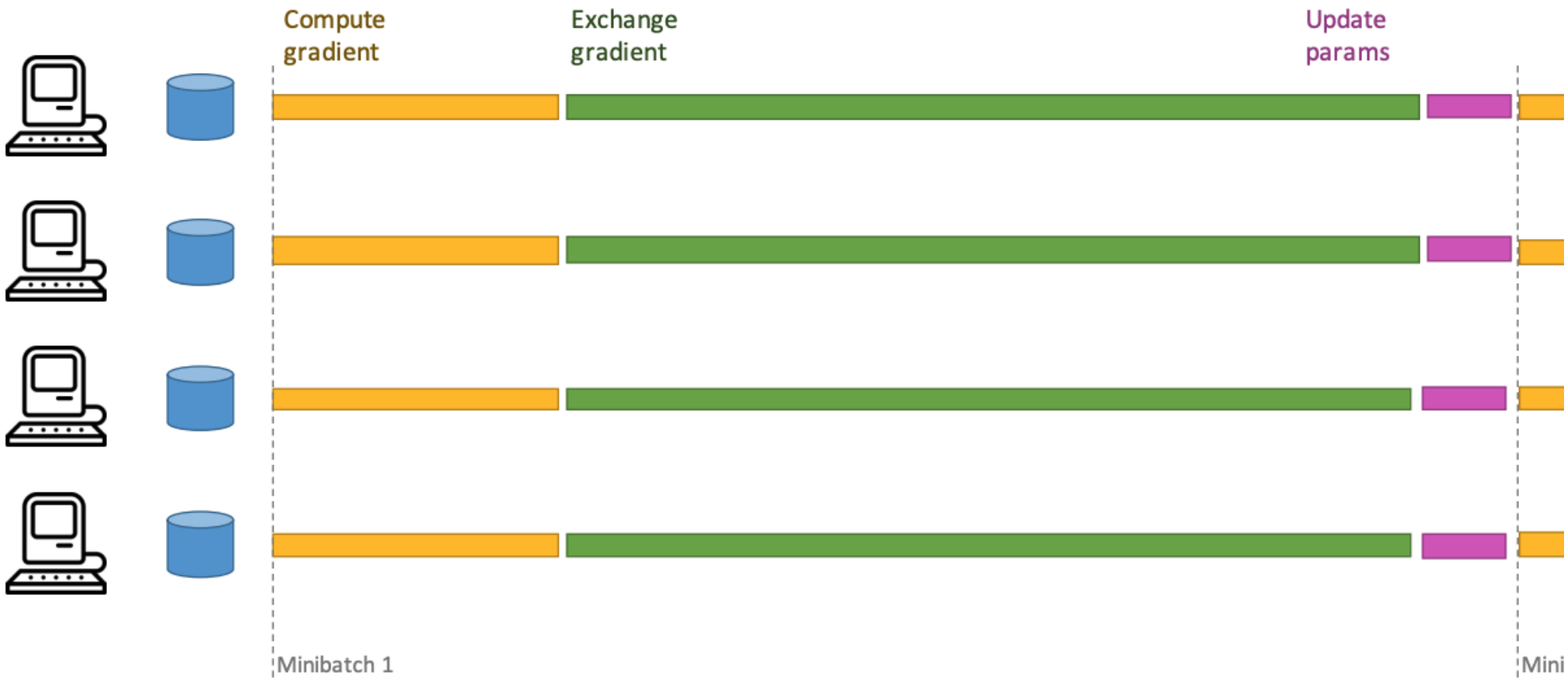
Synchronous Data Parallel SGD



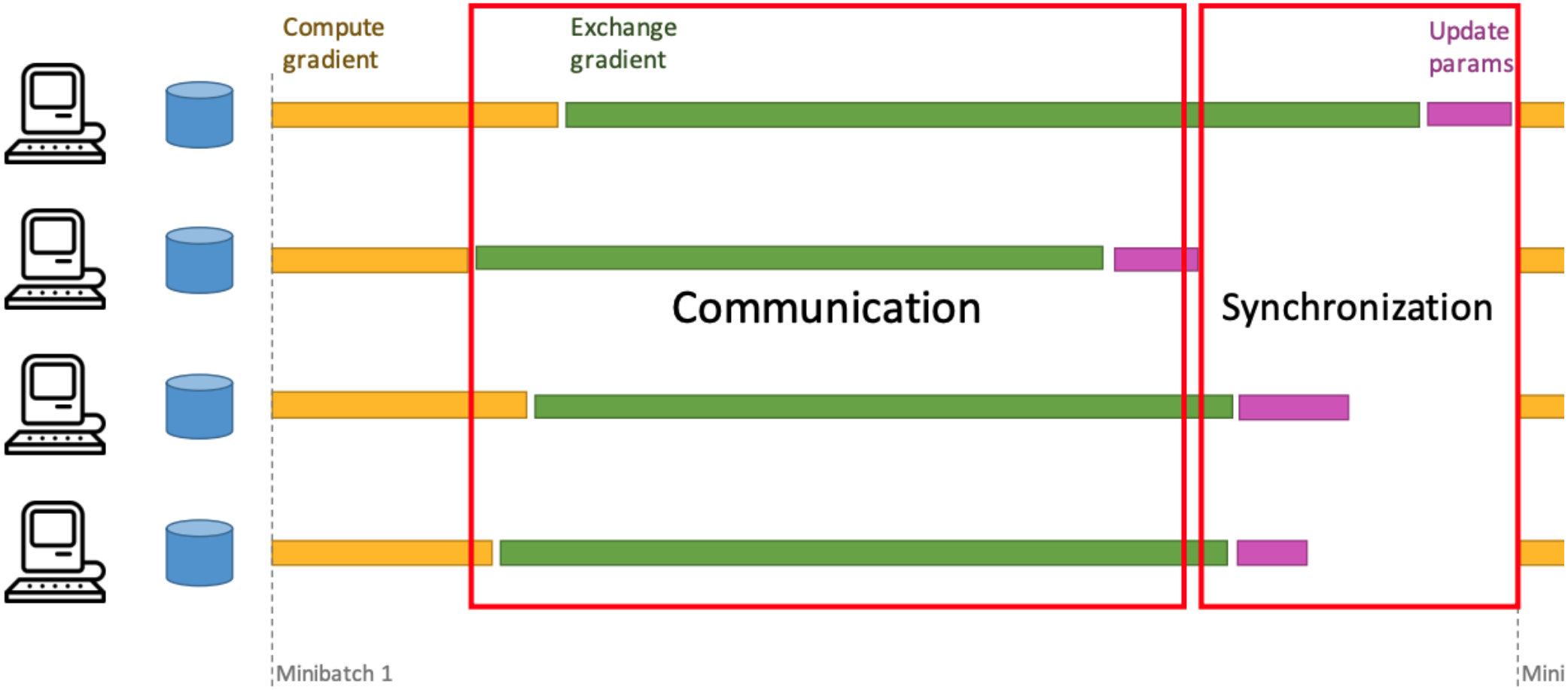
Synchronous Data Parallel SGD



Synchronous Data Parallel SGD



Synchronous Data Parallel SGD



Synchronous parallel SGD

- Objective: compute the gradient in parallel

- Two main overheads
  - Communication: sending gradient updates to the driver / other workers
  - Synchronization: waiting for all the workers to execute the current iteration

- This what is preventing from getting the theoretical speed up
  - With n workers
  - Gradient computation: T / n
  - Comm and sync cost: does not decrease with n but increase !

Communication cost

- Sparse model example
  - Total model weights M
  - But only f << M non zero features per row
  - For the log-loss defined earlier, gradient will be sparse and needs to be represented by a sparse vector

- This is the key to efficiency
  - Gradient update is sparse + sparse operation

Why is the gradient sparse

$$\nabla_\theta l(y, \hat{y}) = -\nabla_\theta [y \,.\, log\, f_\theta(x) + (1-y).\, log(1 - f_\theta(x))]$$

$$\nabla_\theta l(y, \hat{y}) = \left(y - \frac{1}{1 + e^{-x^T\theta}}\right) x$$

$$\nabla_\theta l(y, \hat{y}) = \kappa\, x$$

Sparse feature vector = sparse gradient

Some bad news

- What about regularization ?
  - L2 Regularization term $\frac{1}{2}\lambda\,\theta^T\theta$

$$\nabla_\theta \frac{1}{2}\lambda\,\theta^T\theta = \lambda\theta$$

  - The update rule becomes
    - $\theta \leftarrow \theta\ - \alpha\,(\kappa\,x + \lambda\theta)$
    - It's dense now ☹ no matter how sparse the feature vector is

Regularized SGD with Sparse Updates

- Let's concentrate on the first dimension of the model θ_0

- Suppose we have an update with the first dimension being zero

$$\theta_0 \leftarrow \theta_0 - \alpha\,(\kappa\, x_0 + \lambda\theta)$$
$$\theta_0 \leftarrow \theta_0 \; - \alpha\, \lambda\theta$$
$$\theta_0 \leftarrow (1 \; - \alpha\lambda)\theta_0$$

- After two updates

$$\theta_0 \leftarrow (1 \; - \alpha\lambda)^2\theta_0$$

Regularized SGD with Sparse Updates

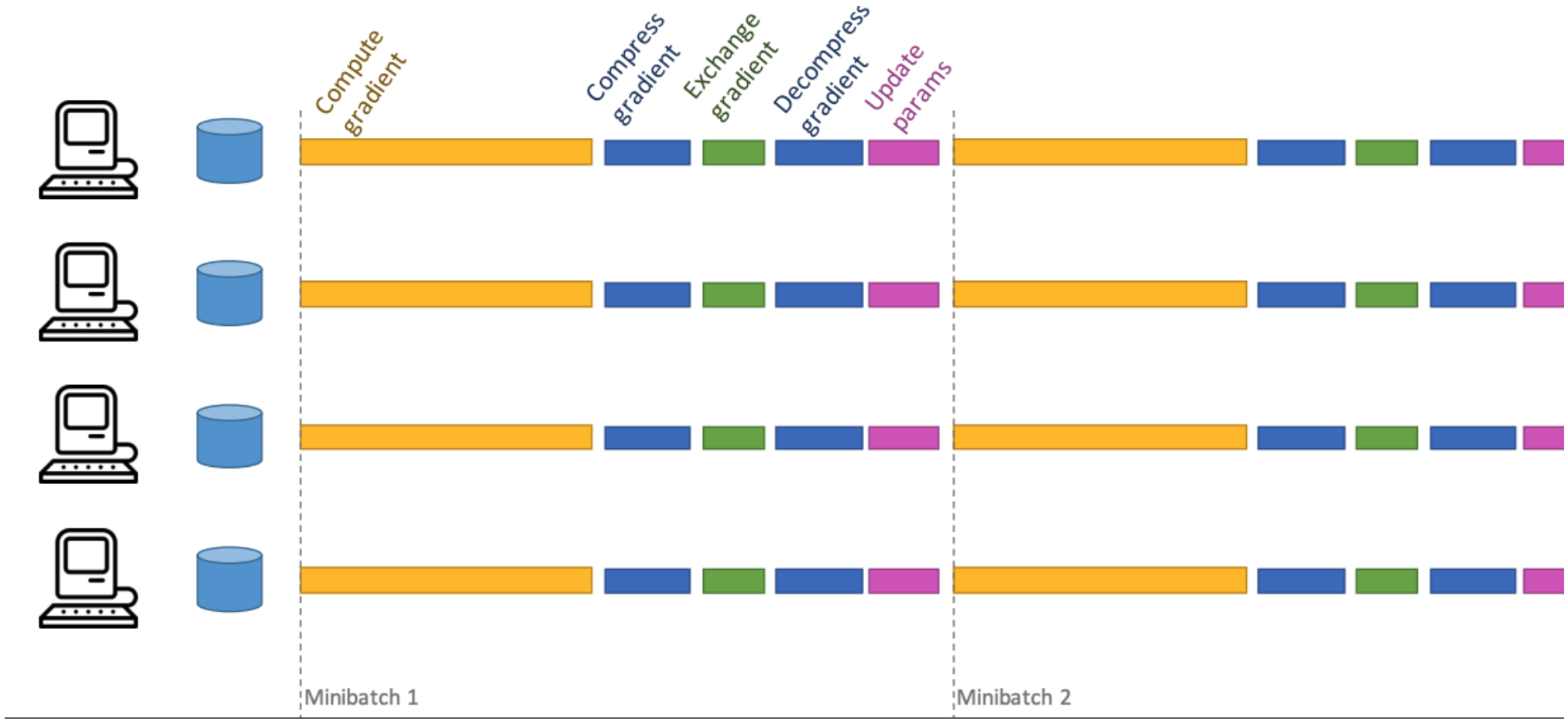
- We can then compress K computations

$$\theta_0 \leftarrow (1 - \alpha\lambda)^K \theta_0$$

- We remember how many steps we didn't update a particular dimension

- Catch up when needed

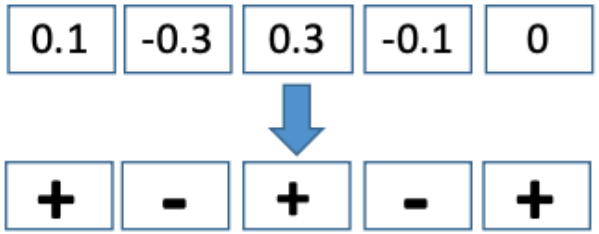
More general pattern for reducing communication cost



How can we compress a vector ?

- Lossy compression

- Quantization function
  - $\boldsymbol{v}$ vector of real values



$$avg_+ = +0.2$$

$$avg_- = -0.2$$

$$Q_i(v) = \begin{cases} avg_+ & \text{if } v_i \geq 0, \\ avg_- & \text{otherwise} \end{cases}$$

where $avg_+ = \text{mean}([v_i \text{ for } i: v_i \geq 0]), avg_- = \text{mean}([v_i \text{ for } i: v_i < 0])$

Why this shouldn't work

- Remember: Mini-batch gradient descent rely on

$$E\left(\nabla_\theta L_b(\theta)\right) = \nabla_\theta L(\theta)$$

- But

$$E(Q(\nabla_\theta L_b(\theta))) \neq E\left(\nabla_\theta L_b(\theta)\right)$$

- We don't have an unbiased estimate anymore

Nice trick: Stochastic Quantization

- Quantization function

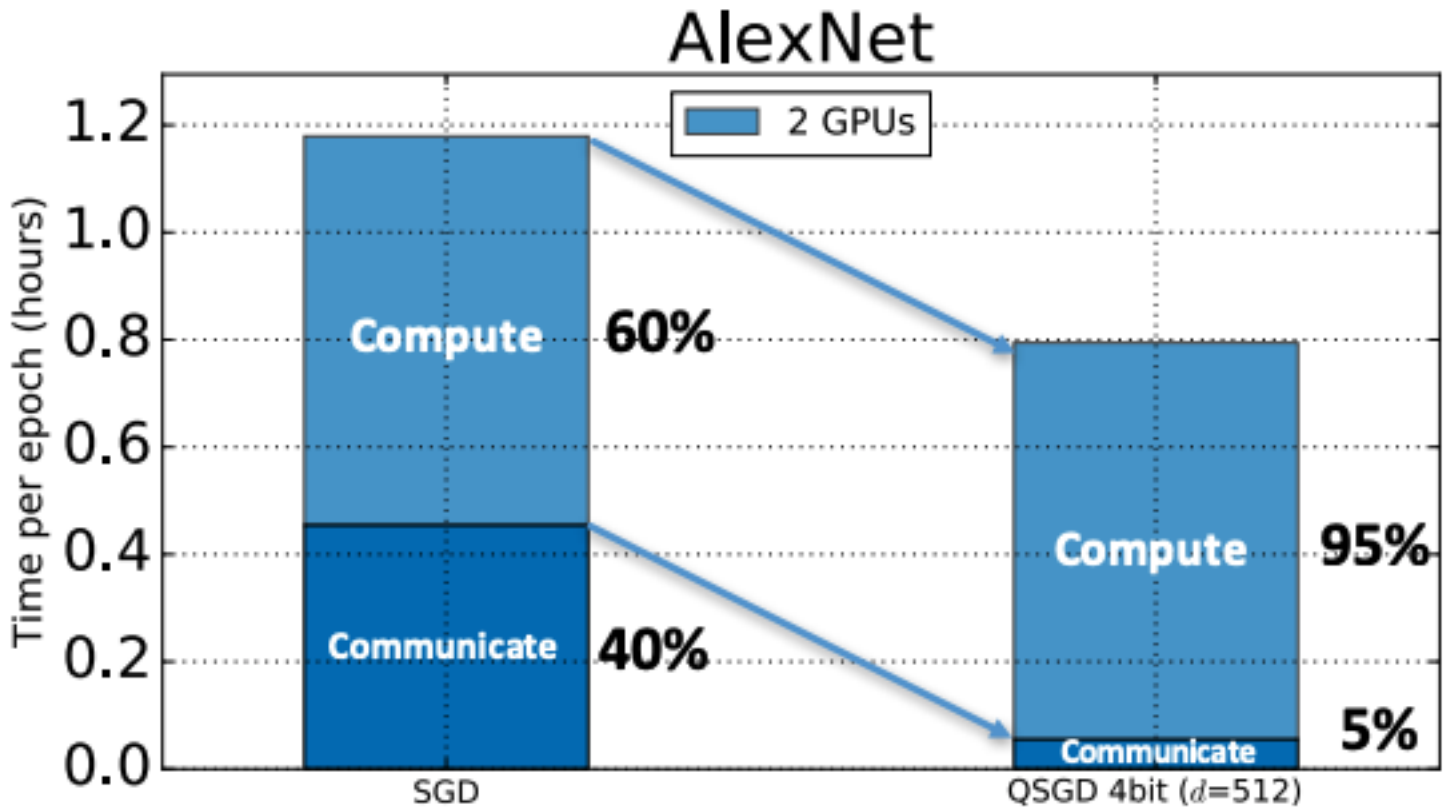
$$Q(vi) = \|v\|_2 \cdot \mathrm{sgn}(v_i) \cdot \xi_i(v_i)$$

where $\xi_i(v_i) = 1$ with probability $|v_i|/\|v\|_2$ and 0 otherwise.

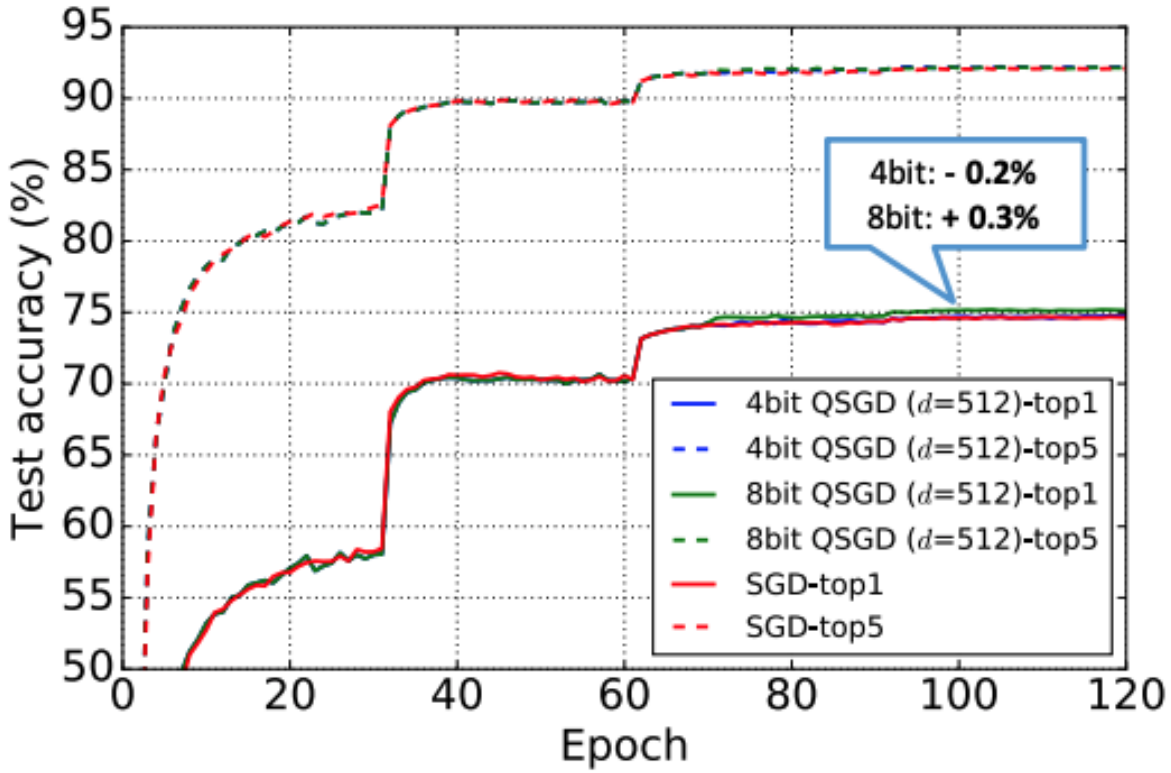
- Now it's unbiased

$$\boldsymbol{E}\left[\boldsymbol{Q}[\boldsymbol{v_i}]\right] = \|v\|_2 \cdot \mathrm{sgn}(v_i) \cdot |v_i|/\|v\|_2 = \mathrm{sgn}(v_i) \cdot |v_i|$$

Nice trick: Stochastic Quantization



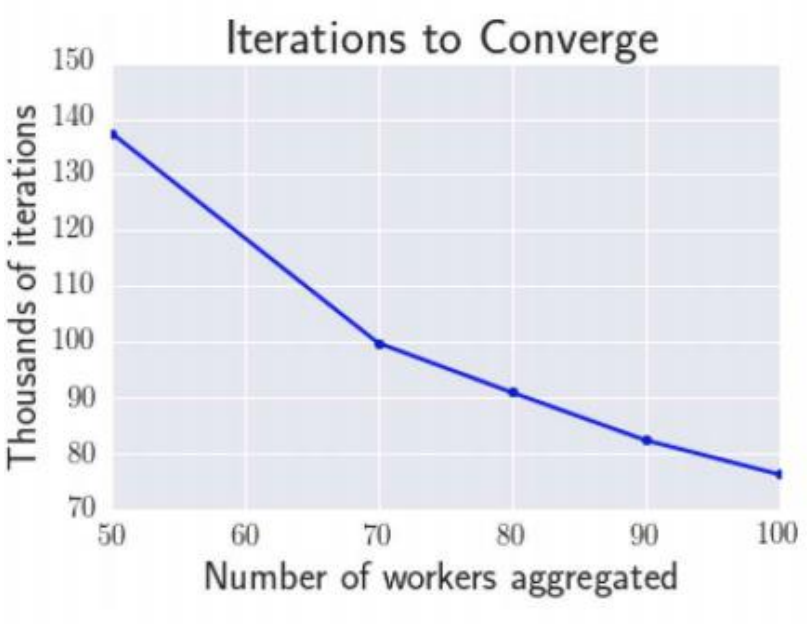
Nice trick: Stochastic Quantization

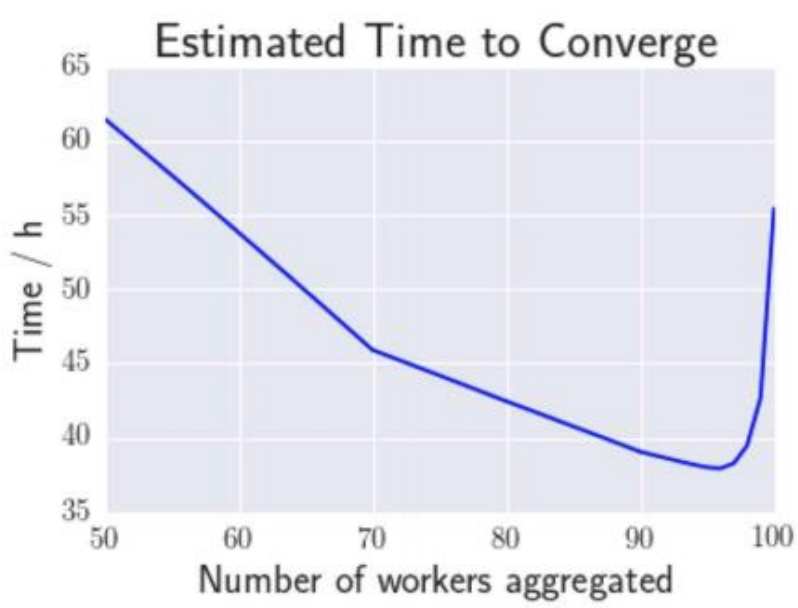


ResNet50 on ImageNet
8 GPU nodes

How about synchronization ?

- We will talk about it a bit later but here is simple solution
  - Drop the gradients of slow workers by imposing a timeout to the computation





Synchronous Distributed SGD

- The name of the game is reducing communication cost
  - Exploiting sparsity
  - Compression
  - Increasing computation
- Works reasonably well in practice
  - Can also give a good initial solution to be fine tuned with more complex methods

In Spark

```
// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")

// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache() val test = splits(1)

// Run training algorithm to build the model
val model = new LogisticRegressionWithSGD()
                        .setNumClasses(10)
                        .run(training)

// Compute raw scores on the test set.
val predictionAndLabels = test.map { case LabeledPoint(label, features) =>
      val prediction = model.predict(features)
      (prediction, label)
}
```

Other ways to distribute

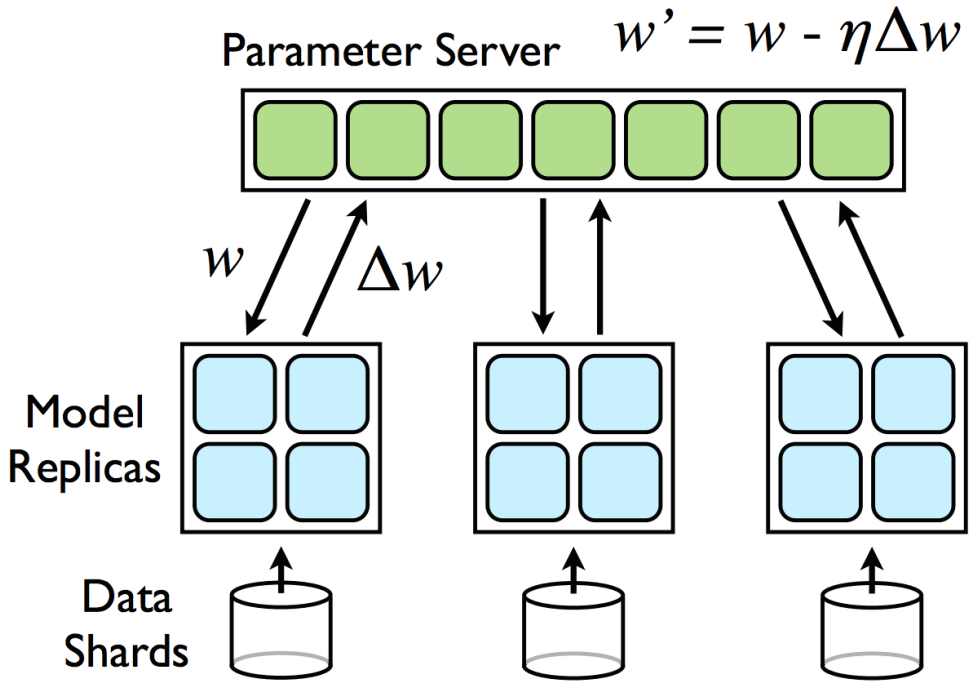
- Two major costs: Communication and synchronization
- We talked about reducing communication cost in different ways while staying in a synchronous centralized manner
- We will discuss two additional ways
  - Asynchronous updates: Parameter Server
  - Efficient Aggregation: All Reduce

Parameter Server design rationale

- Model updates to be *generally* in the right direction
  - Not important to have strong consistency guarantees all the time
- Model updates are often sparse
  - No need to pull all model parameters and push all of them
  - Separate computation and storage

Parameter Server design rationale

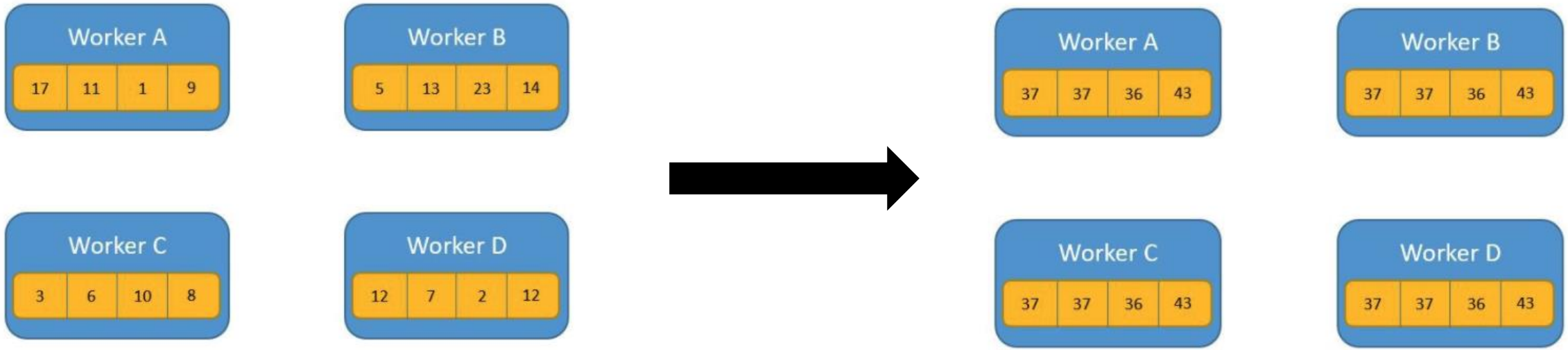
- Lock free asynchronous updates
  - HOGWILD!
- Introducing asynchrony makes the system fault tolerant
  - If a worker fails while computing gradient, no need to wait, just restart the computation



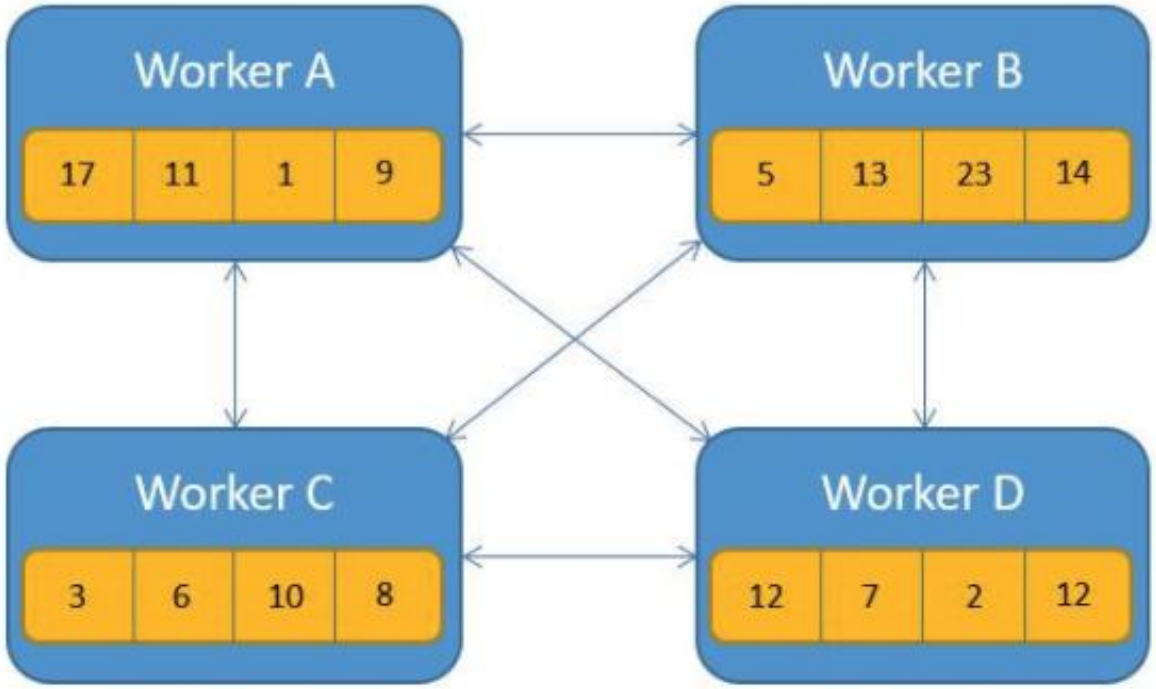
AllReduce

- AllReduce
  - Average all the values in all the computation nodes and send the averages value to all the nodes
  - Example: averaging gradients !
- Implementation
  - Naive: Reduce and then Broadcast like the first Spark snippet
  - Optimized: do both at the same time

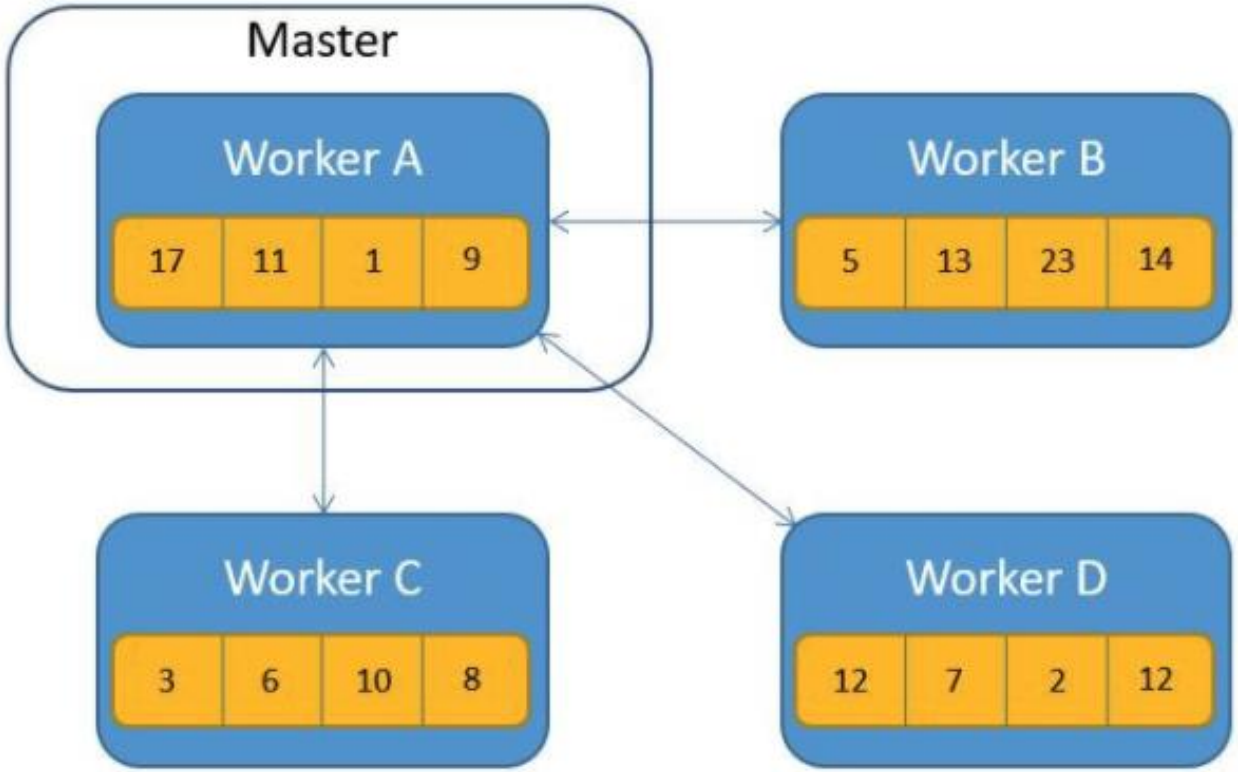
AllReduce


Worker A
17 11 1 9
Worker B
5 13 23 14
Worker C
3 6 10 8
Worker D
12 7 2 12
Worker A
37 37 36 43
Worker B
37 37 36 43
Worker C
37 37 36 43
Worker D
37 37 36 43

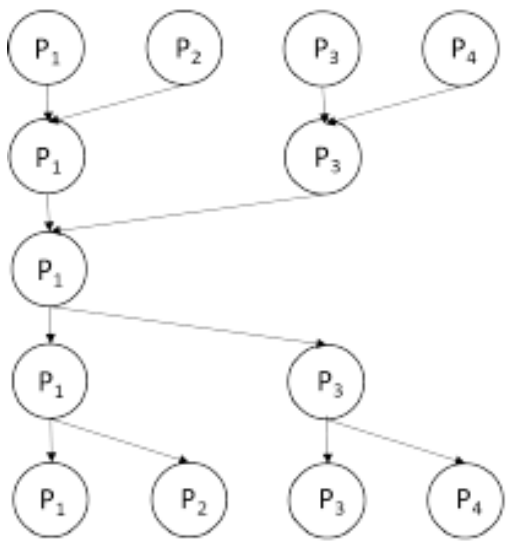
AllReduce: all to all communication



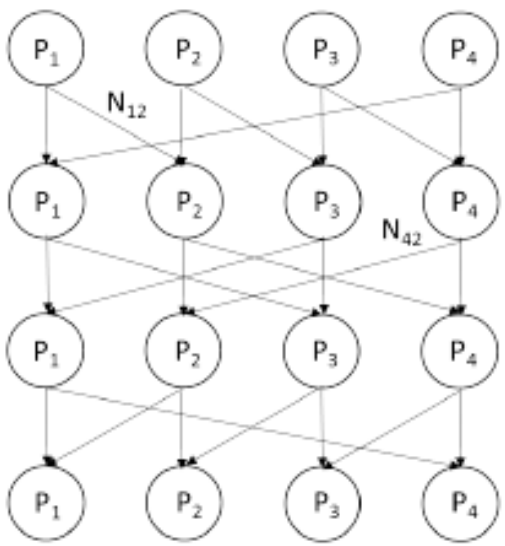
AllReduce: Reduce then Broadcast



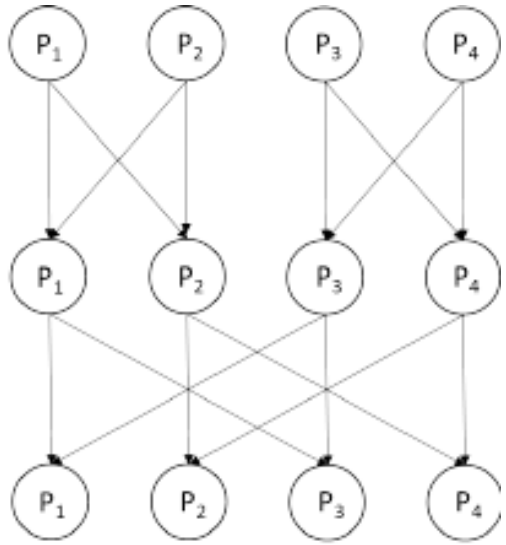
AllReduce



(a) Tree AllReduce

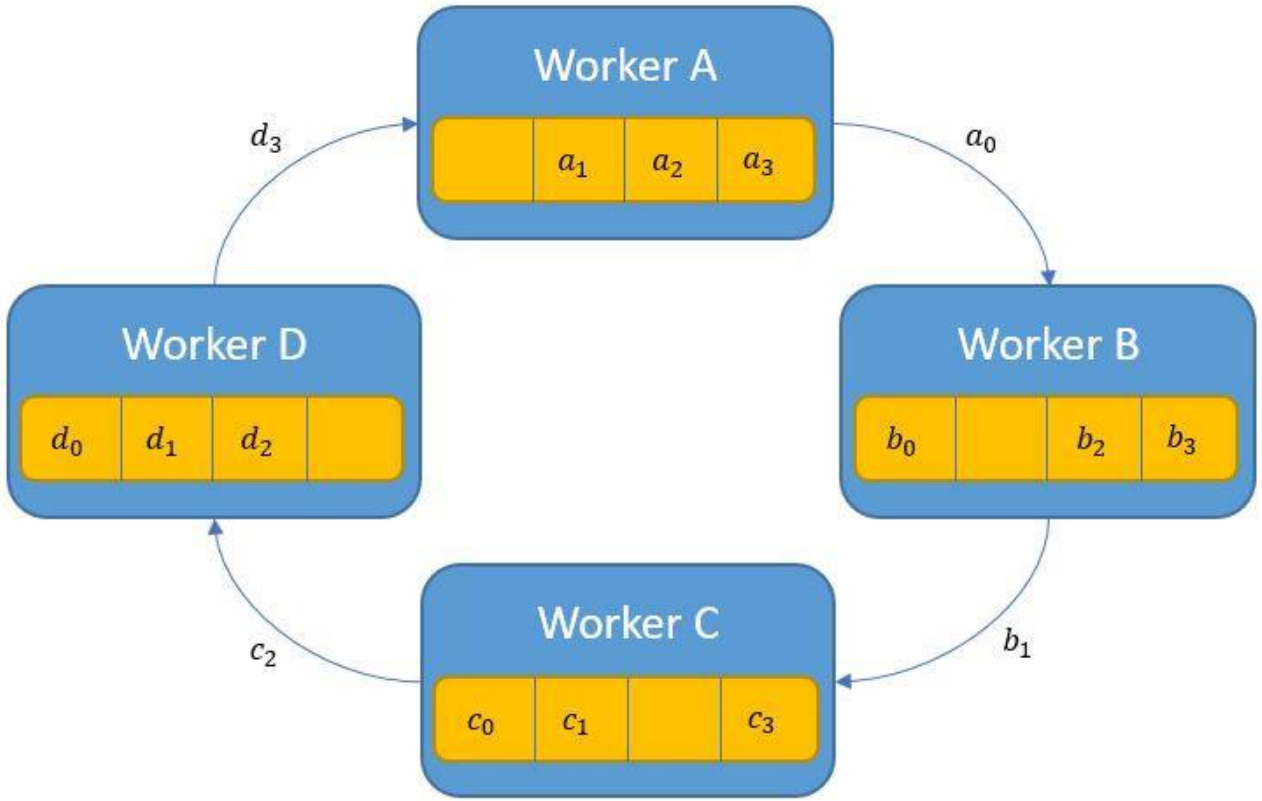


(b) Round-robin AllReduce

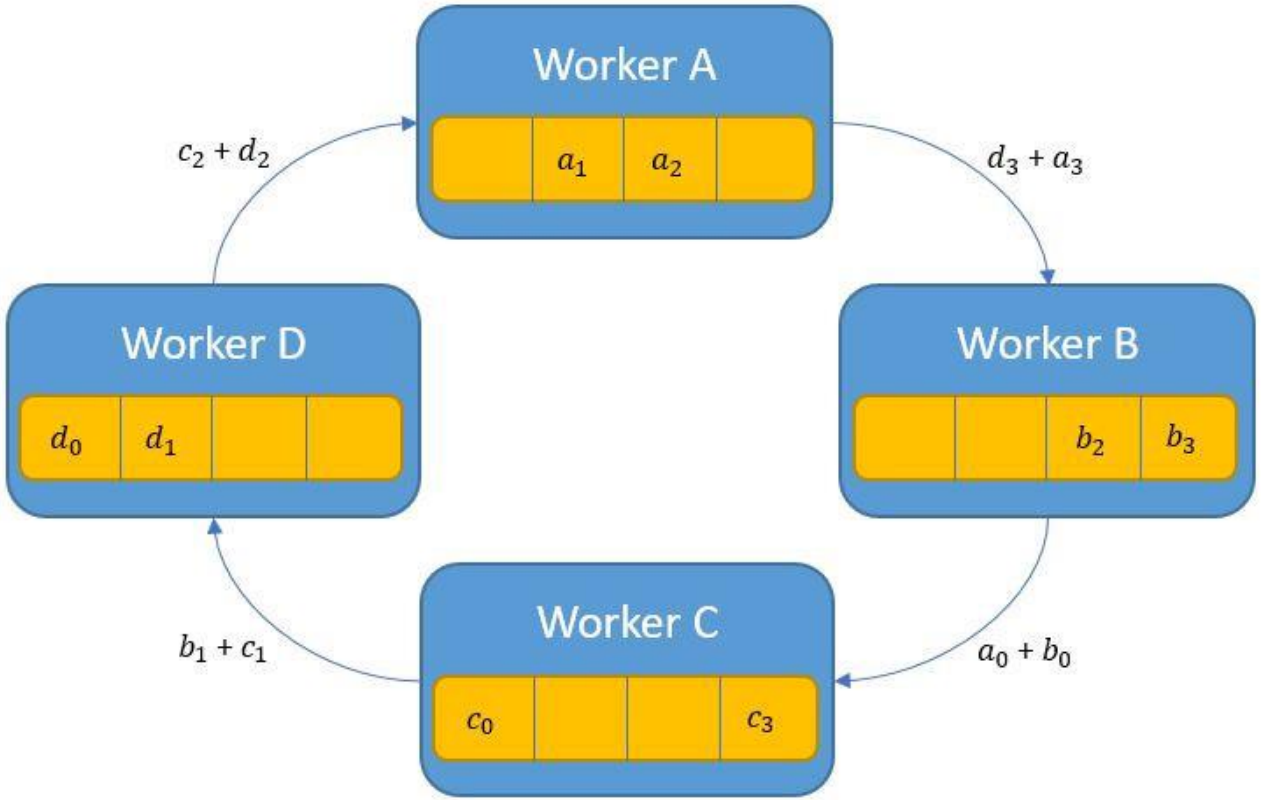


(c) Butterfly AllReduce

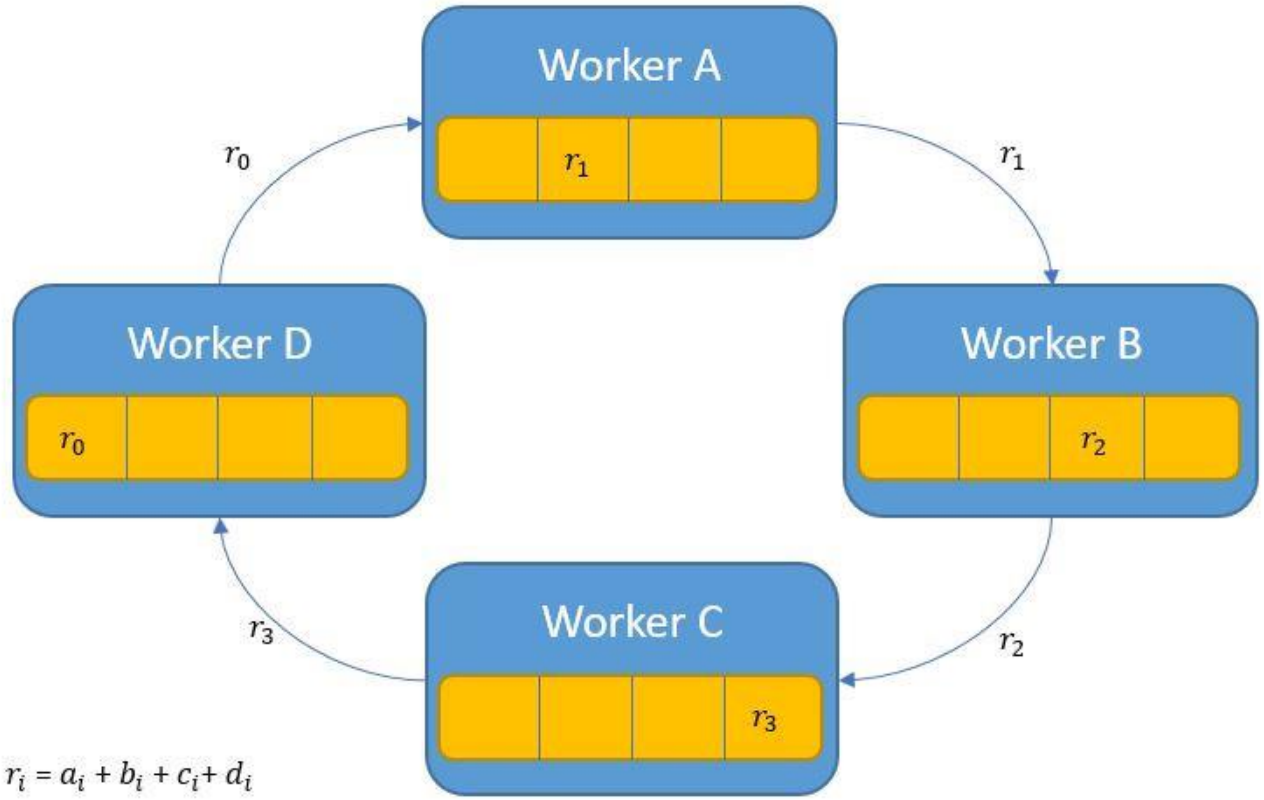
Ring AllReduce



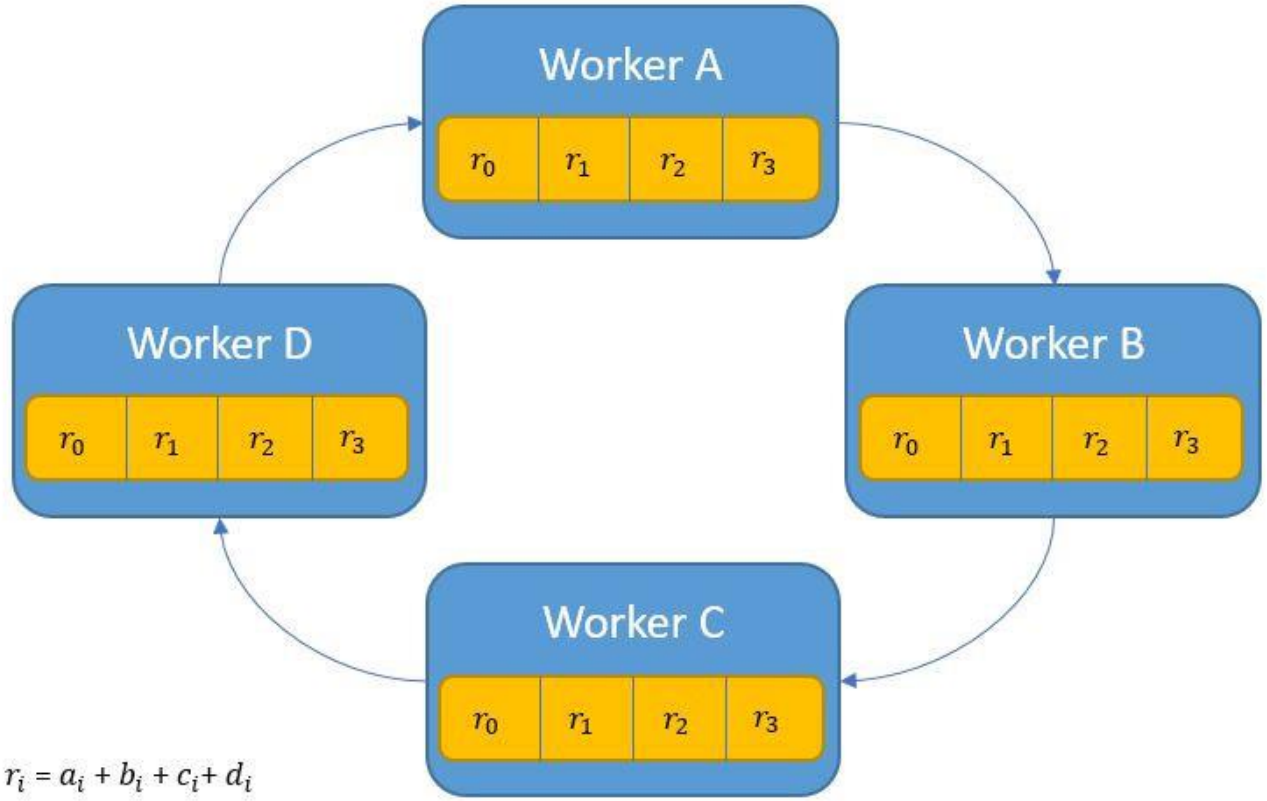
Ring AllReduce



Ring AllReduce



Ring AllReduce



$r_i = a_i + b_i + c_i + d_i$

How to speedup gradient descent?

- Up to now: faster gradient computation
  - Distributed
  - Optimize communication cost
  - Reduce synchronization overhead

- Everything we said is widely applicable
  - You just need a differentiable model
  - Model parameters fit on one machine
  - Loss is a sum of pointwise losses over training examples

- Another way: smarter optimization algorithms
  - 2^{nd} order methods
  - Line search

Line search

- Heavily used with Full Batch Gradient Descent
  - Repeat until convergence
    - Compute descent direction $\nabla L(\theta)$
    - Choose α_t to « loosely » minimize $L(\theta - \alpha_t \nabla L(\theta))$

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla L(\theta)$$

Second order optimization methods

- We want to speed up our convergence in another way
- Using the “curvature” information can help a lot
- Using second order Taylor expansion

$$L(\theta + \Delta\theta) = L(\theta) + \Delta\theta^T \nabla L(\theta) + \frac{1}{2}\Delta\theta^T \left(\nabla^2 L(\theta)\right)\Delta\theta$$

Newton's method

- Let's rewrite using $\theta_{n+1} = \theta_n + \Delta\theta$ and take the gradient of the expansion (with respect to $\Delta\theta$)

- And let's denote $\nabla^2 L(\theta_n) = H_n$ and $\nabla L(\theta_n) = \mathbf{g_n}$

$$\nabla L(\theta_{n+1}) = g_n + H_n \Delta\theta$$

- We want to locally minimize L, so by taking the above gradient to 0, we obtain

$$\Delta\theta = -H_n^{-1} g_n$$

Quasi-newton methods

- Two main steps to newton iteration
  - Compute $\nabla^2 L(\theta_n) = H_n$
  - Compute $\Delta\theta = -H_n^{-1} g_n$

- Both of these steps can be very expensive

- Quasi-newton methods approximate H_n^{-1} by some matrix B_n and updates it appropriately at each step

Quasi-newton methods

- Until convergence
  - Compute update direction

$$\Delta\theta = -B_n g_n$$

  - Line search for learning rate

$$\alpha \leftarrow \min_{\alpha \geq 0} L(\theta_n - \alpha\Delta\theta)$$

  - Update parameters

$$\theta_{n+1} \leftarrow \theta_n - \alpha\Delta\theta$$

  - Store the parameters and gradient deltas

$$g_{n+1} \leftarrow \nabla L(\theta_{n+1})$$
$$s_{n+1} \leftarrow x_{n+1} - x_n$$
$$y_{n+1} \leftarrow g_{n+1} - g_n$$

  - Update inverse hessian approximation

$$B_{n+1} \leftarrow QuasiUpdate(B_n, s_{n+1}, y_{n+1})$$

BFGS

- Update B with a rank two matrix, of the form

$$B_{n+1} \leftarrow B_n + auu^T + bvv^T$$

- Limited memory BFGS or L-BFGS for short
  - Perform the update without actually materializing B matrix and performing an explicit matrix vector multiplication
  - Can be achieved by storing the latest few values and gradients

Why are we talking about this ?

- We can now perform second order updates just with the (full) gradient
- Gradient computation is embarassingly parallel
- L-BFGS works very well in practice and converges in very few epochs (so we increase the computation to reduce the communication overhead)

Conclusion

- Distributed Machine Learning is about trade-offs
  - Communication VS computation cost
- For simple models (like Logistic Regression), a synchronous approach works well
  - Exploit sparsity
  - Use more complex optimization schemes
- There are several ways to distribute and aggregate computation
  - Centralized synchronous or asynchronous model, AllReduce …