Large Scale Machine Learning

Part 2: Distributed Logistic Regression

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

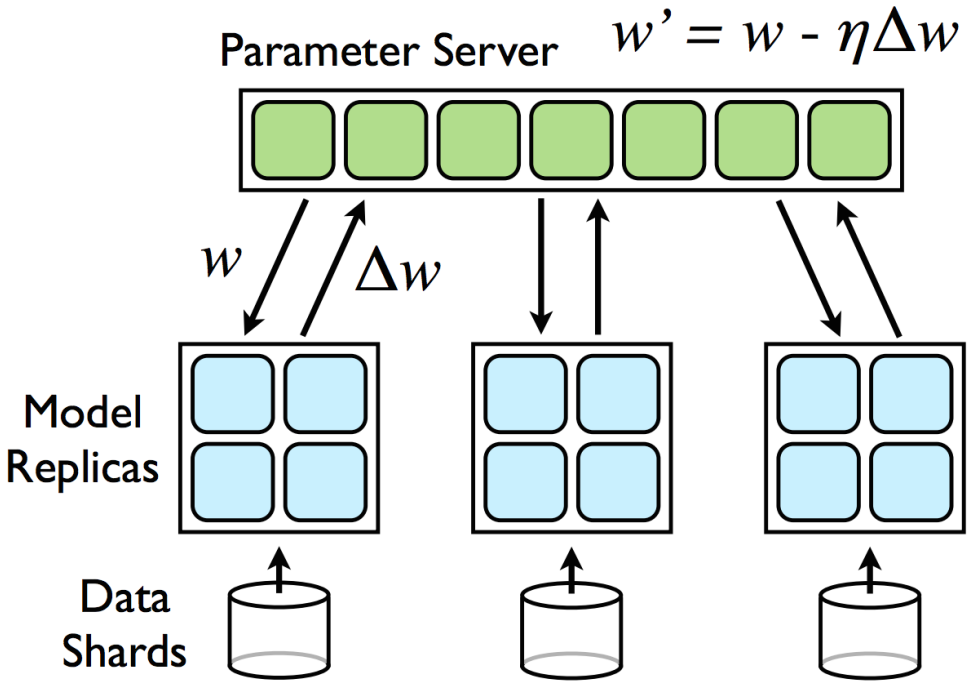
- Two major costs: Communication and synchronization
- We talked about reducing communication cost in different ways while staying in a synchronous centralized manner
- We will discuss two additional ways
  - Asynchronous updates: Parameter Server
  - Efficient Aggregation: All Reduce

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

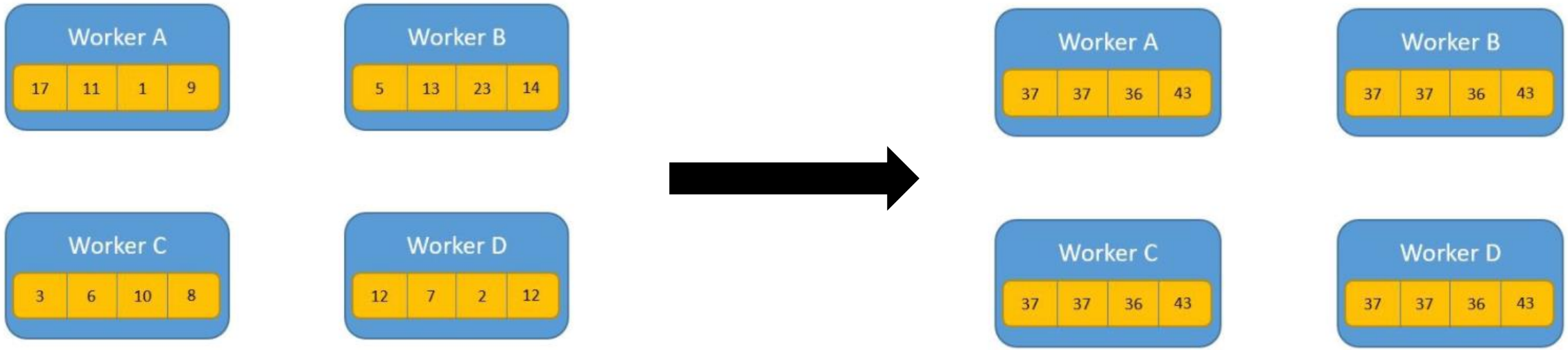
- Lock free asynchronous updates
  - HOGWILD!
- Introducing asynchrony makes the system fault tolerant
  - If a worker fails while computing gradient, no need to wait, just restart the computation



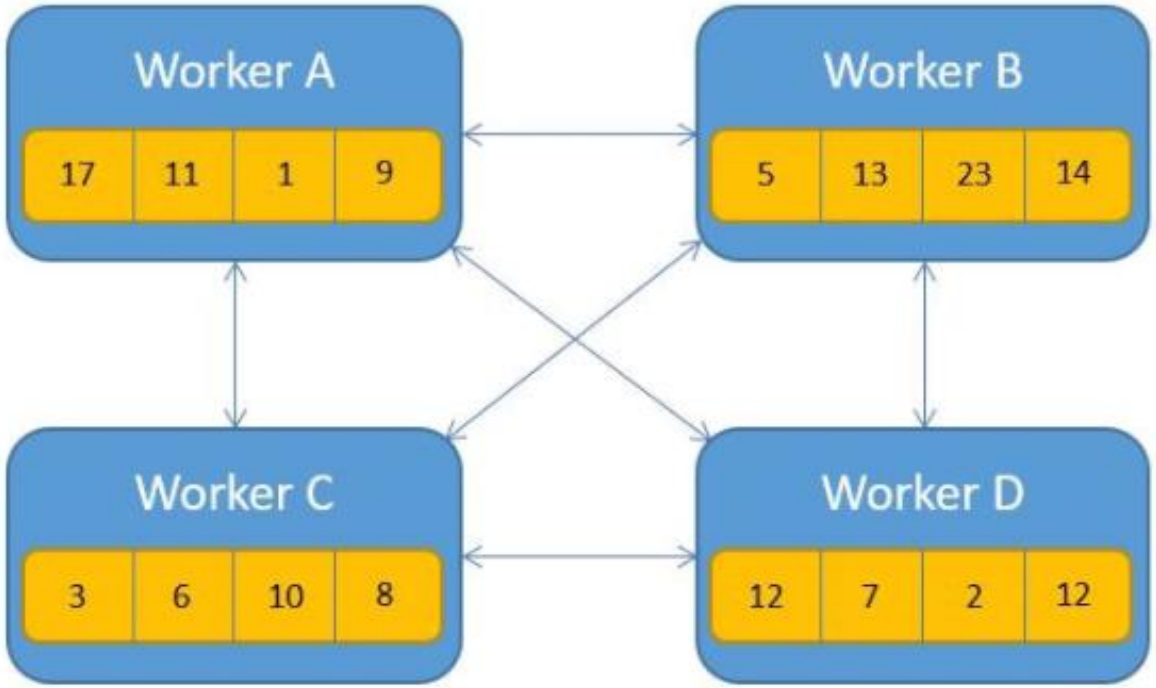
AllReduce

- 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

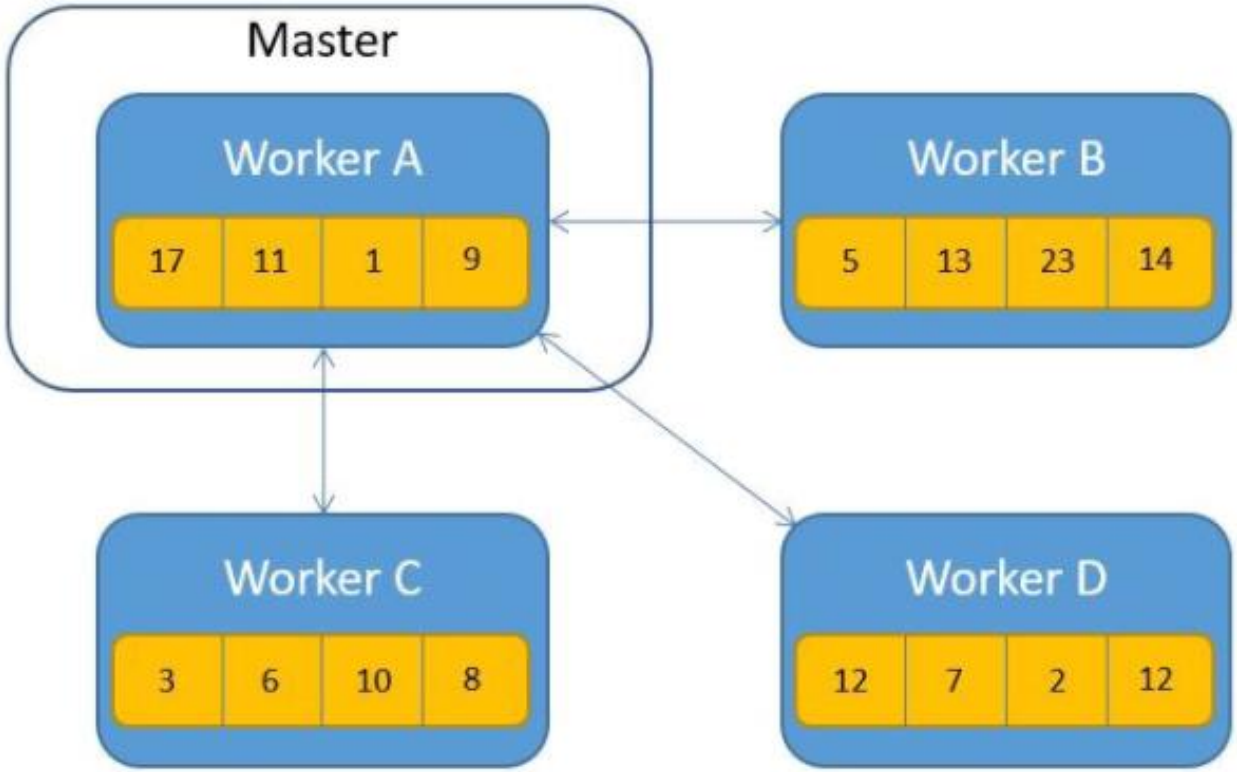
AllReduce


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

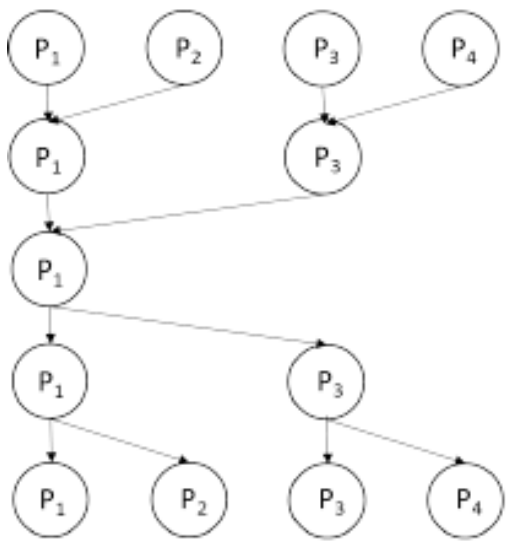
AllReduce: all to all communication



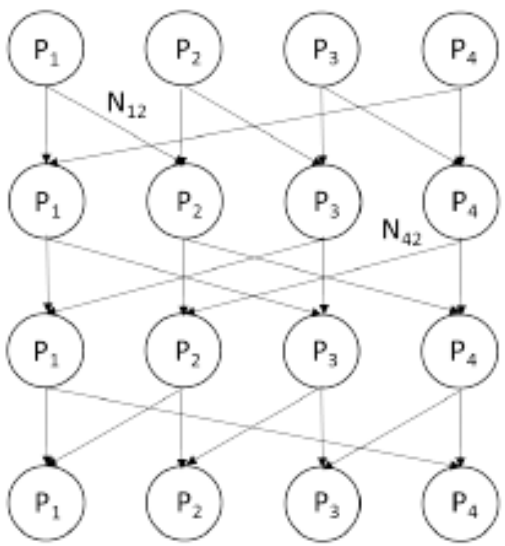
AllReduce: Reduce then Broadcast



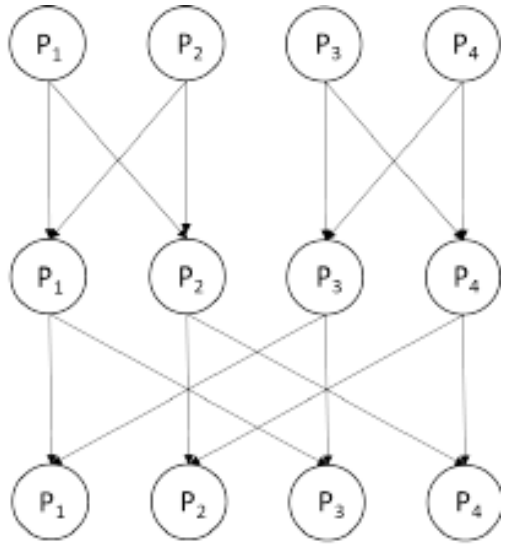
AllReduce



(a) Tree AllReduce

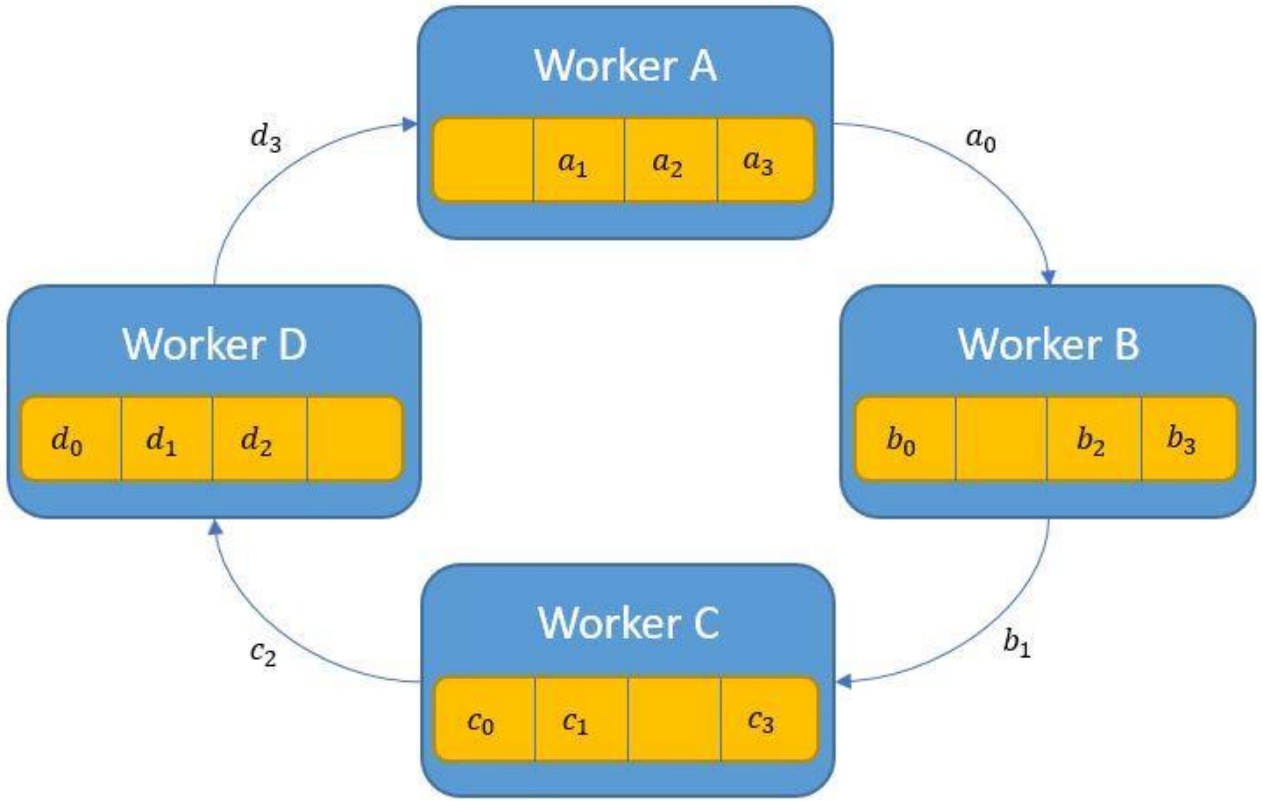


(b) Round-robin AllReduce

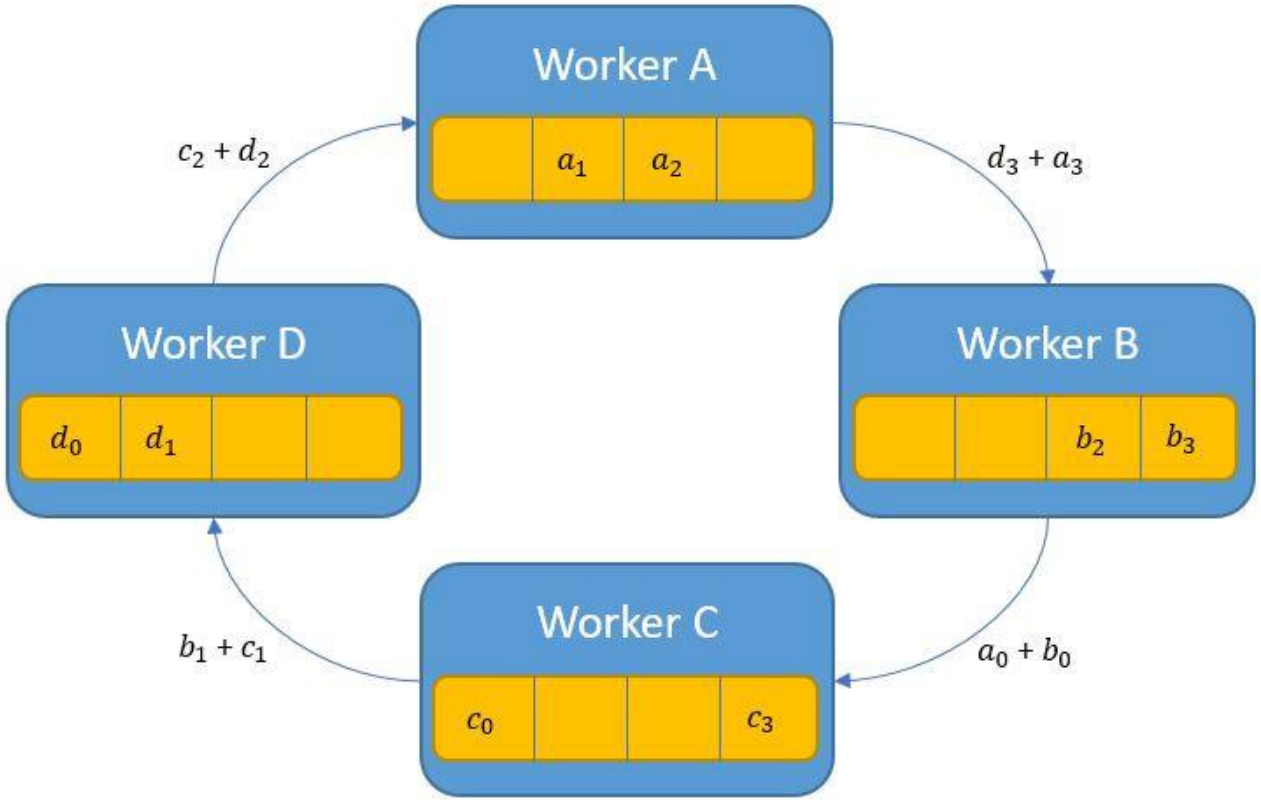


(c) Butterfly AllReduce

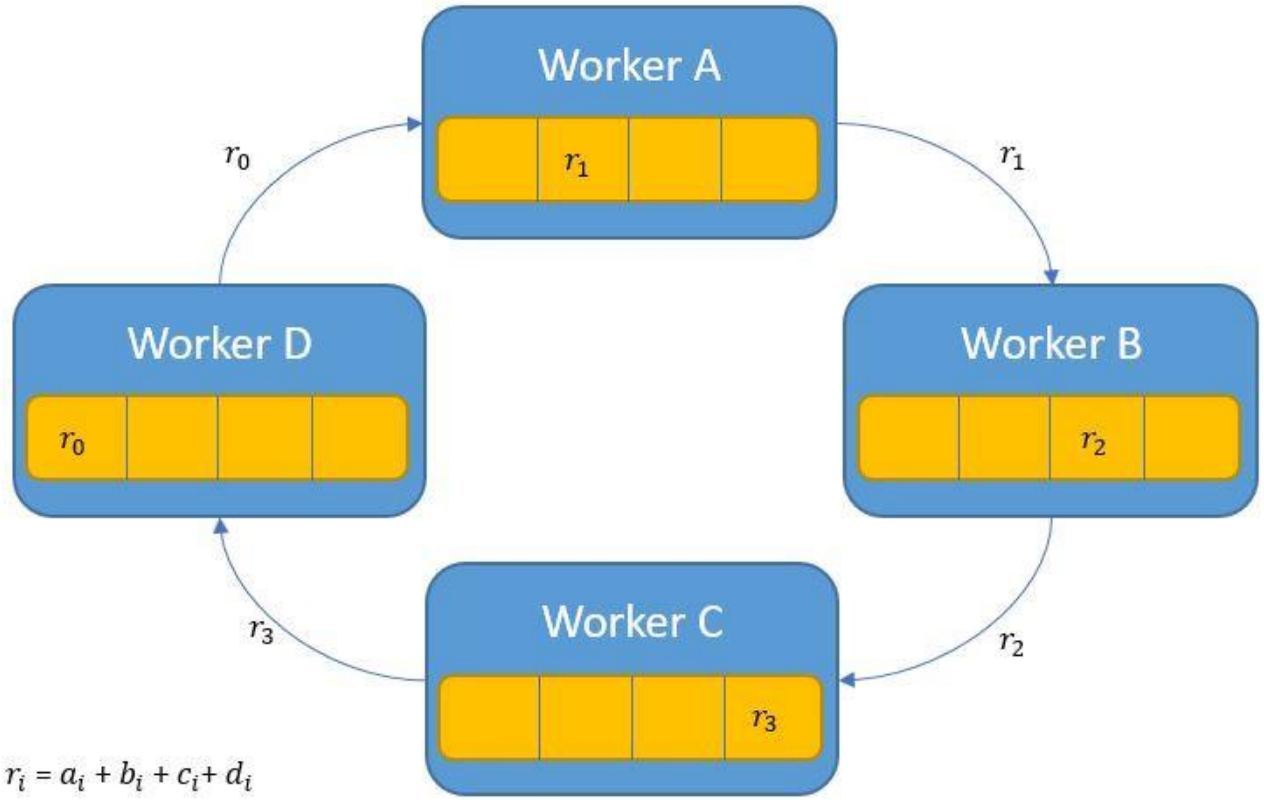
Ring AllReduce



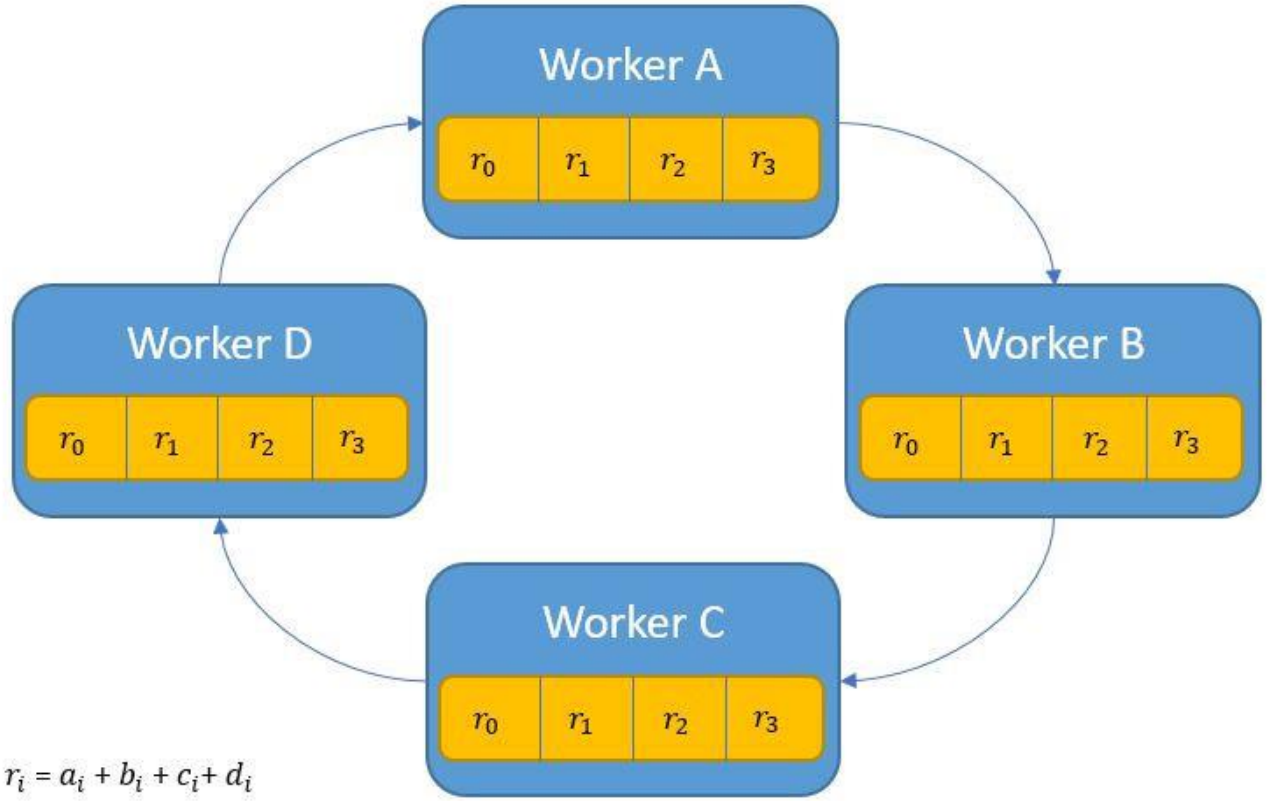
Ring AllReduce



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 …