Large Scale Machine Learning

Distributed learning - Part 2

Previous session: Synchronous Distributed SGD

- Trade-off between
  - Compute time
  - Communication time
  - Synchronisation time
- Reducing communication cost
  - Exploiting sparsity
  - Compression
  - But Increasing computation
- Works reasonably well in practice
  - Can also give a good initial solution to be fine tuned with more complex methods

In Spark

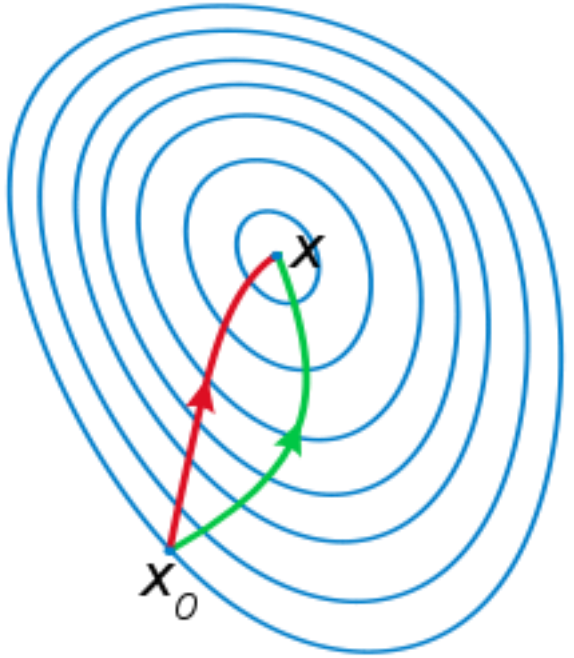
```
// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")

// 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)
}
```

This session

- Decreasing required number of iterations
  - Second order methods, Newton / LBFGS
  - ADAM
- Decreasing communication cost
  - Tree reduce, ring reduce
- Decreasing synchronization cost
  - Asynchronous updates!
- Beyond data parallelism

Second order optimization methods

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



Newton's method (1 dimension)

- In one dimension:

Quadratic approximation of function L to minimize:

$$L(x) = L(x_t) + (x - x_t)\, L'(x_t) + (x - x_t)^2\, L''(x_t)/2 + o((x - x_t)^2)$$

Step minimizing local approximation : $x_{t+1} := x_t - (L''(x_t))^{-1} \cdot L'(x_t)$

- Very good convergence properties

Newton's method

- Taylor expansion of L:

$$L(x) = L(x_t) + (x - x_t) \cdot \nabla L(x_t) + (x - x_t) \cdot \nabla^2 L \cdot (x - x_t) + o((x - x_t)^2)$$

Step minimizing local approximation :

$$x_{t+1} := x_t - (\nabla^2 L(x_t))^{-1} \cdot \nabla L(x_t)$$

Hessian matrix

- Coefficients $\frac{\partial^2}{\partial x_i \partial x_j} L(x)$
- Size n^2

- Very good convergence properties on convex problems

- Hessian is of size n^2

Quasi-newton methods

- Instead of: $\mathrm{dx} = \; -(\nabla^2 \mathrm{L}\,(\mathrm{x_t}))^{-1} \,.\, \nabla L(\mathrm{x_t})$

- Approximate $(\nabla^2 \mathrm{L}\,(\mathrm{x}))^{-1}$ by some matrix B_t
- $\mathrm{dx} = \; - \alpha \; B_t \; \nabla L(\mathrm{x_0})$

Select step size α with Line search

- Several methods, usually estimating B_t from previous gradients and steps
- **L-BFGS**

Quasi-newton methods : generic

- Until convergence
  - Compute update direction ($B_n \approx H_n^{-1}$)

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

  - 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

$$g_{n+1} \leftarrow \nabla L(\theta_{n+1})$$
$$\Delta\theta_{n+1} \leftarrow \theta_{n+1} - \theta_n$$
$$\Delta g_{n+1} \leftarrow g_{n+1} - g_n$$

  - Update inverse hessian approximation

$$B_{n+1} \leftarrow QuasiUpdate(B_n, \Delta\theta_{n+1}, \Delta g_{n+1})$$

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

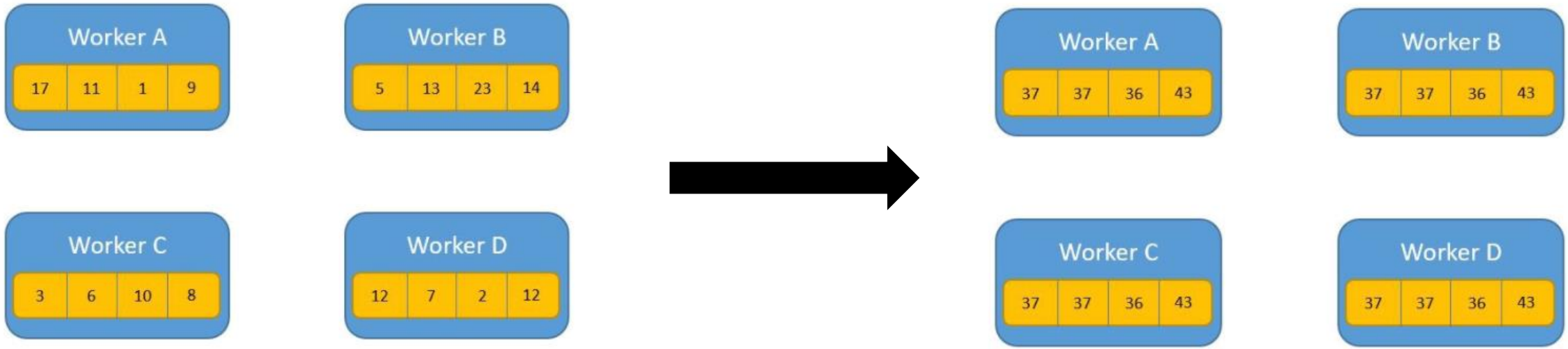
ADAM

```
        alpha = 0.01  # Stepsize
        beta1 = 0.9
        beta2 = 0.999
        epsilon = 1e-8
        epsilon = epsilon
        m = v = t = 0

    def AdamStep(gradient):
        m = beta1 * m + (1.0 - beta1) * gradient
        v = beta2 * v + (1.0 - beta2) * gradient * gradient
        mhat = m / (1.0 - beta1**(t+1))
        vhat = v / (1.0 - beta2**(t+1))
        step = alpha * mhat / (np.sqrt(vhat) + 1e-8)
        step = step / ( 1+ decay * t )
        t += 1
        return step

# ... on each example ...
w = w - AdamStep(gradient)
```

AllReduce

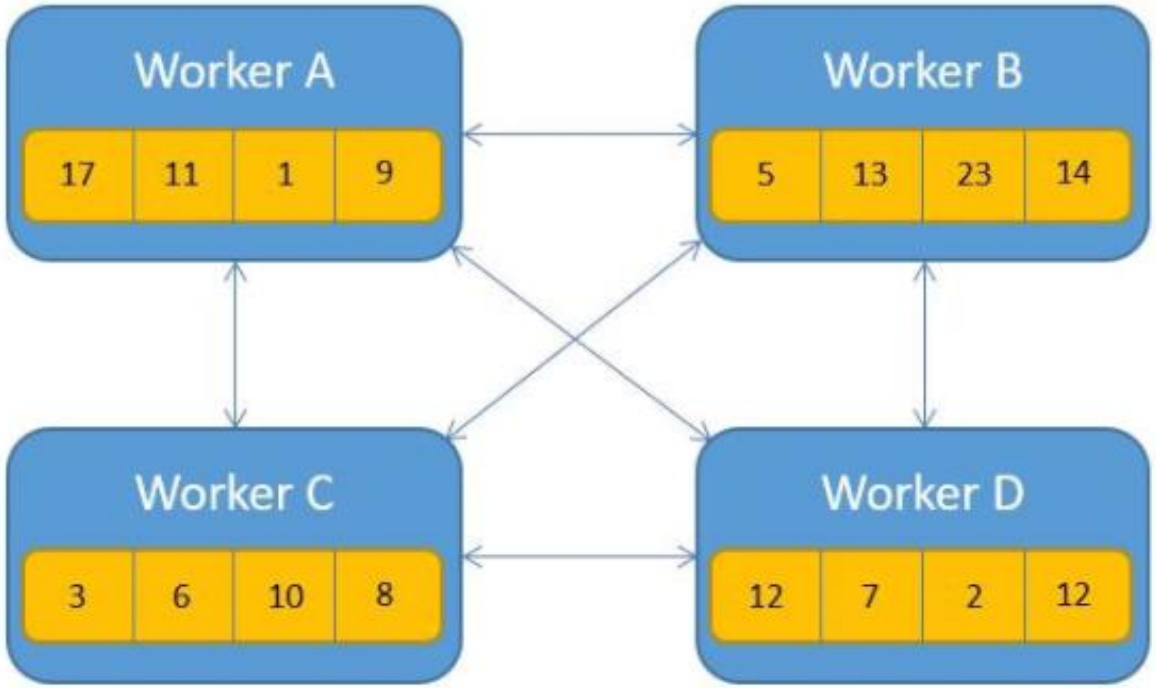
Local gradients on the workers
We need to :

- Sum all local gradients
- Send the sum to all workers

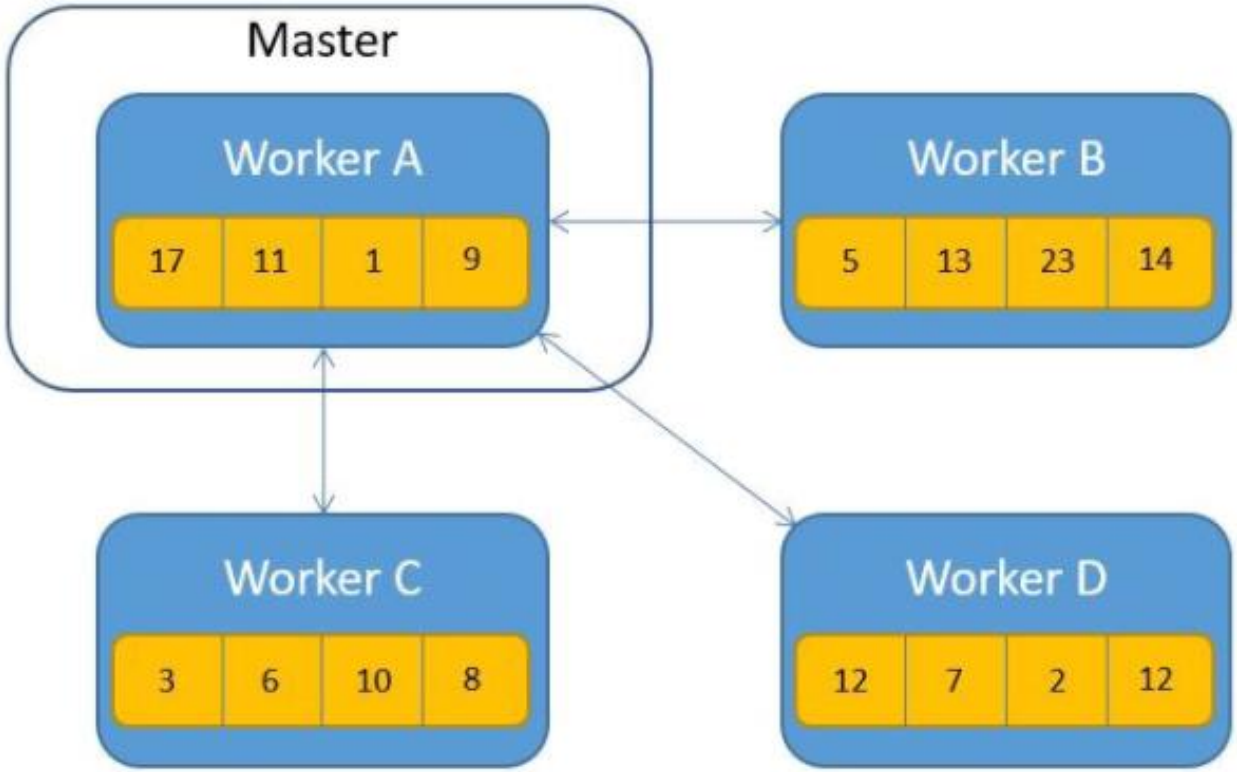


https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

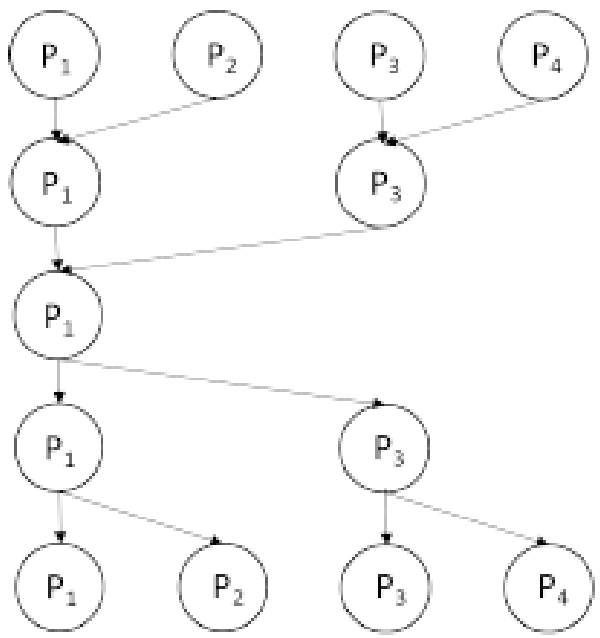
AllReduce: all to all communication



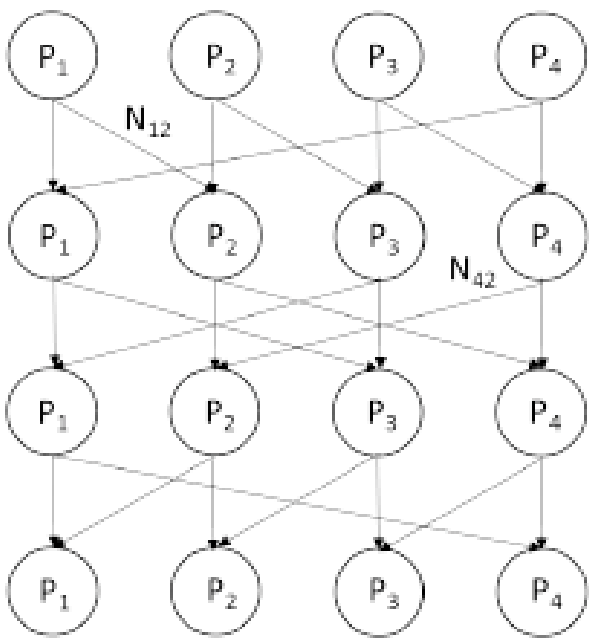
AllReduce: Reduce then Broadcast



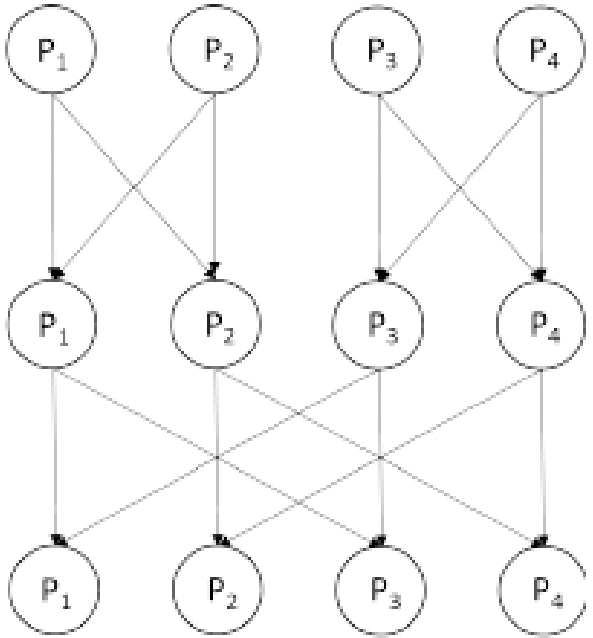
AllReduce



(a) Tree AllReduce

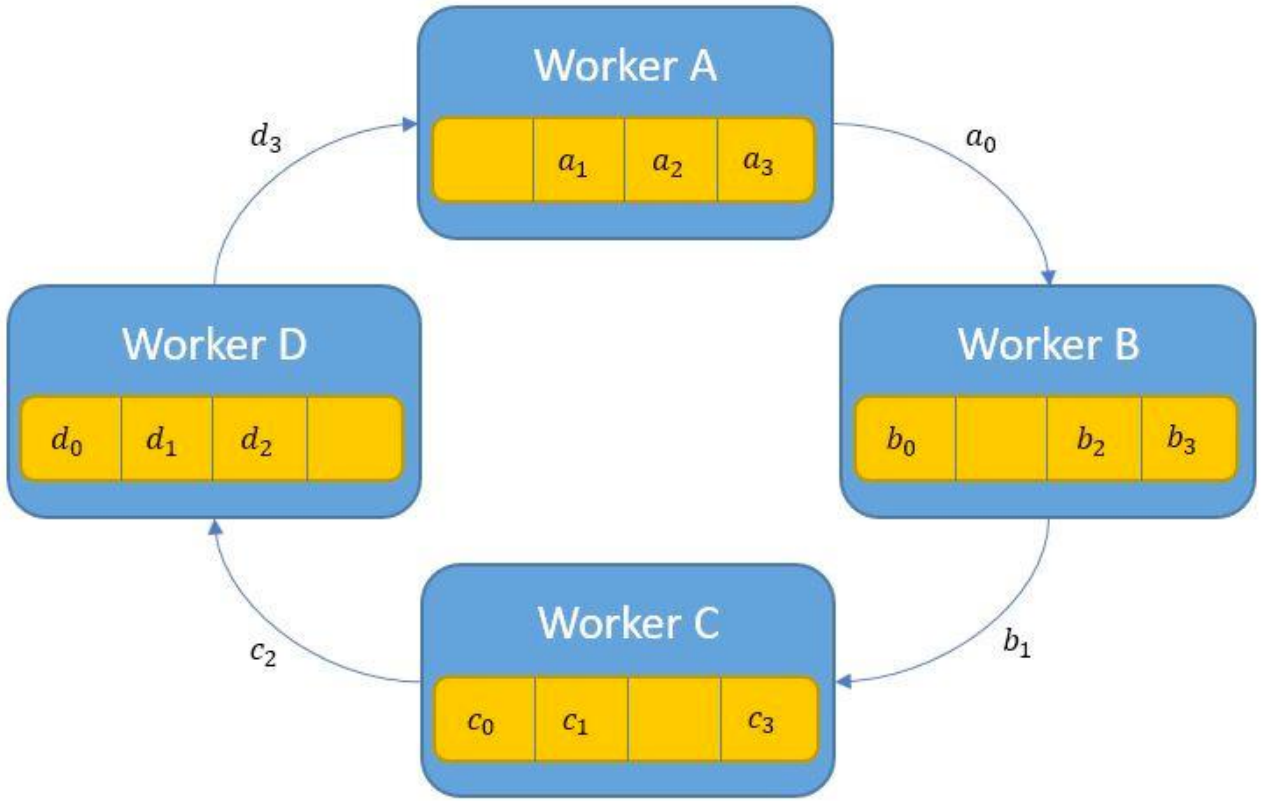


(b) Round-robin AllReduce

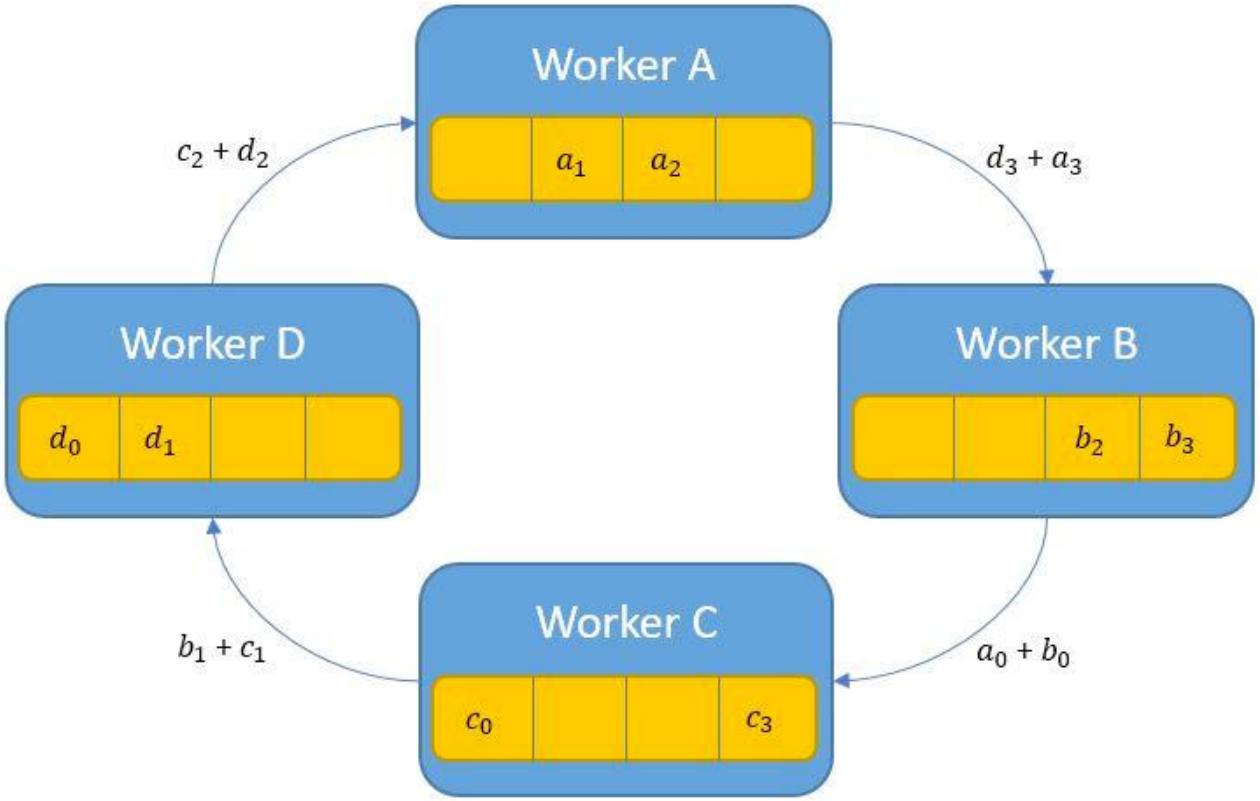


(c) Butterfly AllReduce

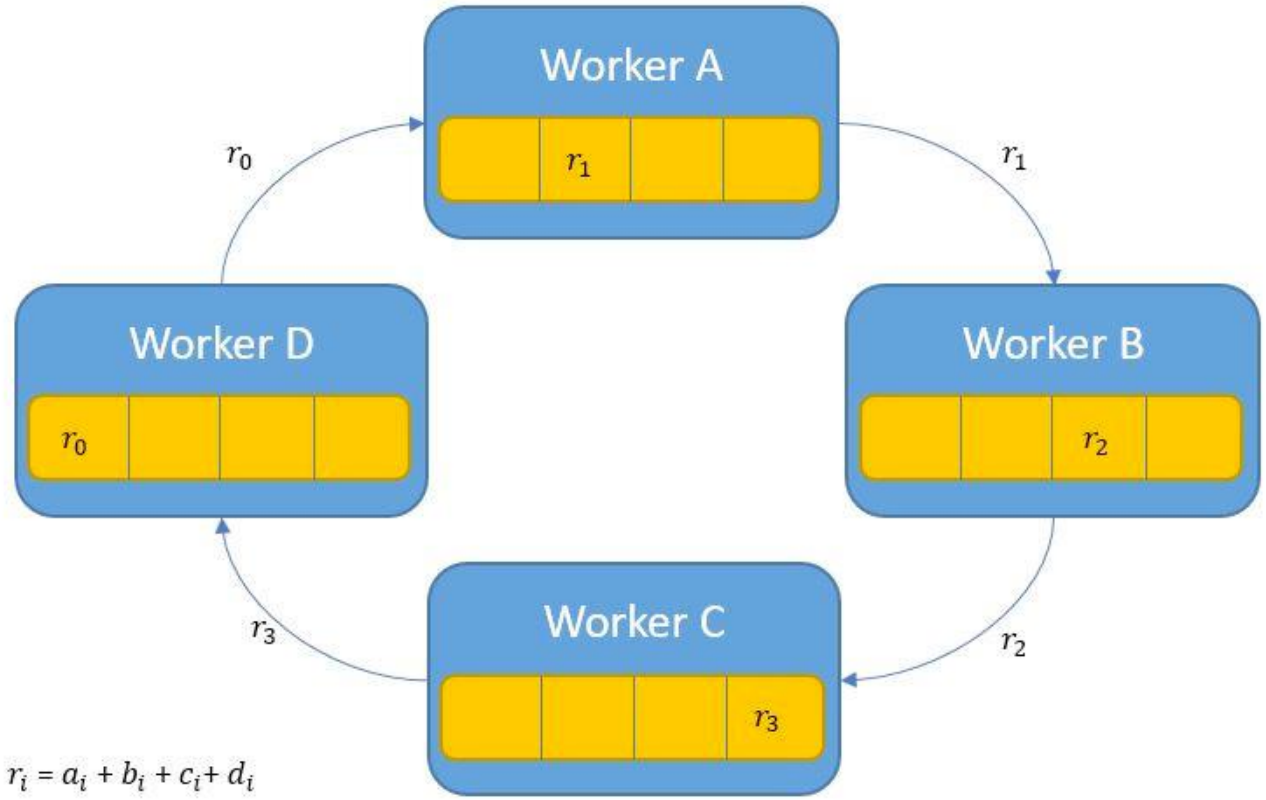
Ring AllReduce



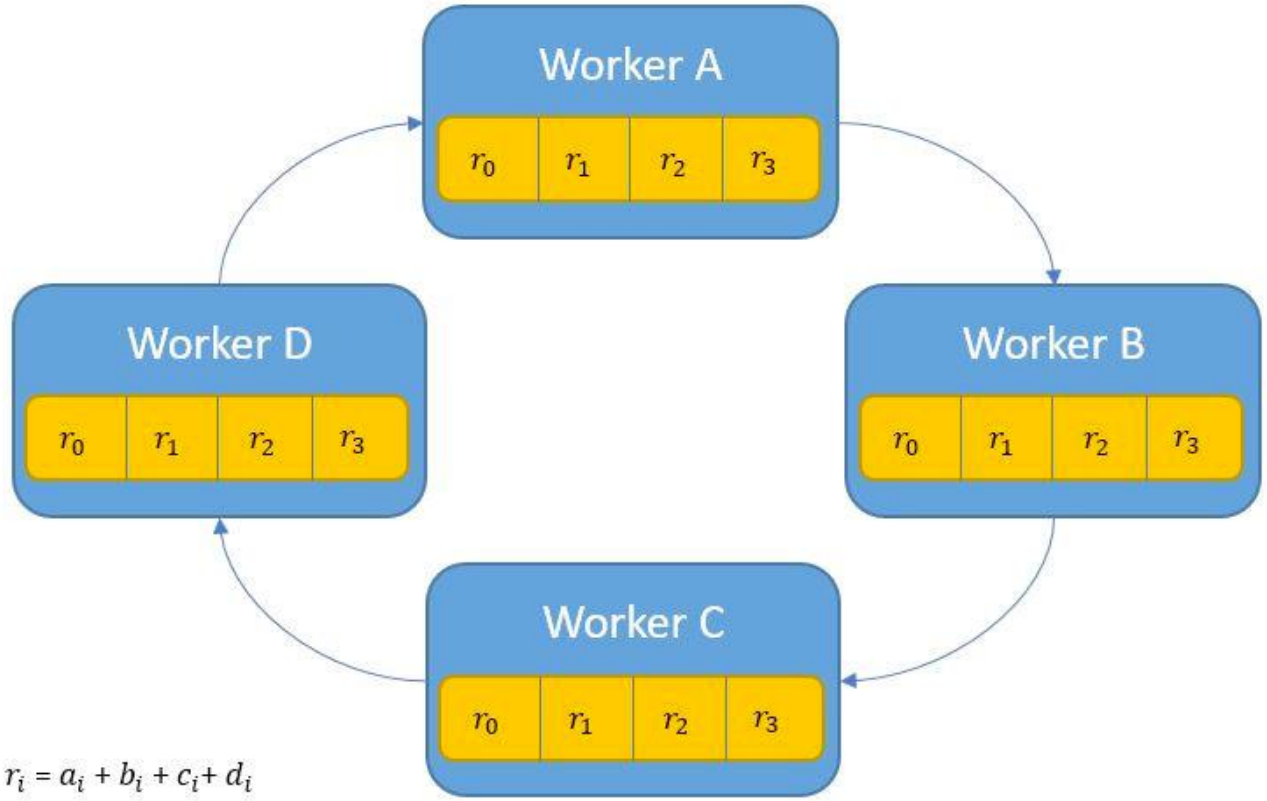
Ring AllReduce



Ring AllReduce



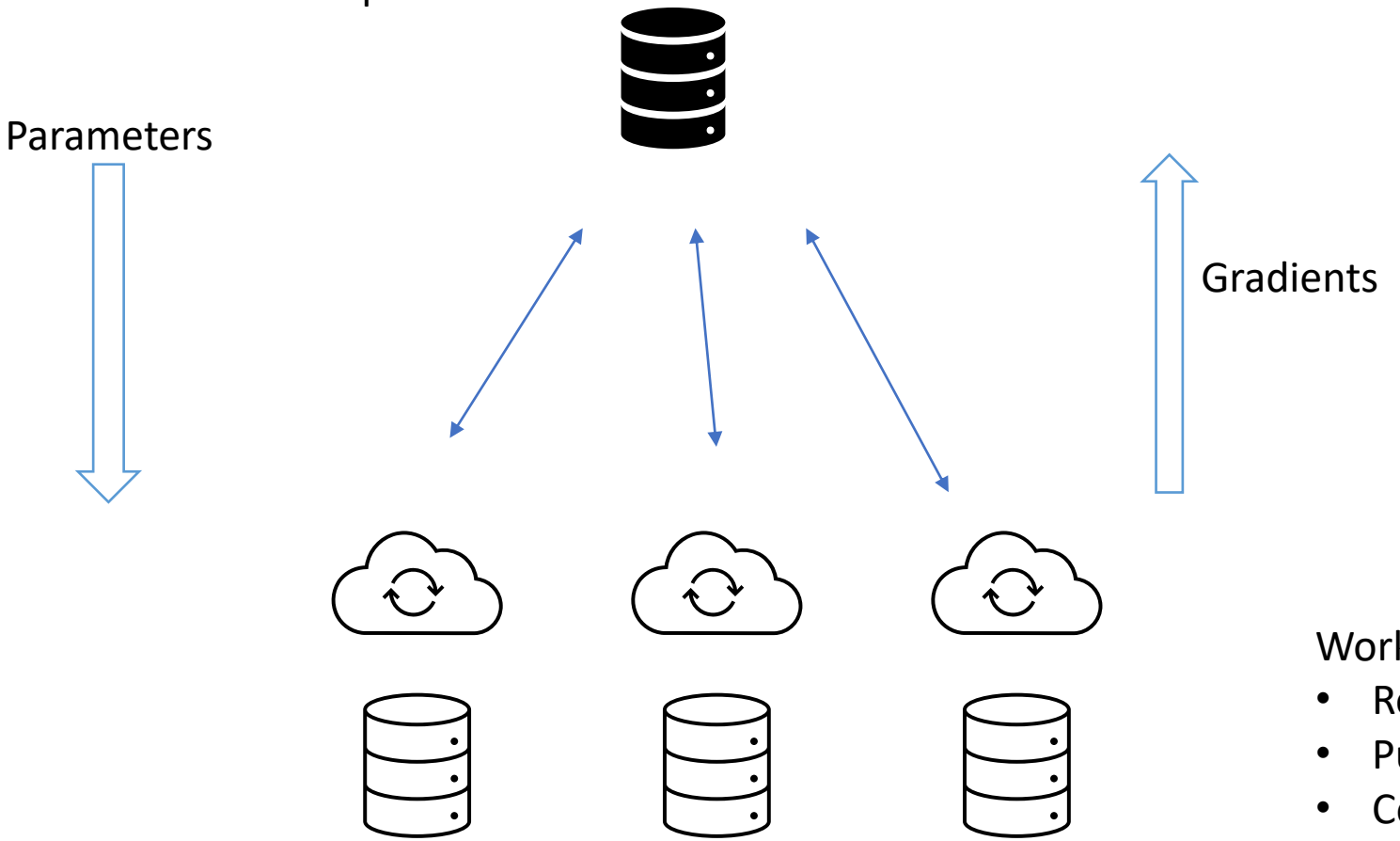
Ring AllReduce



$r_i = a_i + b_i + c_i + d_i$

Parameter server to avoid synchronization

One server with “latest” parameters

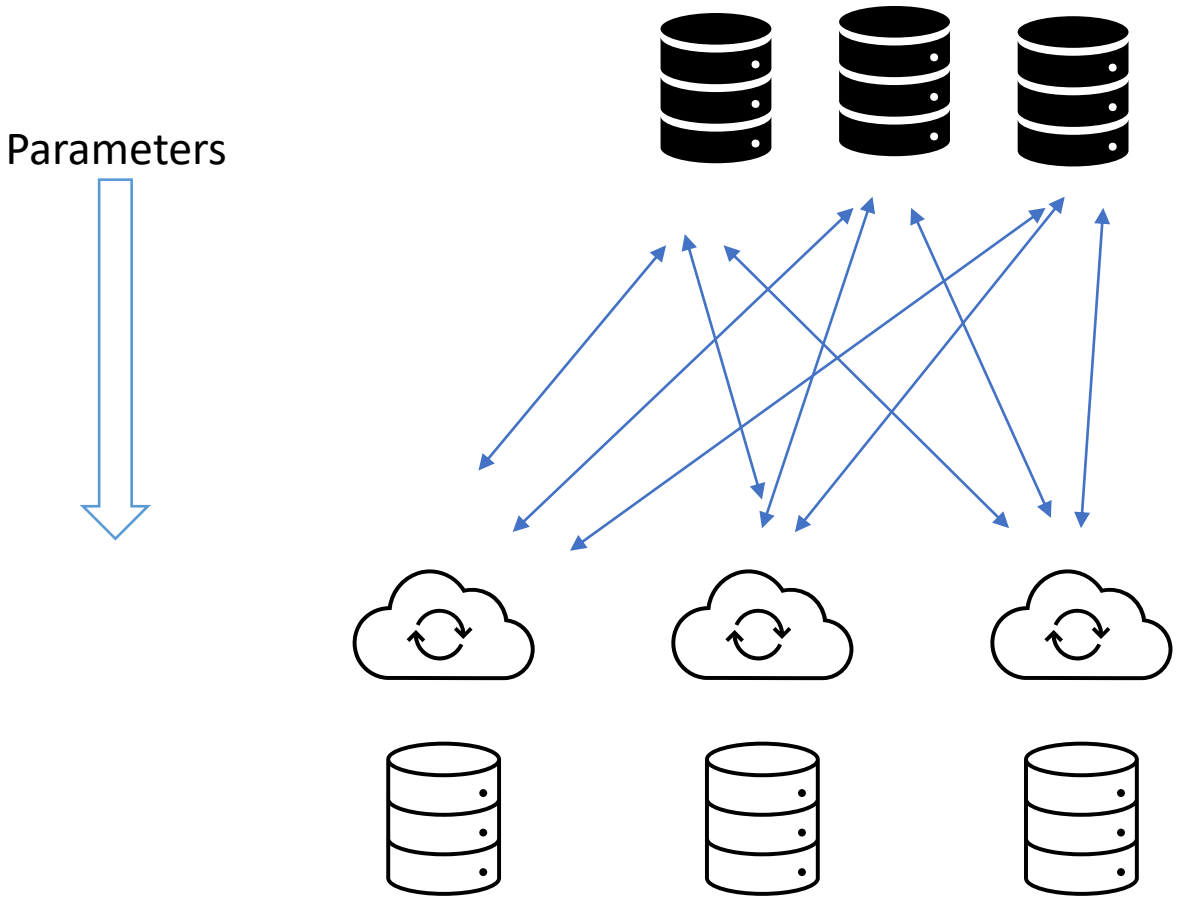


No sync between workers !

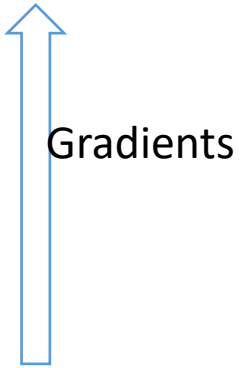
Workers iterate:

- Read a minibatch of data
- Pull latest parameter
- Compute the gradient
- Send parameters updates

Bottleneck: Bandwidth on the master?



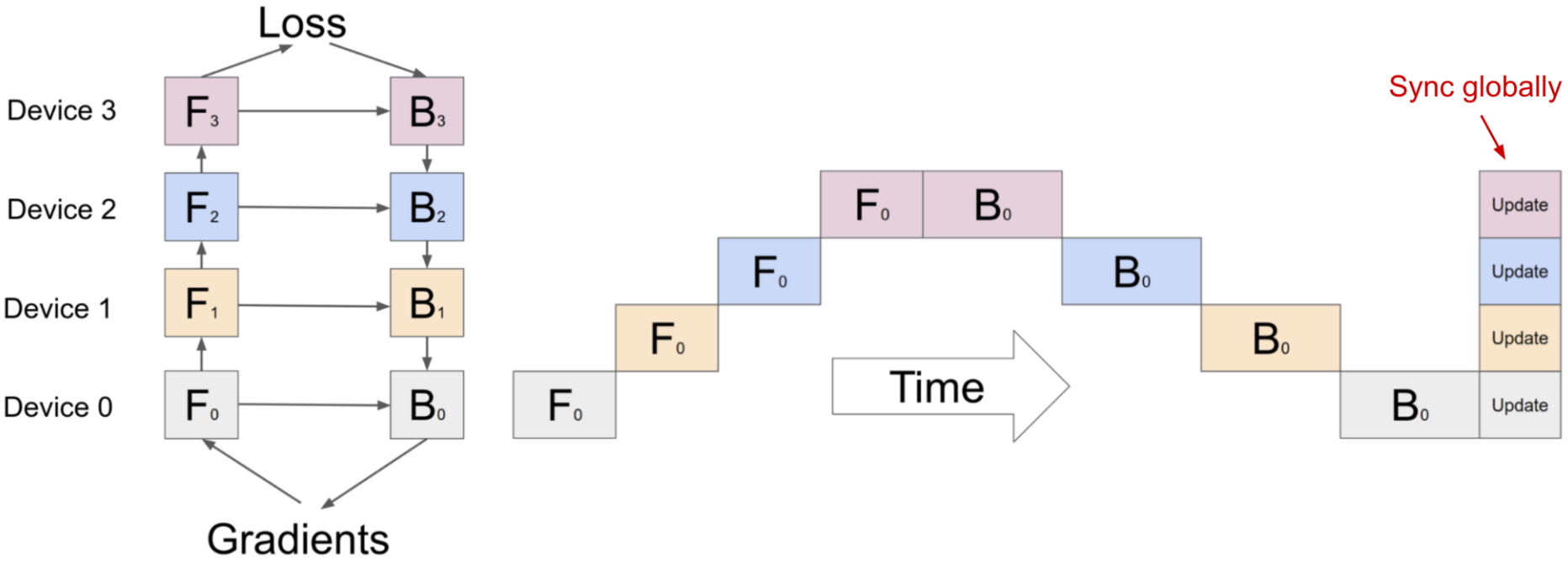
Split the parameters on several servers!



Beyond data parallelism

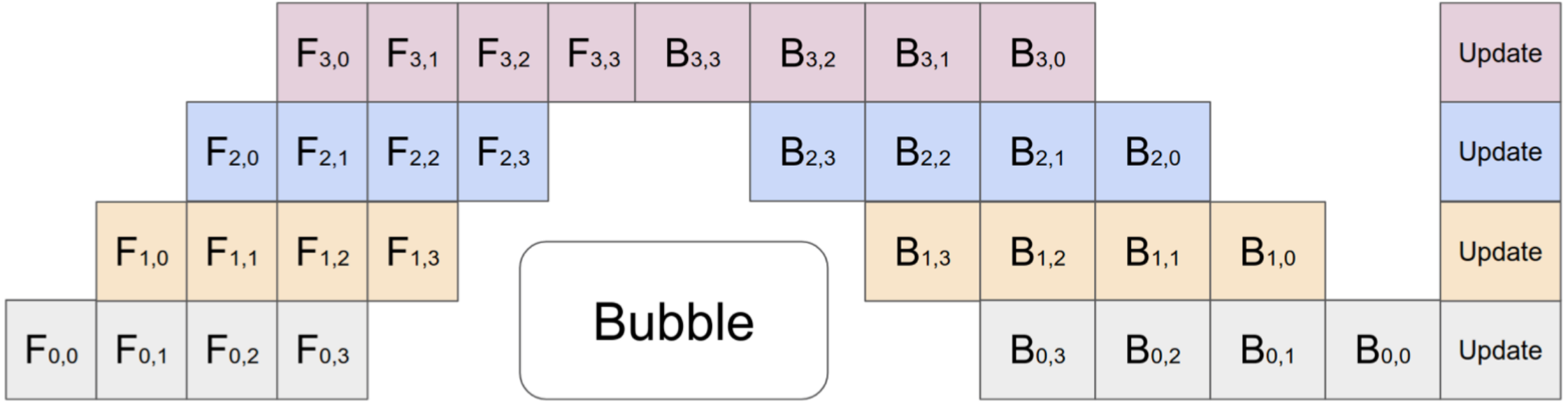
- Very large models: too big to fit on a single worker

=> Requires to split a model among several workers, eg one layer / worker



Pipeline parallelism
model parallelism + data parallelism

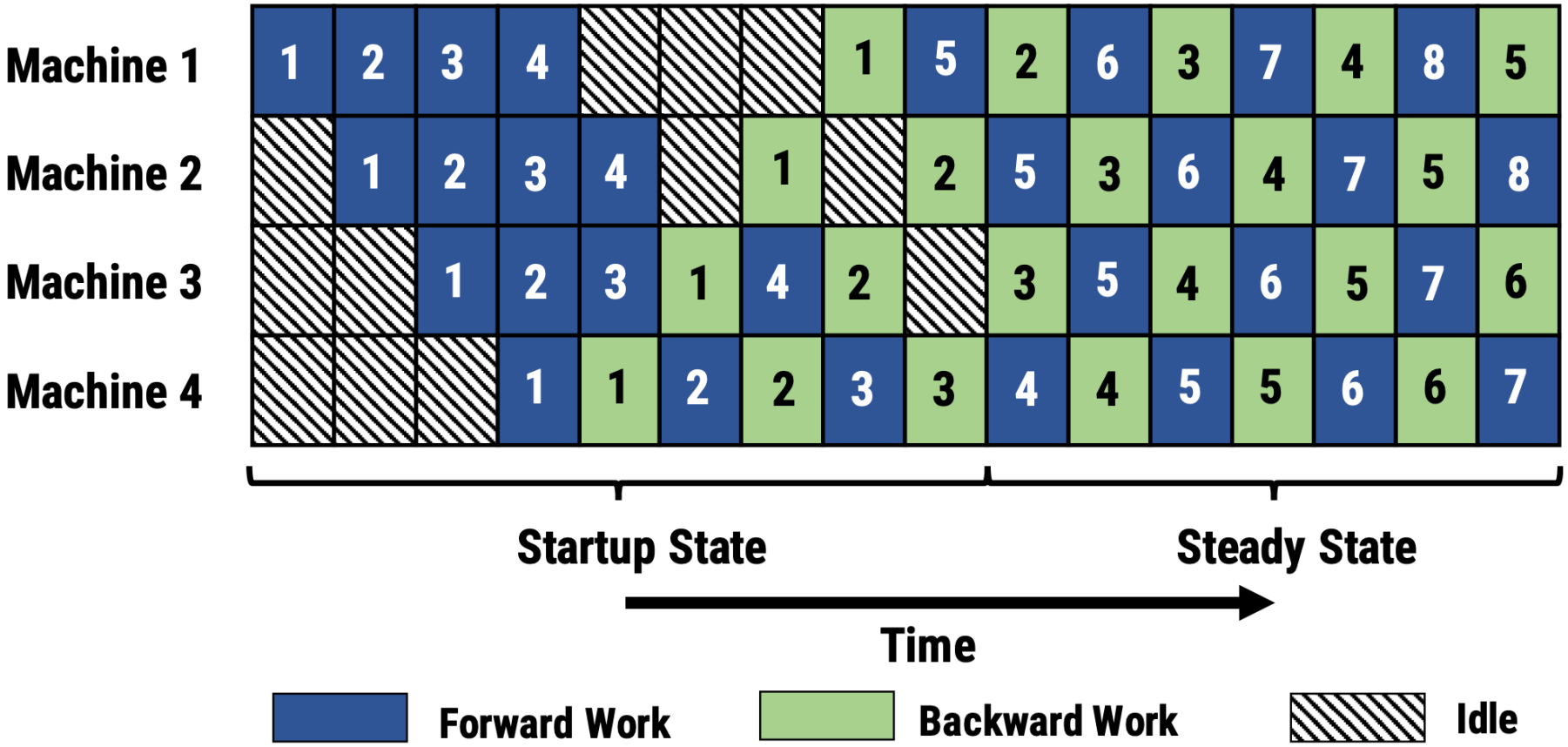
GPipe https://arxiv.org/abs/1811.06965



See also https://lilianweng.github.io/posts/2021-09-25-train-large/

Asynchonous pipeline parallelism

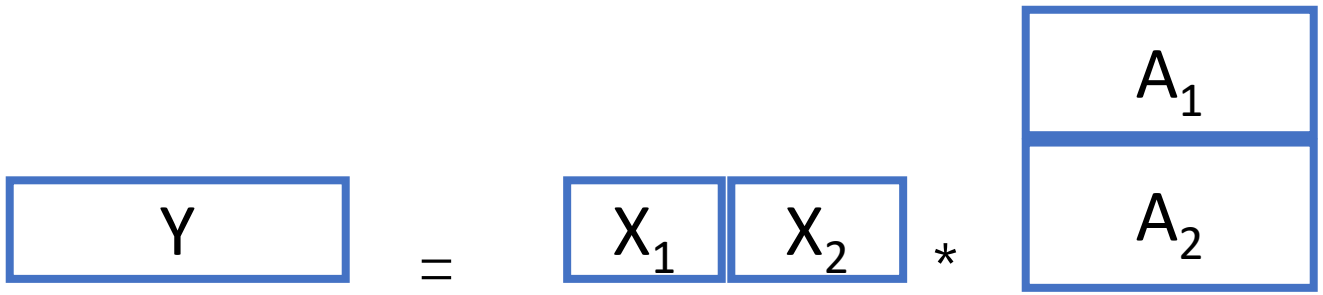
Pipedream https://cs.stanford.edu/~matei/papers/2019/sosp_pipedream.pdf



Complication:
Ensuring the same version of the weights are used in forward and backward passes

Tensor parrallelism

- When a single layer is too large (eg large input embedding layer)
- Typically, decomposing a matrix multiplication by block



- compute each block on different workers
- requires a synchronization to gather the blocks

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 …