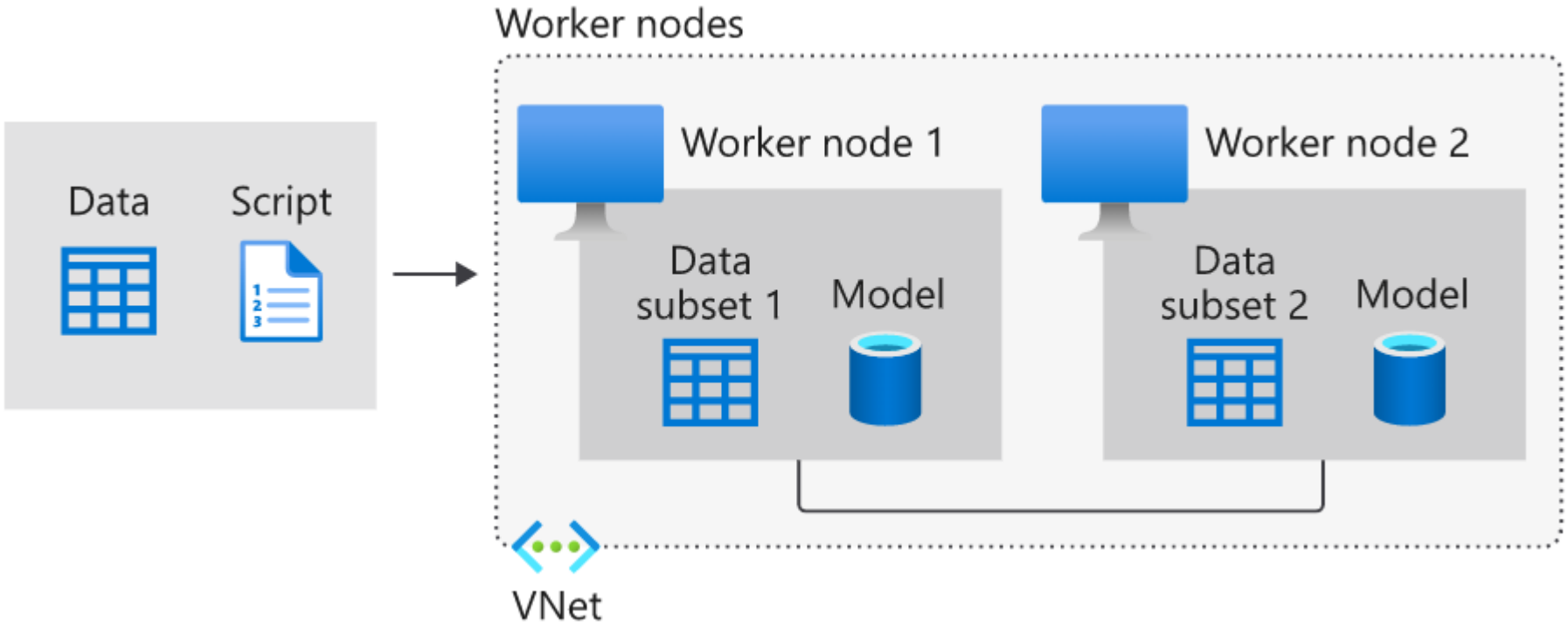
Big data

Trong-Hop Do

Distributed deep learning

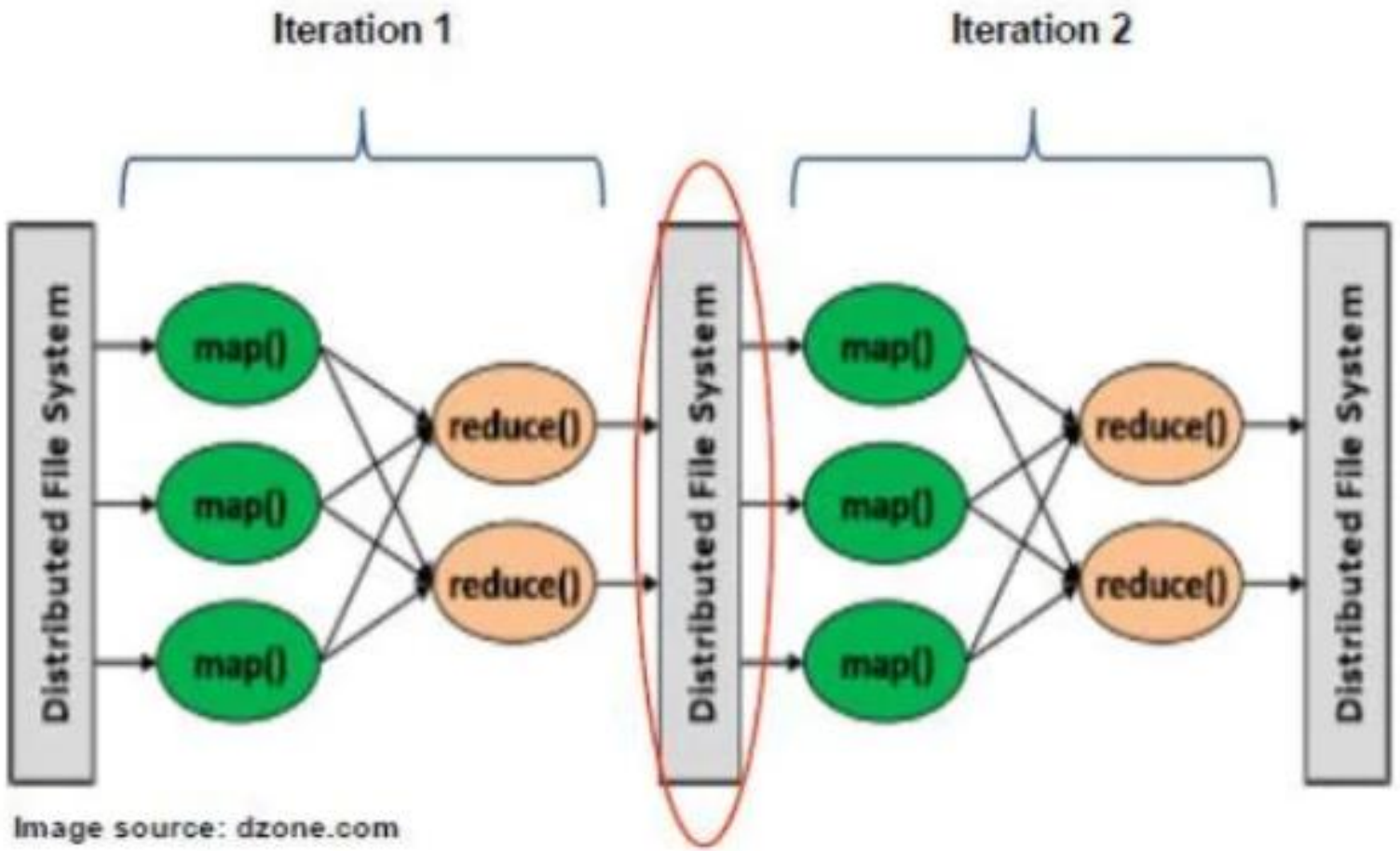
What is distributed ML?



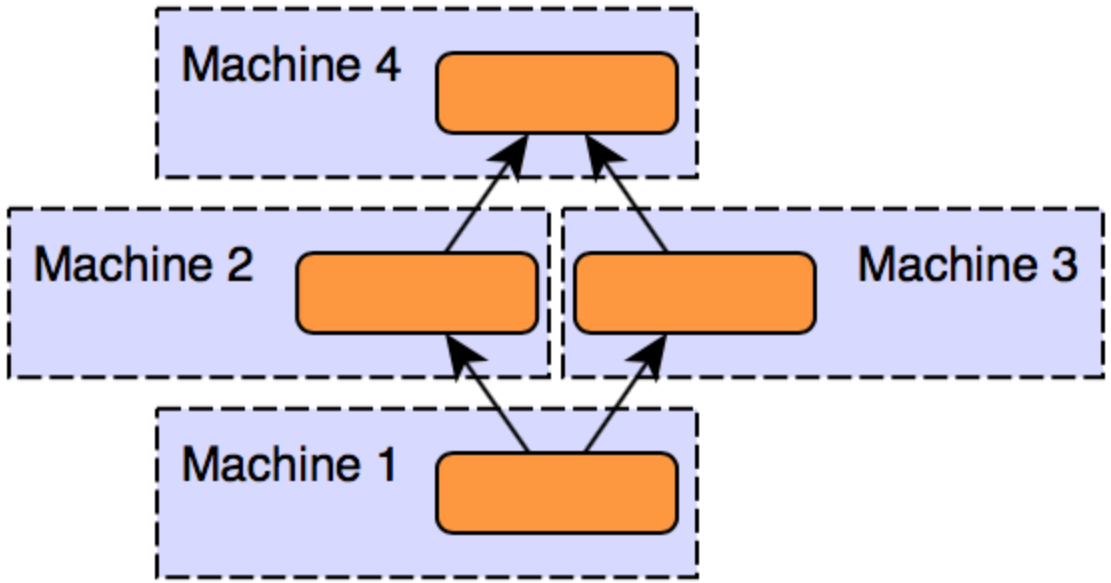
Why distributed ML?

- Large amounts of data to train
  - Explosion in types, speed, and scale of data
    - Types: image, time-series, structured, sparse
    - Speed: sensors, feed, financial
    - Scale: amount of data generated growing exponentially
  - Public datasets: processed splice genomic dataset is 250 GB and data subsampling is unhelpful
  - Private datasets: Google, Baidu perform learning over TBs of data
- Model sizes can be huge
  - Models with billions of parameters do not fit in a single machine
- Other benefits:
  - Faster IO
  - Hardware failure tolerance
  - Economical to use commondity hardware

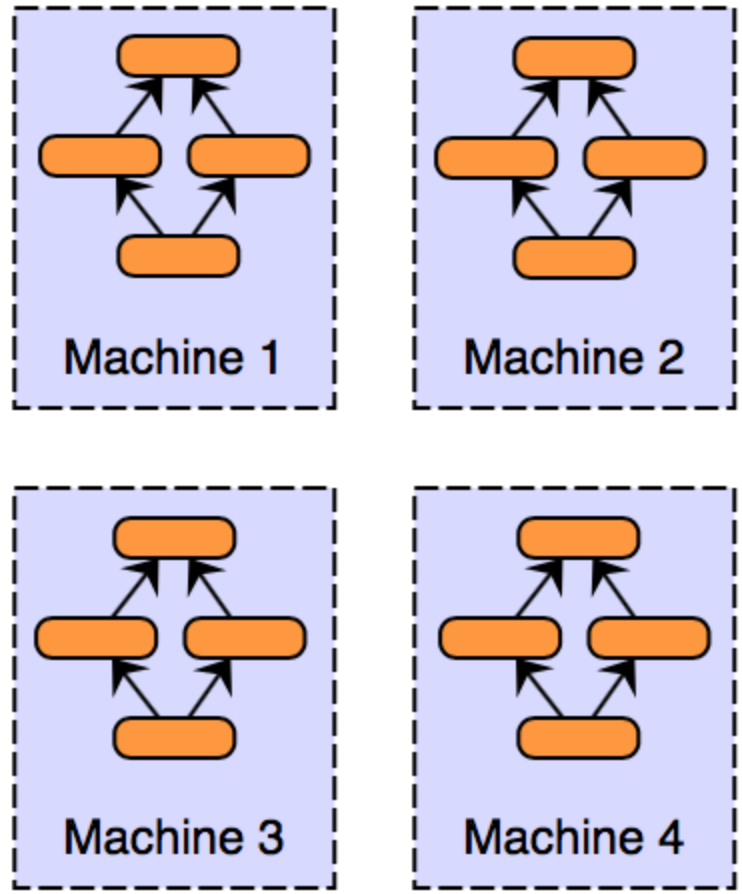
Naïve MapReduce is not good for ML

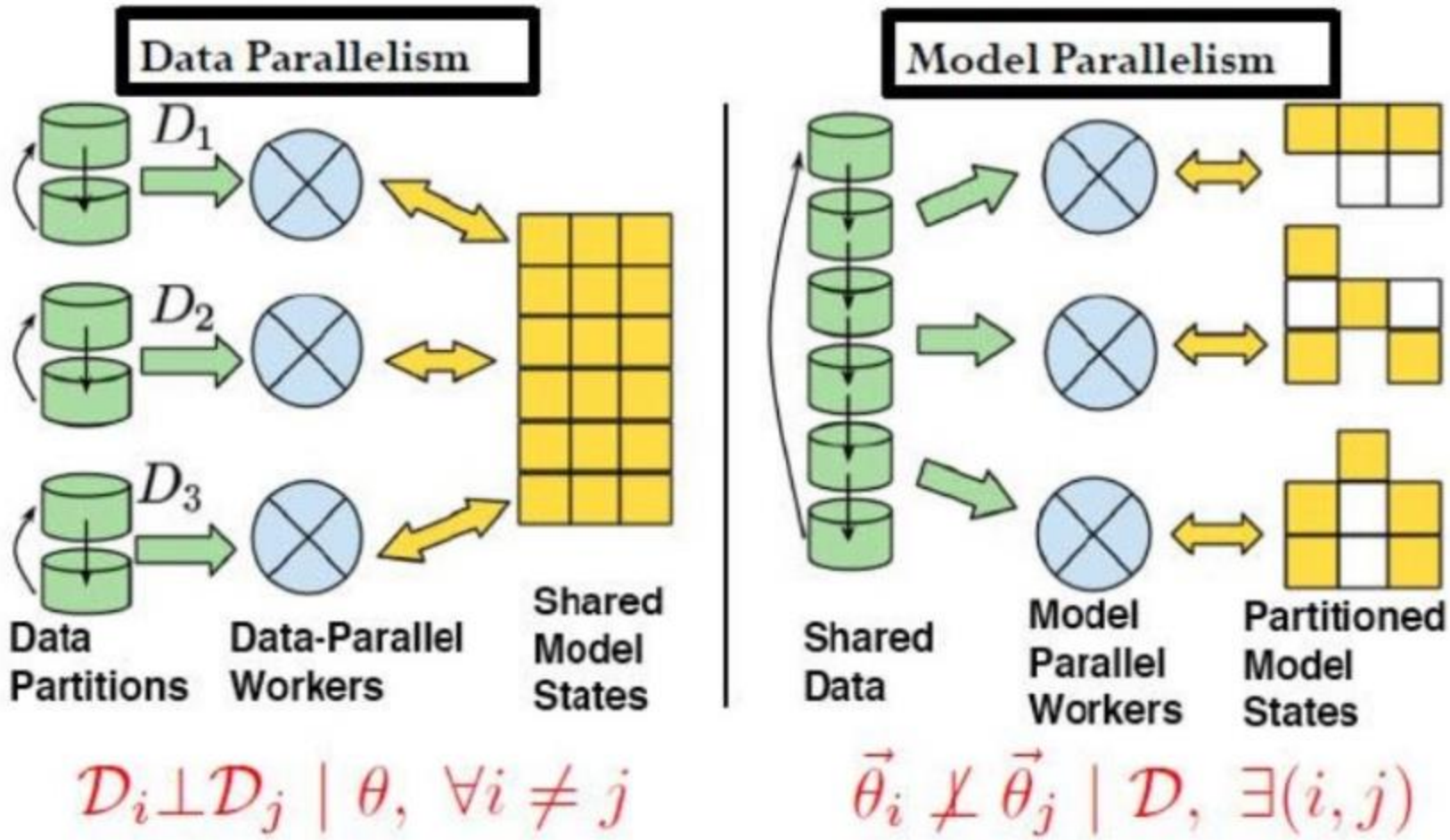


Model Parallelism



Data Parallelism

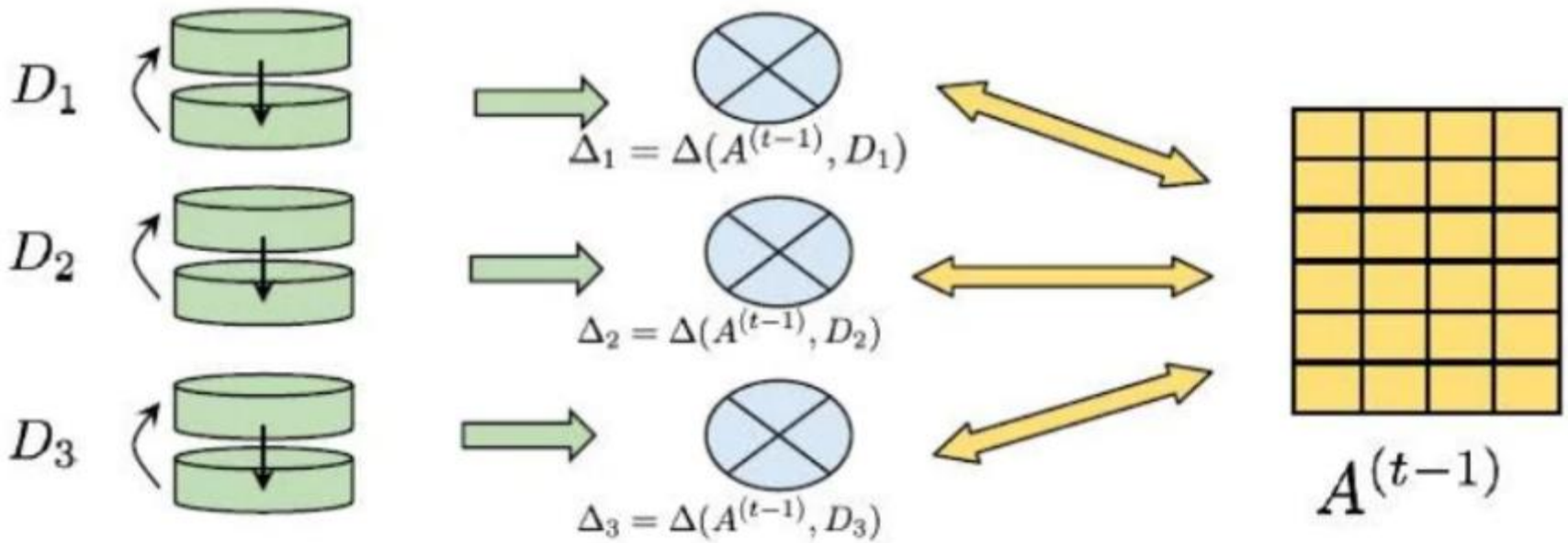




$$\mathcal{D}_i \perp \mathcal{D}_j \mid \theta, \ \forall i \neq j$$

$$\vec{\theta}_i \not\perp \vec{\theta}_j \mid \mathcal{D}, \ \exists (i, j)$$

Data Parallelism



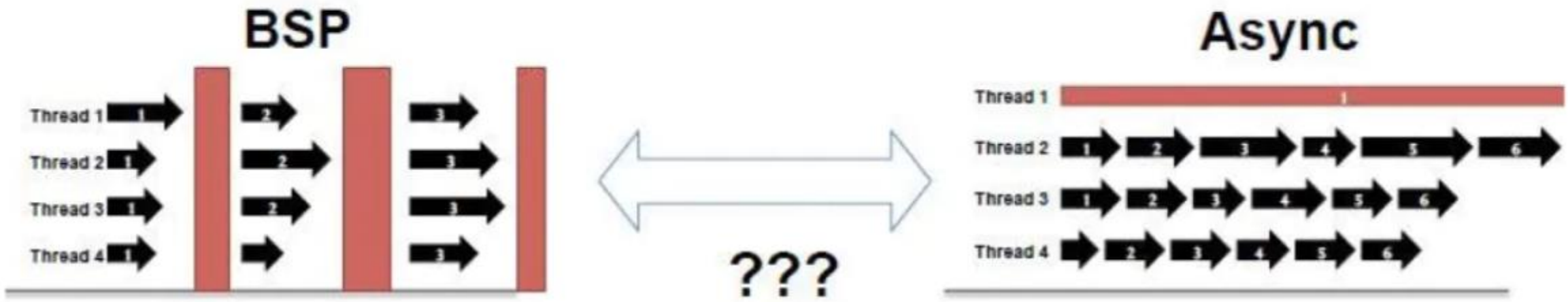
Additive Updates

$$\Delta = \sum_{p=1}^{3} \Delta_p$$

$$A^{(t)} = F(A^{(t-1)}, \Delta)$$

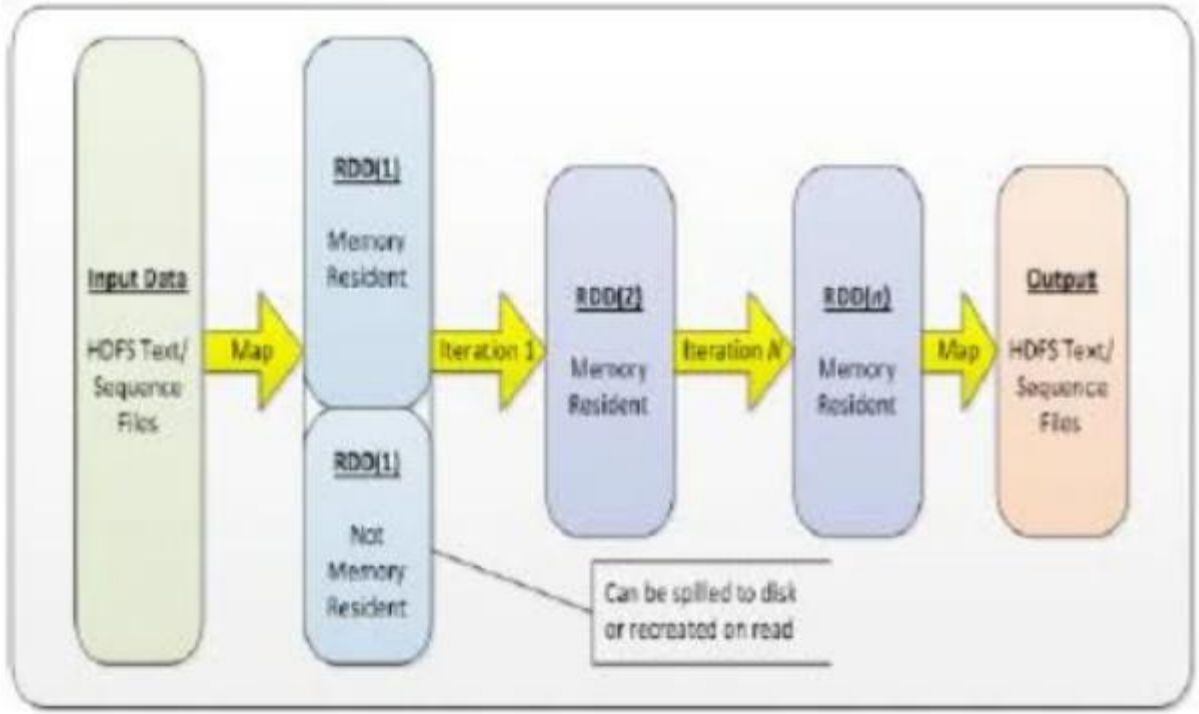
How to speed up Data-Parallelism?

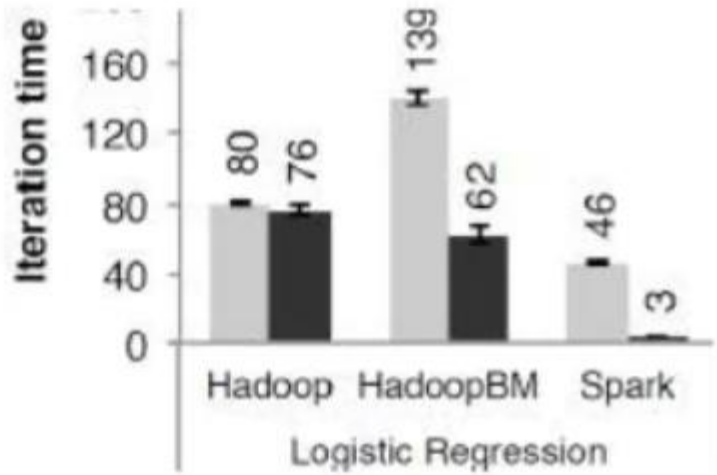
- **Existing ways are either safe/slow (BSP), or fast/risky (Async)**
- **Need "Partial" synchronicity**
  - Spread network comms evenly (don't sync unless needed)
  - Threads usually shouldn't wait – but mustn't drift too far apart!
- **Need straggler tolerance**
  - Slow threads must somehow catch up



Spark: Faster MapR on Data-Parallel

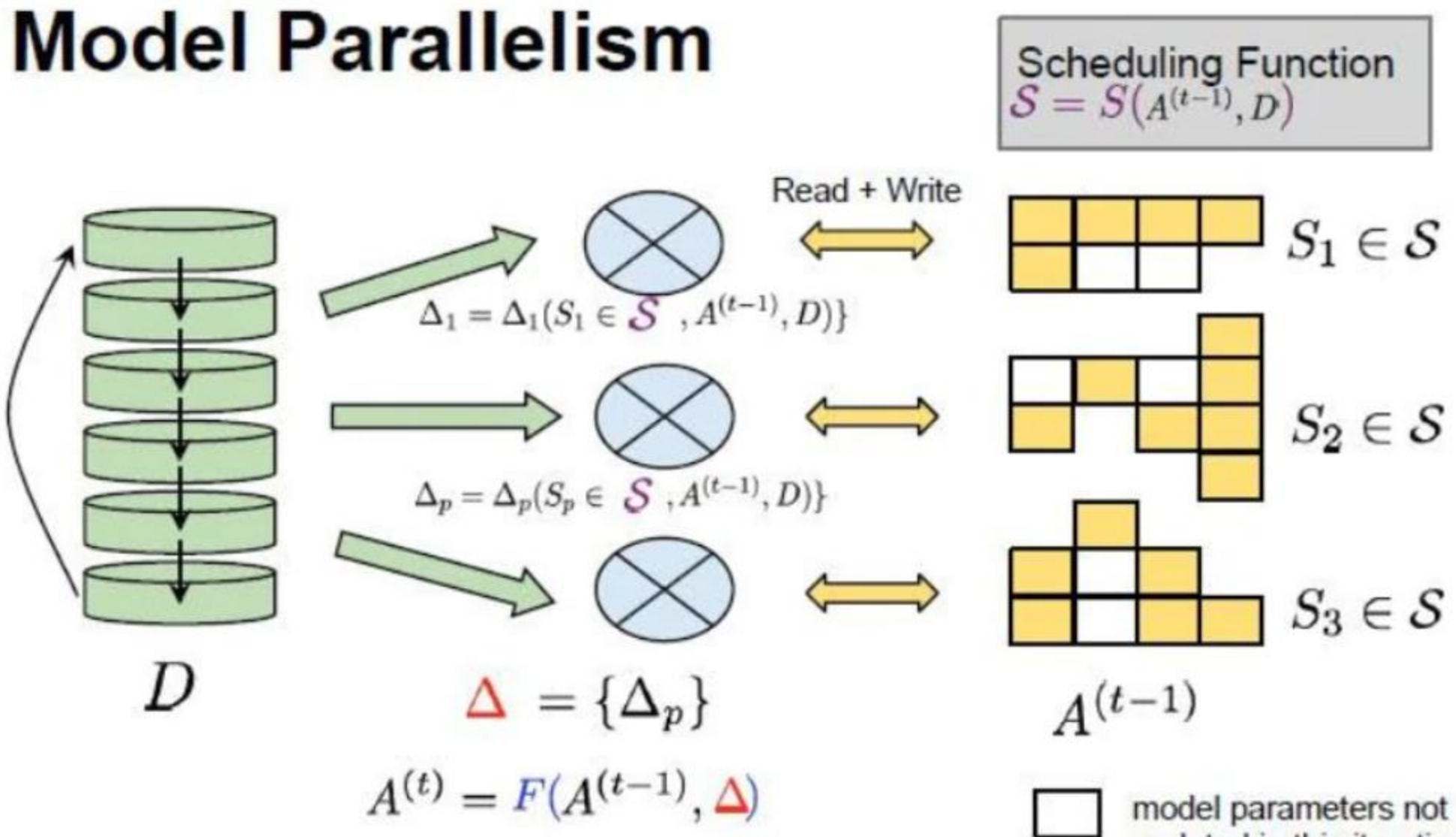
- Spark's solution: **Resilient Distributed Datasets (RDDs)**
  - Input data → load as RDD → apply transforms → output result
  - RDD transforms strict superset of MapR
  - RDDs cached in memory, avoid disk I/O





- **Spark ML library supports data-parallel ML algos, like Hadoop**

Model Parallelism



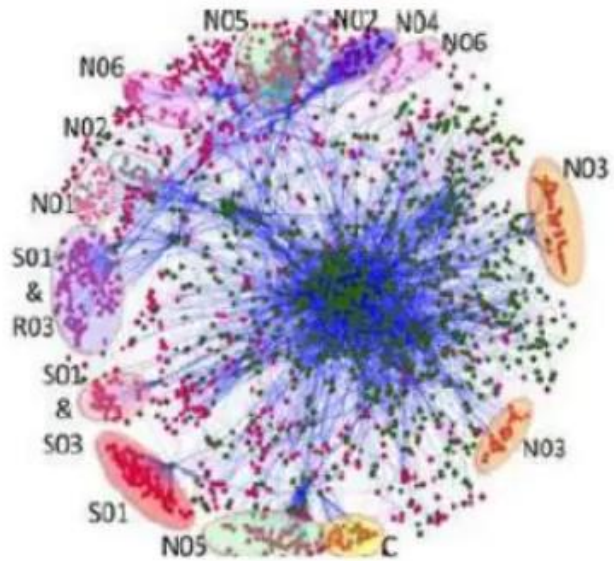
Model Parallel ML – Parameter Server

- Central server to merge updates every few iterations
  - Workers send updates asynchronously and receive whole models from the server
  - Central server merges incoming models and returns the latest model
  - Example: Distbelief (NIPS 2012), Parameter Server (OSDI 2014), Project Adam (OSDI 2014)

Graph Parallel ML – BSP, Pregel, GAS

- Big graphs emerge in many real applications
- Graph-based machine learning is a hot research topic with wide applications: *relational learning, manifold learning, PageRank, community detection, etc*
- Distributed graph computing frameworks for big graphs

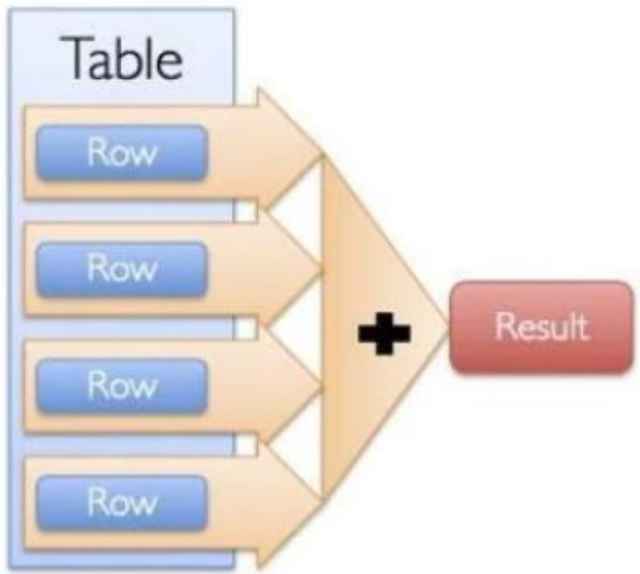
(a) Social Network



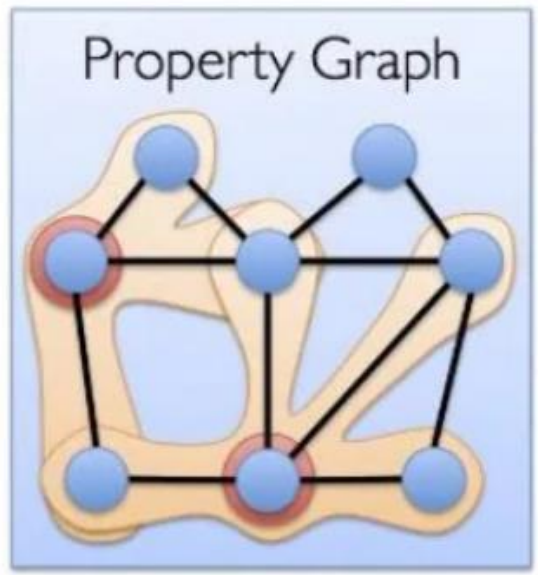
(b) Biological Network

Graph Parallel vs Data Parallel

Data-Parallel



Graph-Parallel



Graph parallel is new technique to partition and distribute data and execute machine learning algorithm orders of magnitude faster than data parallel approach!

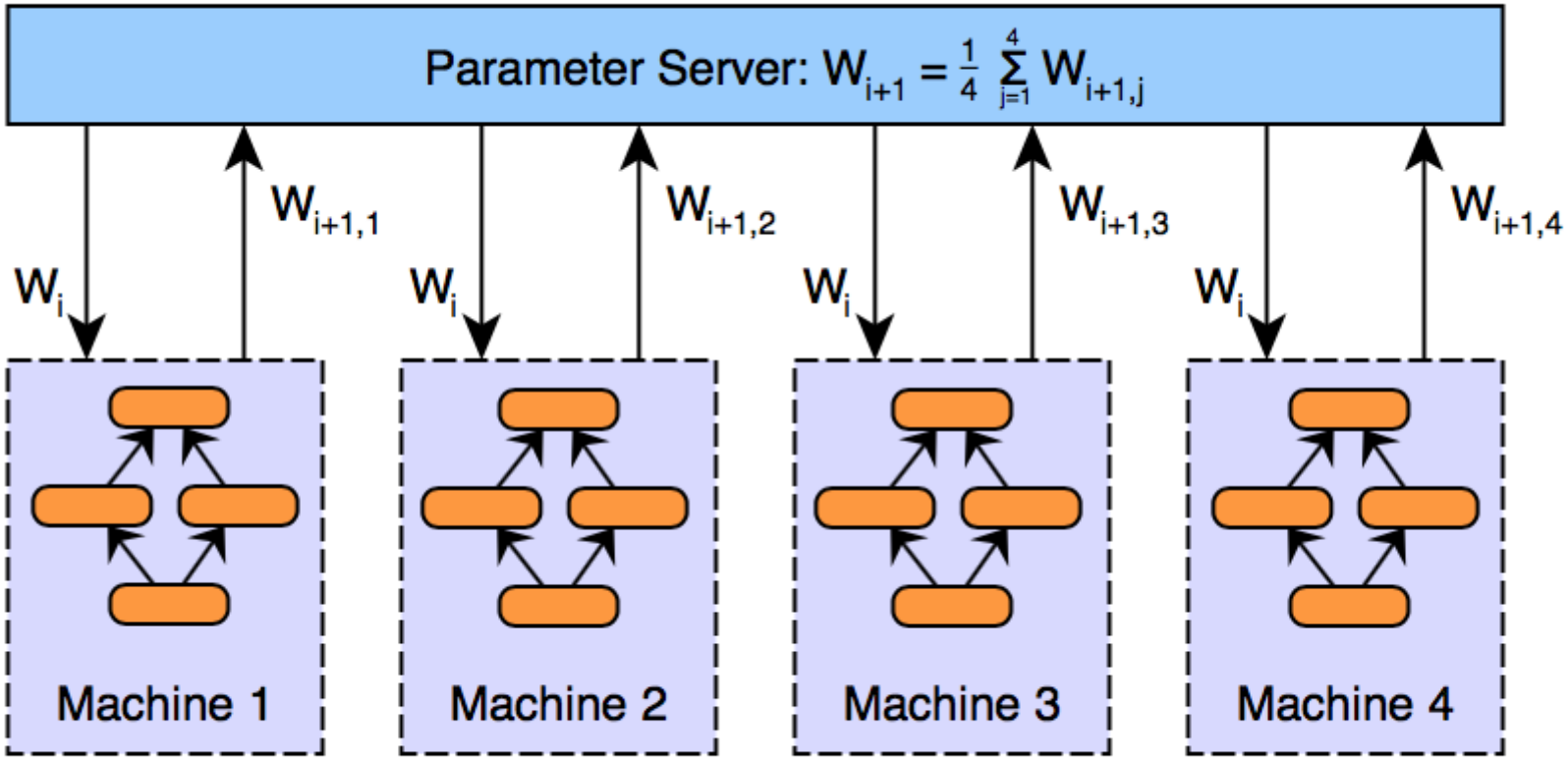
Distributed deep learning

- Data parallelism is preferred
  - Easy implementation
  - Fault tolerance
  - good cluster utilization

Differences between Data Parallelism approaches

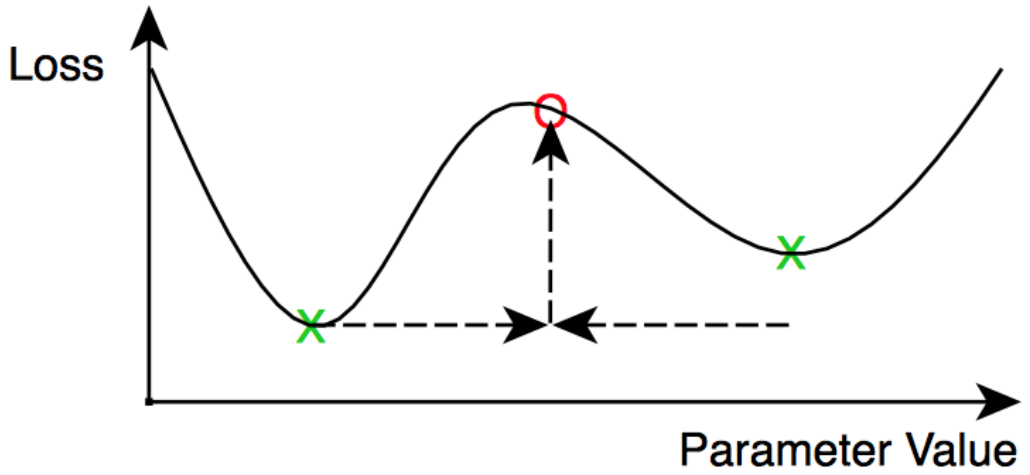
- Parameter averaging vs. update (gradient)-based approaches
- Synchronous vs. asynchronous methods
- Centralized vs. distributed synchronization

Parameter averaging

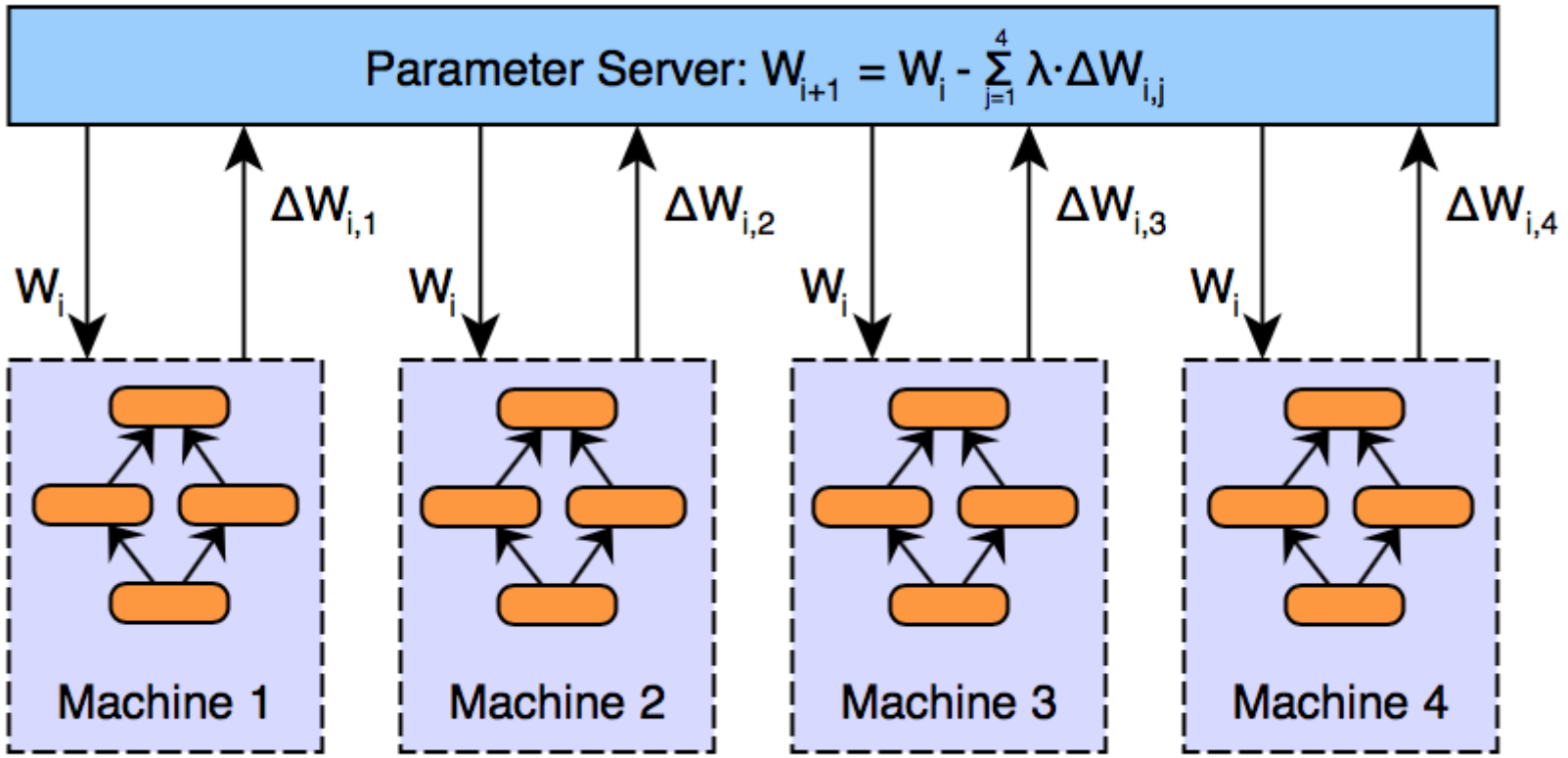


Parameter averaging

- Naive approach: average the parameters after each iteration
  - Problems: network communication and synchronization costs may overwhelm the benefit obtained from the extra machines
- Better approach: parameter averaging with an averaging period
  - Problems: choosing averaging period is difficult. Too high averaging period results in divergence in local parameters and thus poor model after averaging



Update (gradient)-based

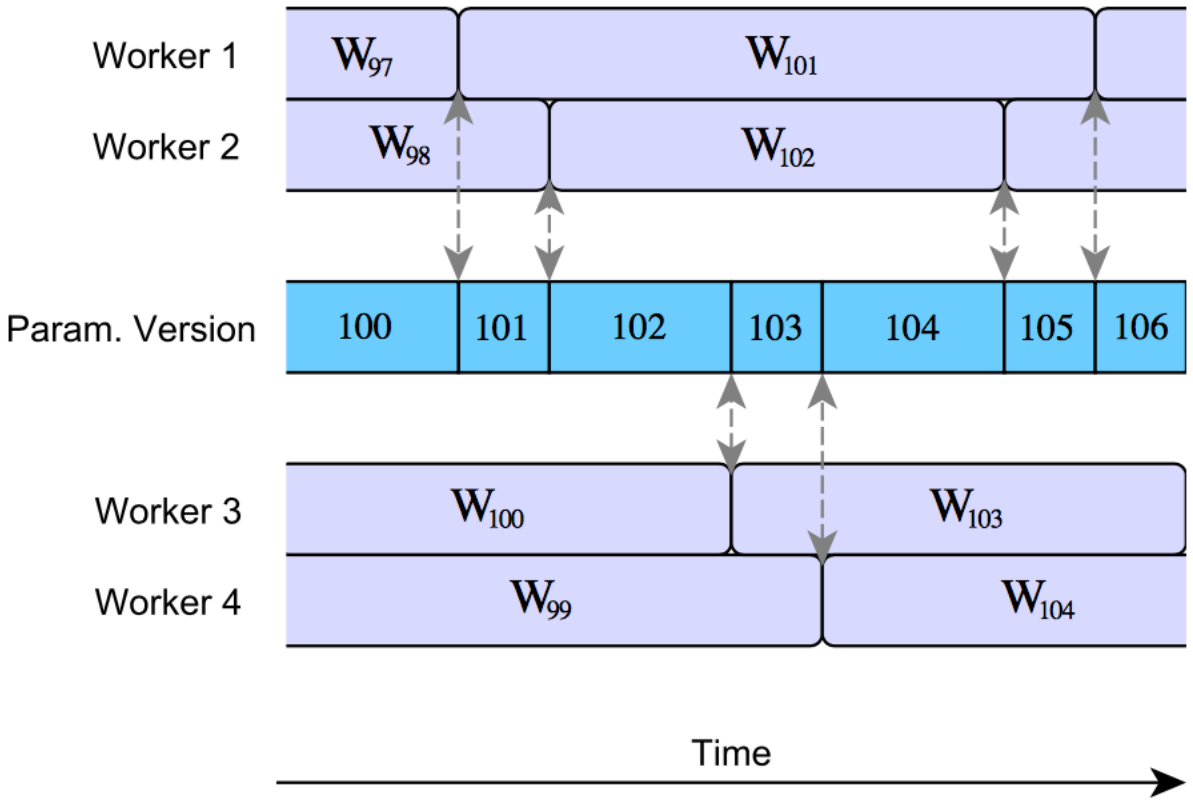


Asynchronous Stochastic Gradient Descent

- Key point: updates is applied to the parameter vector as soon as they are computed
- Benefits:
  - First, we can potentially gain higher throughput in our distributed system: workers can spend more time performing useful computations, instead of waiting around for the parameter averaging step to be completed.
  - Second, workers can potentially incorporate information (parameter updates) from other workers sooner than when using synchronous

Asynchronous Stochastic Gradient Descent

- Problem: gradient staleness



$$\vdots$$

$$W_{101} = W_{100} - \lambda \Delta W_{97}$$

$$W_{102} = W_{101} - \lambda \Delta W_{98}$$

$$W_{103} = W_{102} - \lambda \Delta W_{100}$$

$$W_{104} = W_{103} - \lambda \Delta W_{99}$$

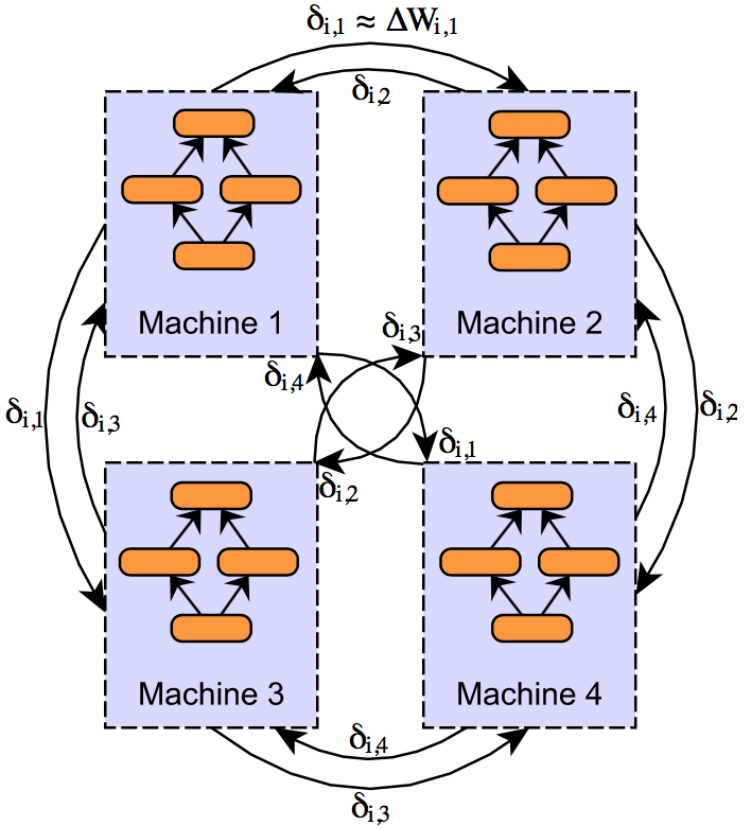
$$\vdots$$

Asynchronous Stochastic Gradient Descent

- Some approaches to dealing with stale gradients include:
  - Scaling the value λ separately for each update $ΔWi,j$ based on the staleness of the gradients
  - Implementing ‘soft’ synchronization protocols
  - Use synchronization to bound staleness. For example, delay faster workers when necessary, to ensure that the maximum staleness is below some threshold

Decentralized Asychronous Stochastic Gradient Descent

- No centralized parameter server is present in the system (instead, peer to peer communication is used to transmit model updates between workers).
- Updates are heavily compressed, resulting in the size of network communications being reduced by some 3 orders of magnitude.



Decentralized Asychronous Stochastic Gradient Descent

- Update vectors:
  - Sparse: only some of the gradients are communicated in each vector (the remainder are assumed to be 0) - sparse entries are encoded using an integer index
  - Quantized to a single bit: each element of the sparse update vector takes value +τ or −τ.
  - Integer indexes (used to identify the entries in the sparse array) are optionally compressed using entropy coding to further reduce update sizes

Compression	Update Size	Reduction
None (32-bit floating point)	58.4 MB	-
16-bit floating point	29.2 MB	50%
Quantized, $\tau=2$	0.21 MB	99.6%

Decentralized Asychronous Stochastic Gradient Descent

- Three main downsides:
  - Convergence can suffer in the early stages of training
  - Compression and quantization is not free: these processes result in extra computation time per minibatch, and a small amount of memory overhead per executor
  - The process introduces two additional hyperparameters to consider: the value for τ and whether to use entropy coding for the updates or not (though notably both parameter averaging and async SGD also introduce additional hyperparameters)

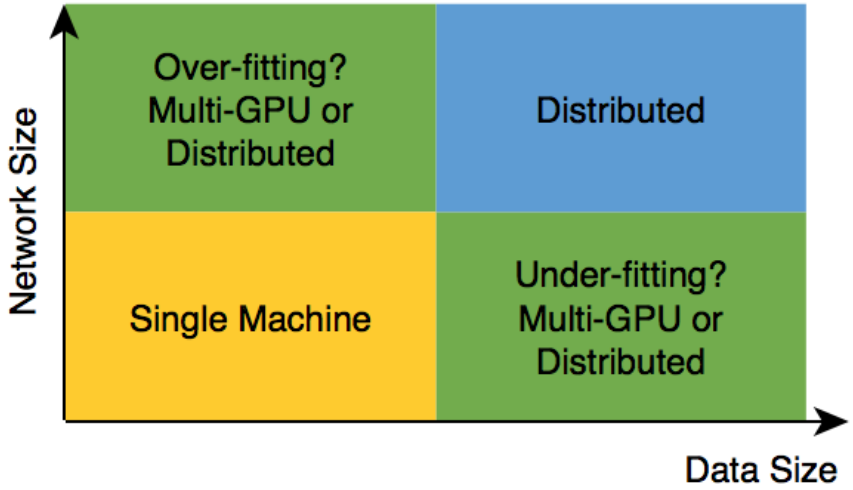
Distributed Neural Network Training: Which Approach is Best?

- Synchronous parameter averaging
  - Best accuracy per epoch, and the overall attainable accuracy, especially with small averaging periods (N = 1 averaging period most closely approximates single machine training)
  - slower per epoch due to synchronization cost
  - greatest issue with parameter averaging (and synchronous approaches in general) is the so-called ‘last executor’ effect: that is, synchronous systems have to wait on the slowest executor before completing each iteration.
- Consequently, synchronous systems are less viable as the total number of workers increases.

Distributed Neural Network Training: Which Approach is Best?

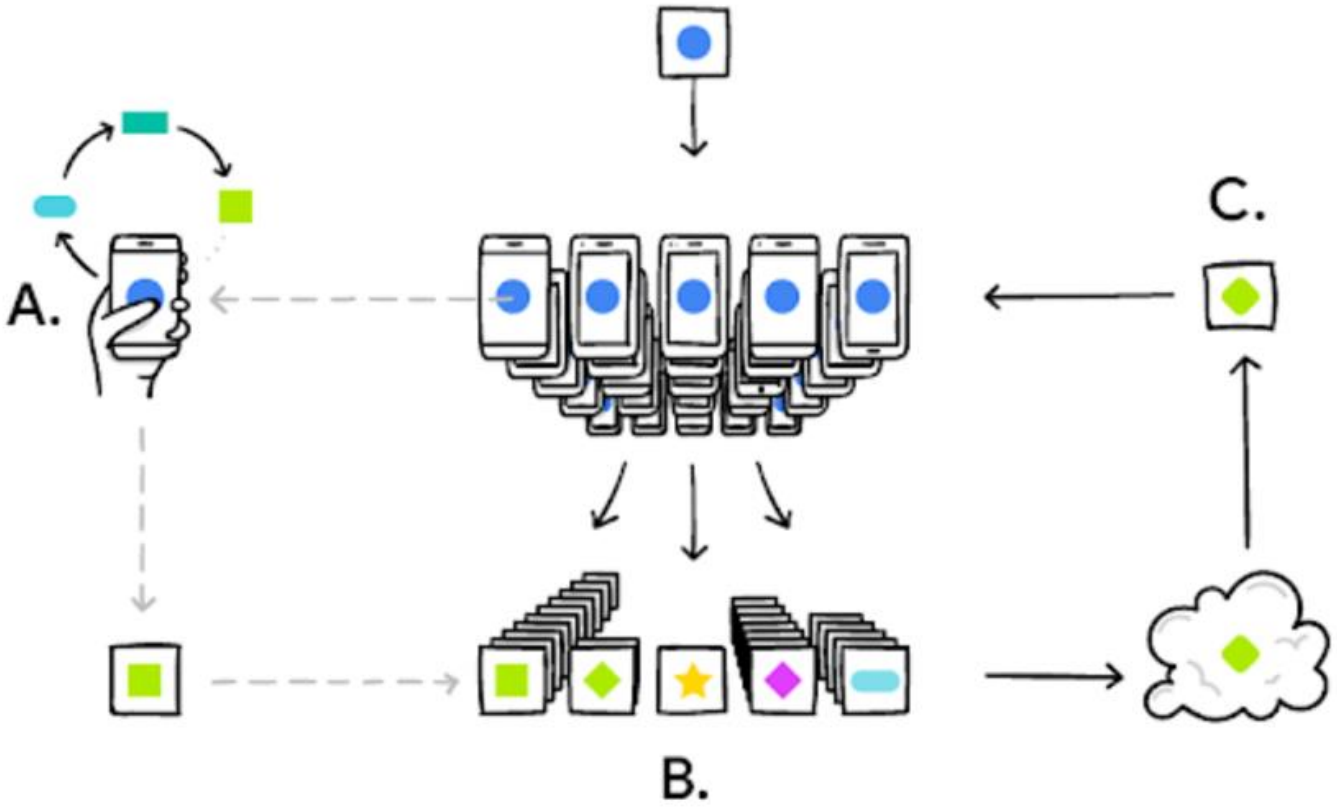
- Asynchronous stochastic gradient descent is a good option for training and has been shown to work well in practice, as long as gradient staleness is appropriately handled
- Async SGD implementations with a centralized parameter server may introduce a communication bottleneck. Utilizing N parameter servers, each handling an equal fraction of the total parameters is a conceptually straightforward solution to this problem.
- Finally, decentralized asynchronous stochastic gradient descent is a promising idea, though further research is required before we can conclusively recommend this over ‘standard’ async SGD

When to Use Distributed Deep Learning



- Data size and network size matter
- Small and shallow networks are not good candidates for distributed training as they don't have much computation per iteration. Networks with parameter sharing (such as CNNs and RNN) tend to be good candidates for distributed training: the amount of computation per parameter is much higher than, for example, a multi-layer perceptron

Federated Learning



Options

- Distributed Deep Learning with Spark 3.4
- Intel Analytics BigDL
- Yahoo TensorFlowOnSpark
- Distributed training with Keras
- Distributed training with TensorFlow
- Deep Learning Pipelines DataBricks