School of Computer Science
Carnegie Mellon

Probabilistic Graphical Models

Distributed Systems for ML

Qirong Ho

Lecture 22, April 10, 2017

$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$

An ML Program

$$\arg\max_{\vec{\theta}} \equiv \mathcal{L}(\{\mathbf{x}_i, \mathbf{y}i\}_{i=1}^{N} \; ; \; \vec{\theta}) + \Omega(\vec{\theta})$$

Model **Data** **Parameter**

Solved by an iterative convergent algorithm

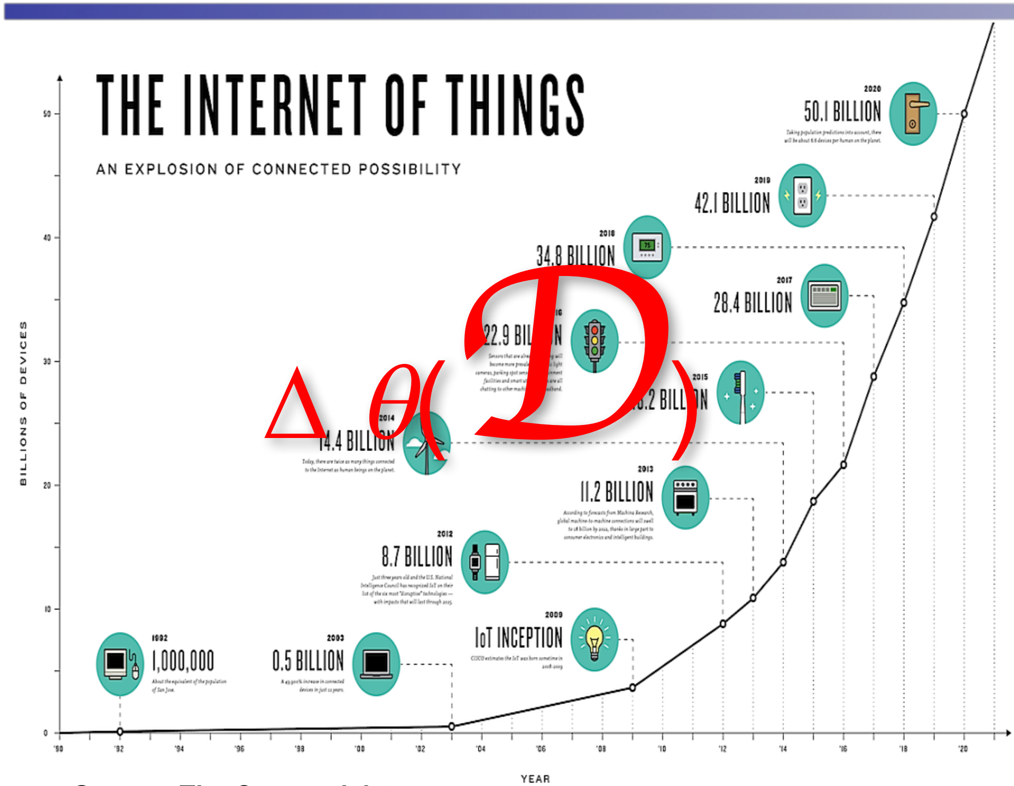
```
for (t = 1 to T) {
  doThings()
```

$$\vec{\theta}^{t+1} = g(\vec{\theta}^{t}, \; \underline{\Delta_f \vec{\theta}(\mathcal{D})})$$

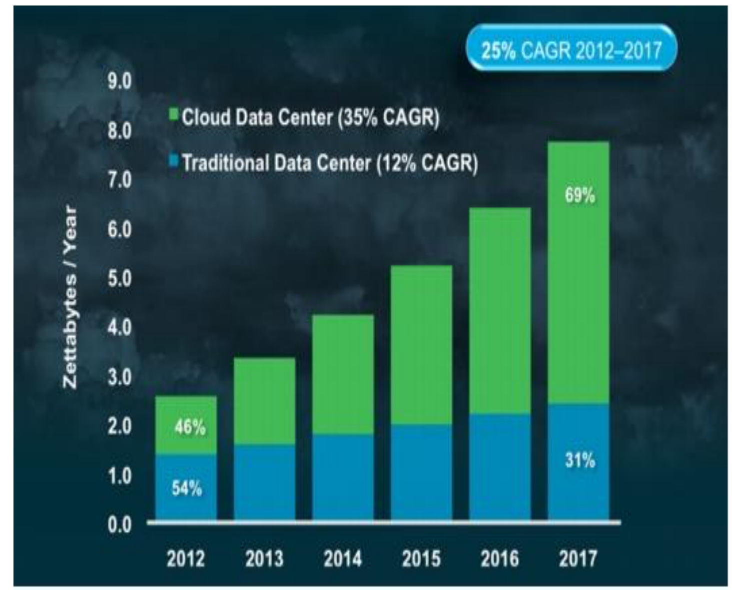
```
  doOtherThings()
}
```

This computation needs to be scaled up !

Challenge 1 – Massive Data Scale



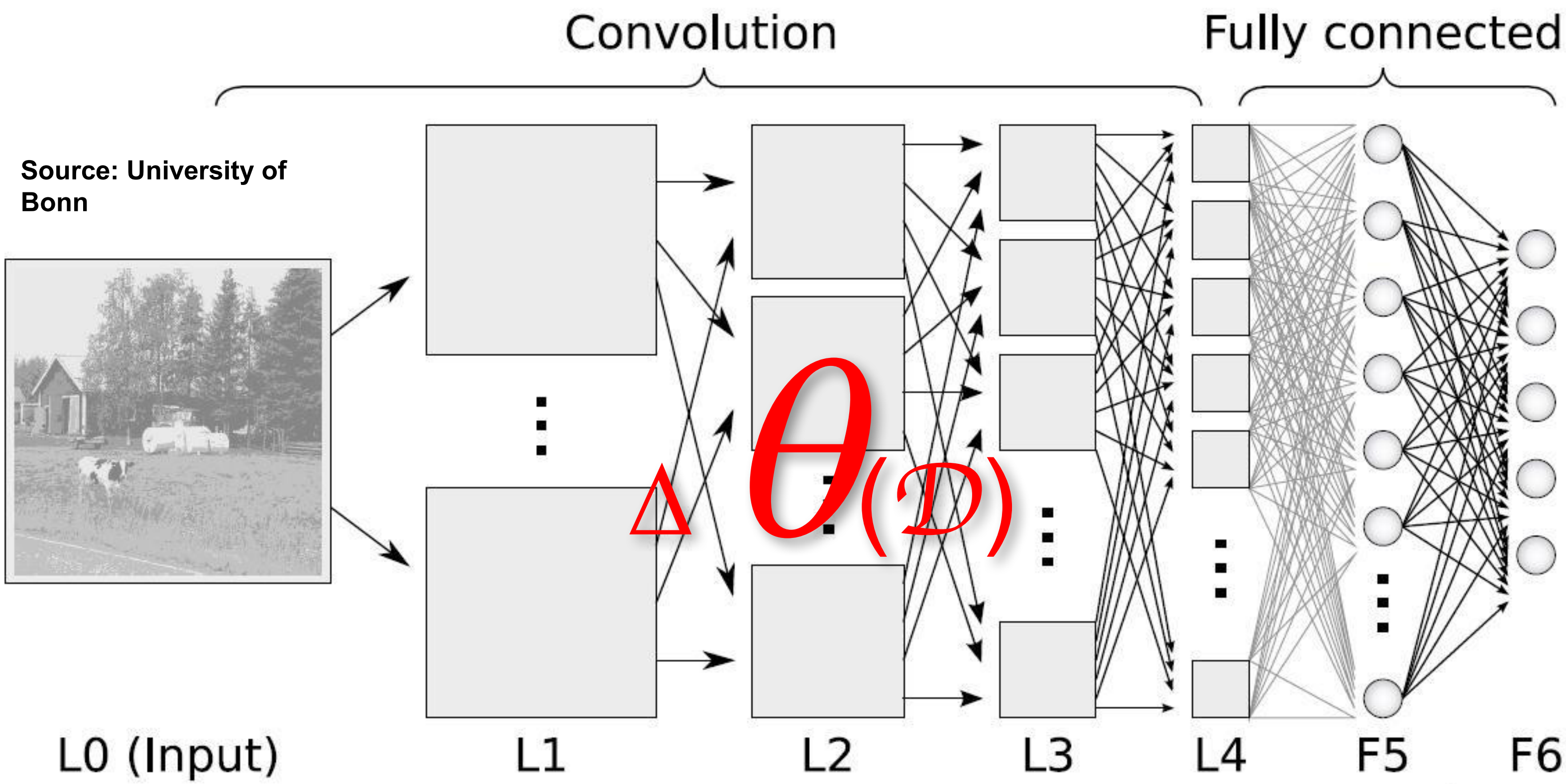
Source: The Connectivist



Source: Cisco Global Cloud Index

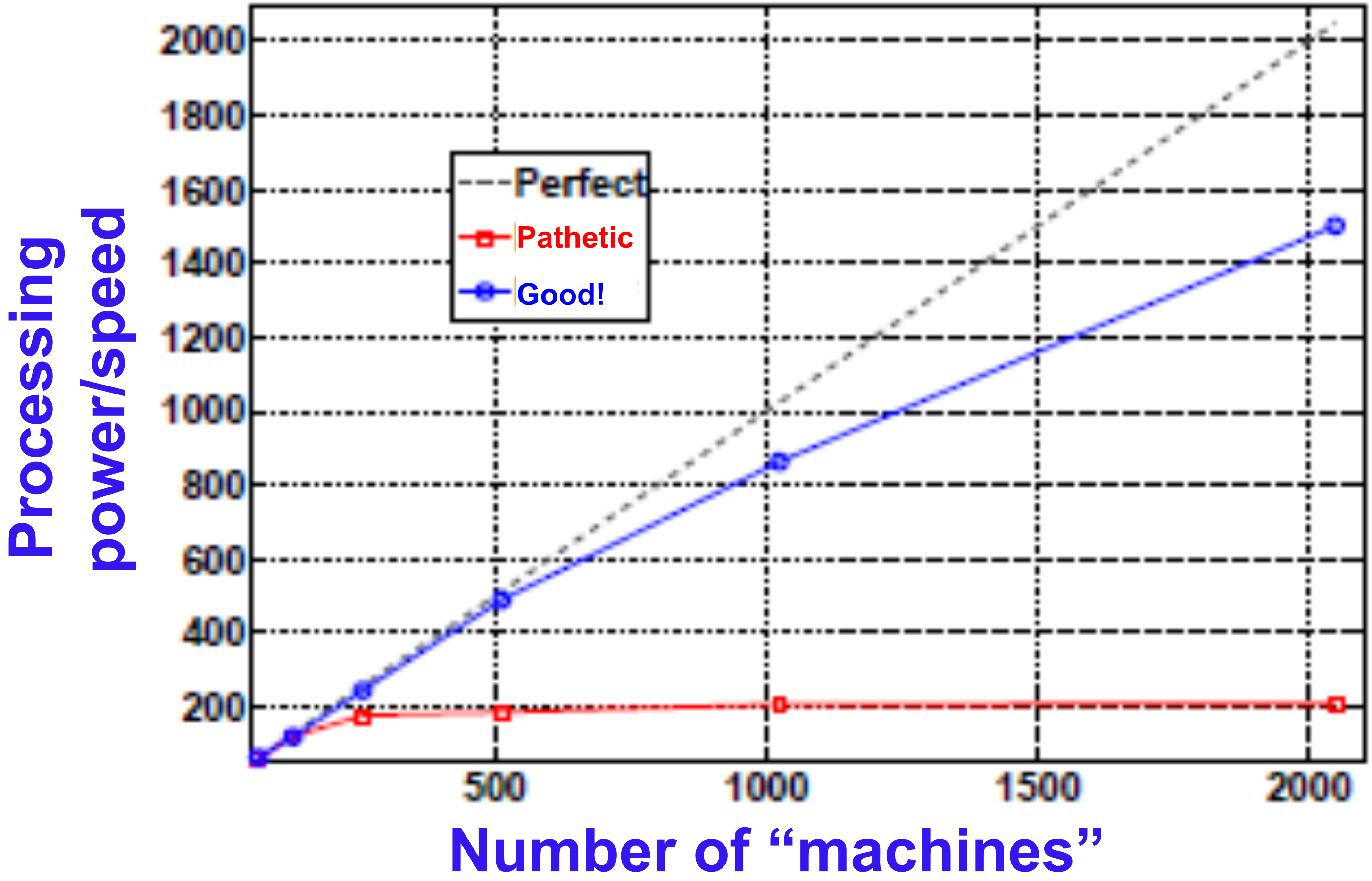
Familiar problem: data from 50B devices, data centers won't fit into memory of single machine

Challenge 2 – Gigantic Model Size

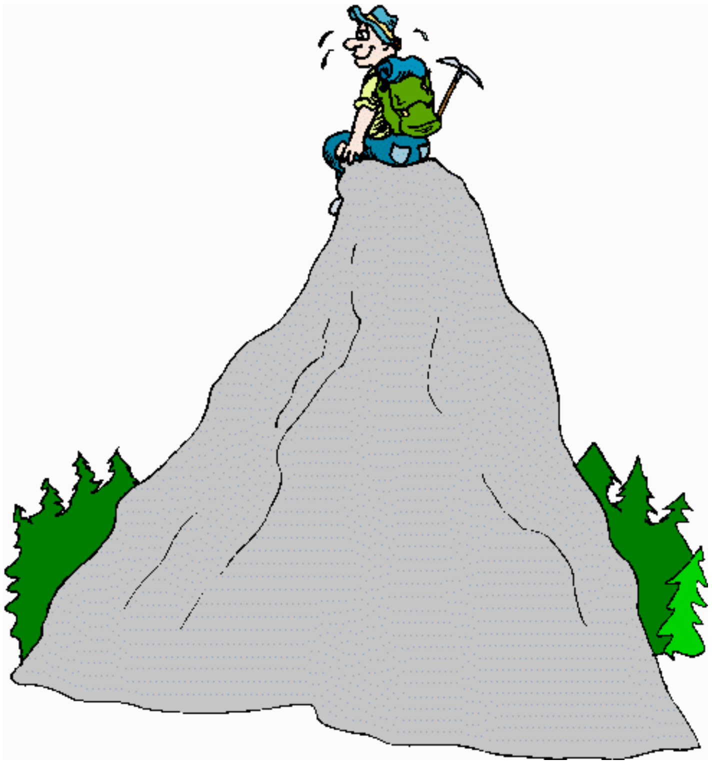


Maybe Big Data needs Big Models to extract understanding?
But models with >1 trillion params also won't fit!

The Scalability Challenge



ML Computation vs. Classical Computing Programs

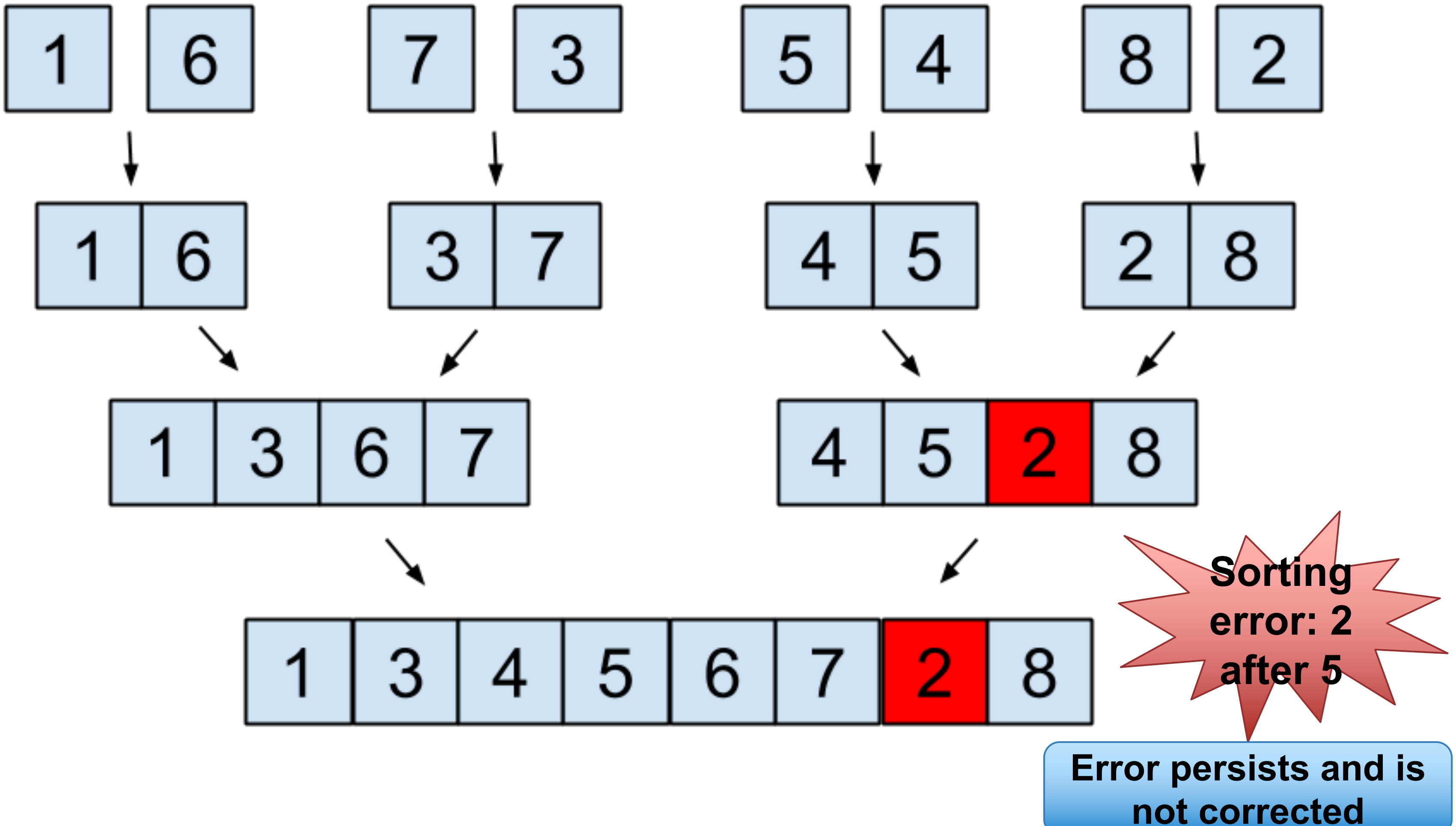


ML Program:
optimization-centric and iterative convergent

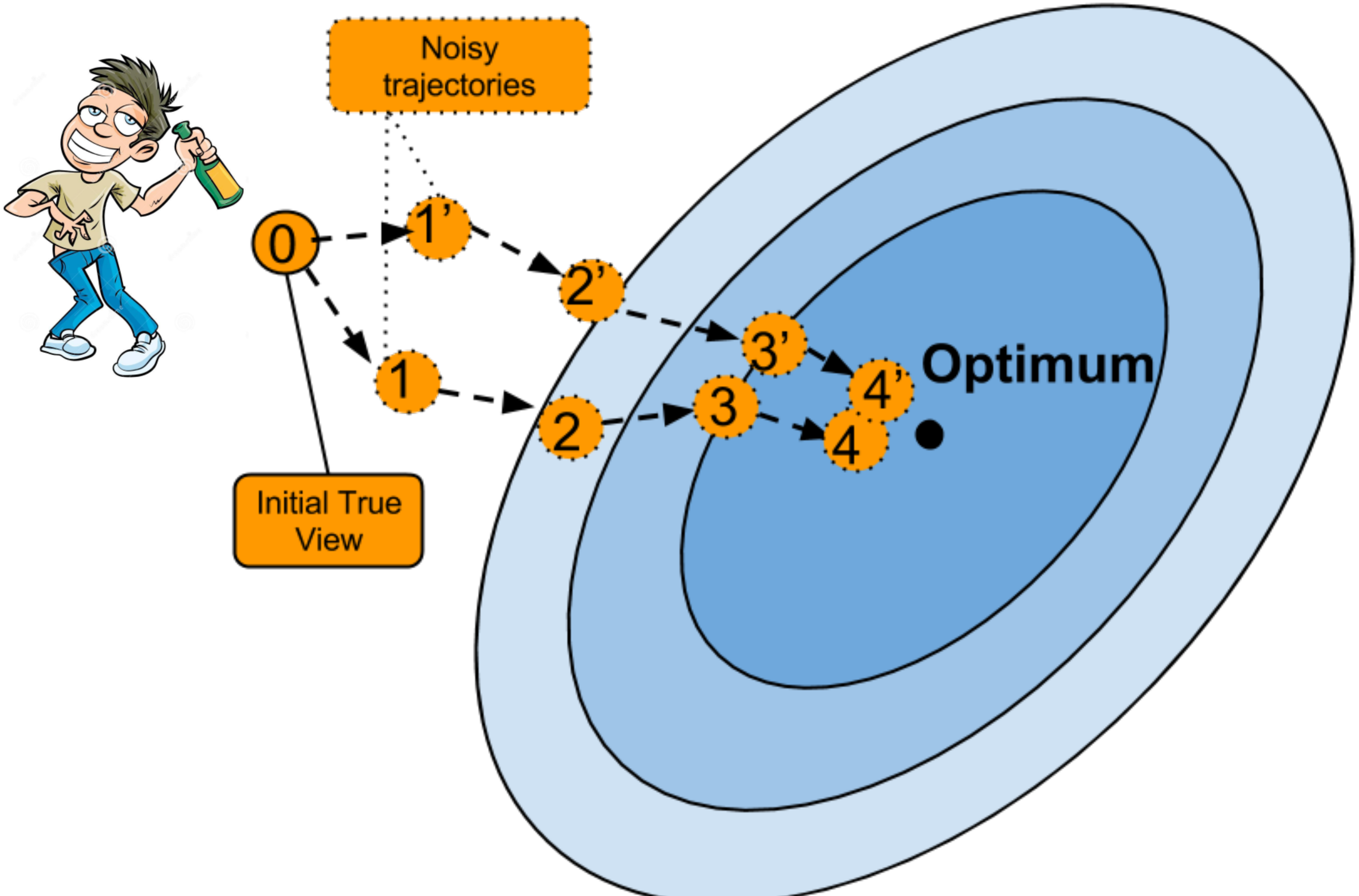
Traditional Program:
operation-centric and deterministic

Traditional Data Processing needs operational correctness …

Example: Merge sort



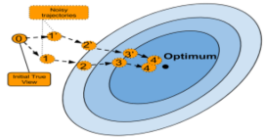
… but ML Algorithms can Self-heal



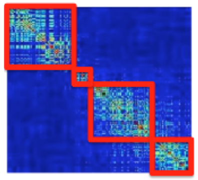
More Intrinsic Properties of ML Programs

- ML is **optimization-centric**, and admits an **iterative convergent** algorithmic solution rather than a one-step closed form solution

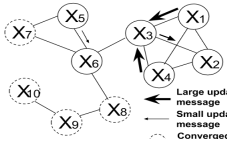
  - **Error tolerance**: often robust against limited errors in intermediate calculations



  - **Dynamic structural dependency**: changing correlations between model parameters critical to efficient parallelization



  - **Non-uniform convergence**: parameters can converge in very different number of steps



- Whereas traditional programs are **transaction-centric**, thus only guaranteed by **atomic correctness** at every step

An ML Program

$$\arg\max_{\vec{\theta}} \equiv \mathcal{L}(\{\mathbf{x}_i, \mathbf{y}i\}_{i=1}^{N} \; ; \; \vec{\theta}) + \Omega(\vec{\theta})$$

Model **Data** **Parameter**

Solved by an iterative convergent algorithm

```
for (t = 1 to T) {
  doThings()
```

$$\vec{\theta}^{t+1} = g(\vec{\theta}^{t}, \underbrace{\Delta_f \vec{\theta}(\mathcal{D})}_{})$$

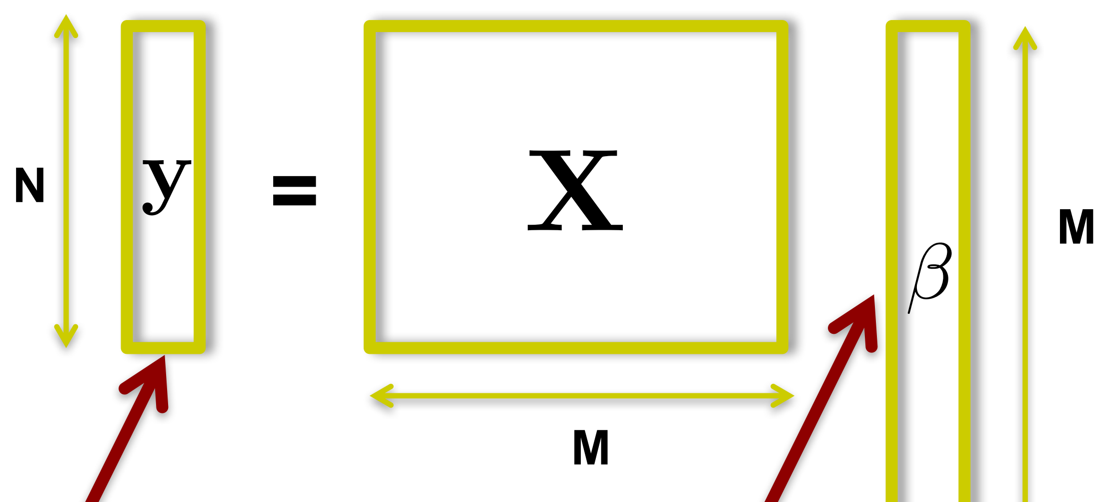
```
  doOtherThings()
}
```

This computation needs to be parallelized!

Challenge

- **Optimization programs:**

$$\Delta \leftarrow \sum_{i=1}^{N} \left[\frac{d}{d\theta_1}, \ldots, \frac{d}{d\theta_M}\right] f(\mathbf{x}_i, \mathbf{y}_i; \vec{\theta})$$



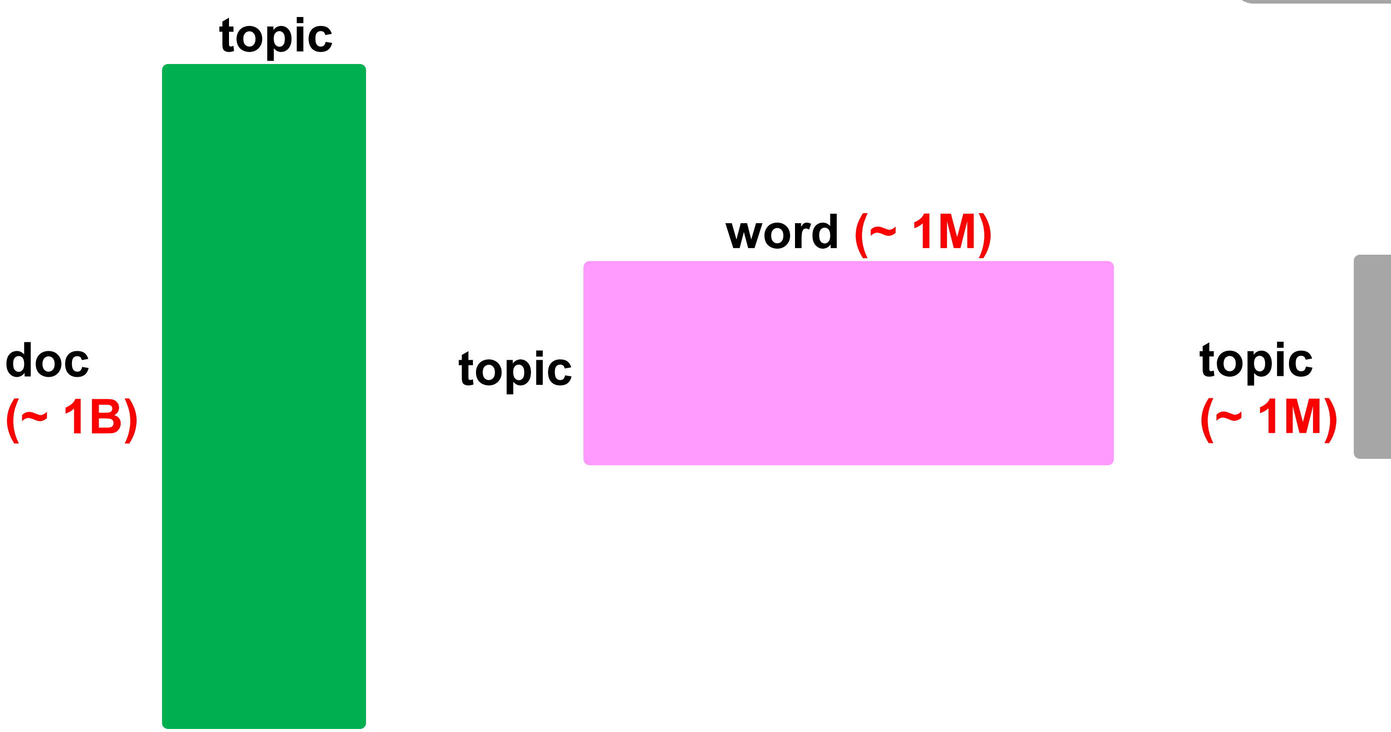
A huge number of parameters
(e.g.) M = 1B

A huge volume of data
(e.g.) N = 1B

Challenge

- **Probabilistic programs**

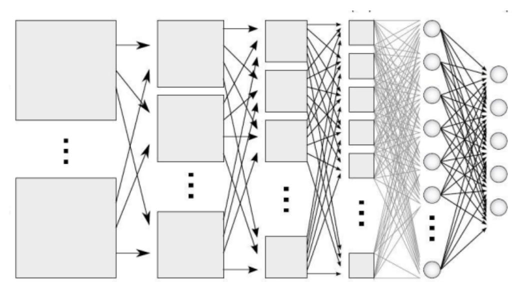
$$z_{ij} \sim p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\boxed{\delta}_{ik} + \alpha_k) \cdot \frac{\beta_{x_{ij}} + \boxed{B}_{k,x_{ij}}}{V\beta + \boxed{\sum_{v=1}^{V} B_{k,v}}}$$



Parallelization Strategies

$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

New Model = Old Model + Update(Data)



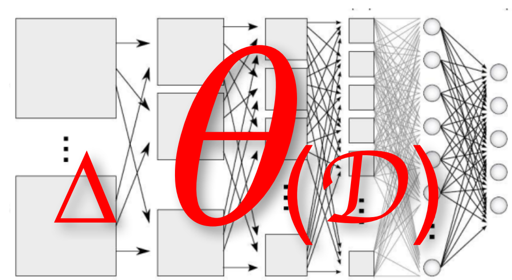
Data Parallel

$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$

Parallelization Strategies

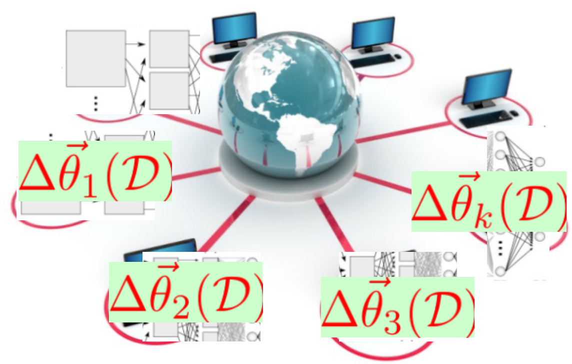
$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

New Model = Old Model + Update(Data)



Data Parallel

Model Parallel



$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$

$$\vec{\theta} \equiv [\vec{\theta}_1^{\mathrm{T}}, \vec{\theta}_2^{\mathrm{T}}, \ldots, \vec{\theta}_k^{\mathrm{T}}\}^{\mathrm{T}}$$

Recap from Distributed Algos

- Many parallel algorithms for Optimization, MCMC
- Common parallelization themes
  - **Embarrassingly parallel:** combine results from multiple independent problems, e.g. PSGD, EP-MCMC
  - **Stochastic over data:** approximate functions/ gradients with expectation over subset of data, then parallelize over data subsets, e.g. SGD
  - **Model-parallel:** parallelize over model variables, e.g. Coordinate Descent
  - **Auxiliary variables:** decompose problem by decoupling dependent variables, e.g. ADMM, Auxiliary Variable MCMC
- Considerations
  - **Regularizers, model structure:** may need sequential proximal or projection step, e.g. Stochastic Proximal Gradient
  - **Data partitioning:** for data-parallel, how to split data over machines?
  - **Model partitioning:** for model-parallel, how to split model over machines? Need to be careful as model variables are not necessarily independent of each other.

Implementing Distributed ML Algorithms

- Distributed ML requires care
- If not careful, slower than single machine!
  - Non-trivial systems bottlenecks (load imbalance, network bandwidth & latency)
- Even if algorithm is theoretically sound and has attractive properties, still need to pay attention to system aspects
  - Bandwidth (comms volume limits)
  - Latency (comms timing limits)
  - Data and Model partitioning (machine memory limitation, also affects comms volume)
  - Data and Model scheduling (affects convergence rate, comms volume & timing)
  - **Non-ideal systems behavior:** uneven machine performance, other cluster users

Implementing Distributed ML Algorithms

- Ad-hoc and partial solutions can lack theoretical analysis
  - **Major barrier:** hard to analyze solutions because algorithm/systems sometimes not fully/transparently described in papers
  - **Possible solution:** a universal language and principles for design could facilitate theoretical analysis of existing and new solutions

- Let us look at some software, which distributed ML algorithms can be implemented upon

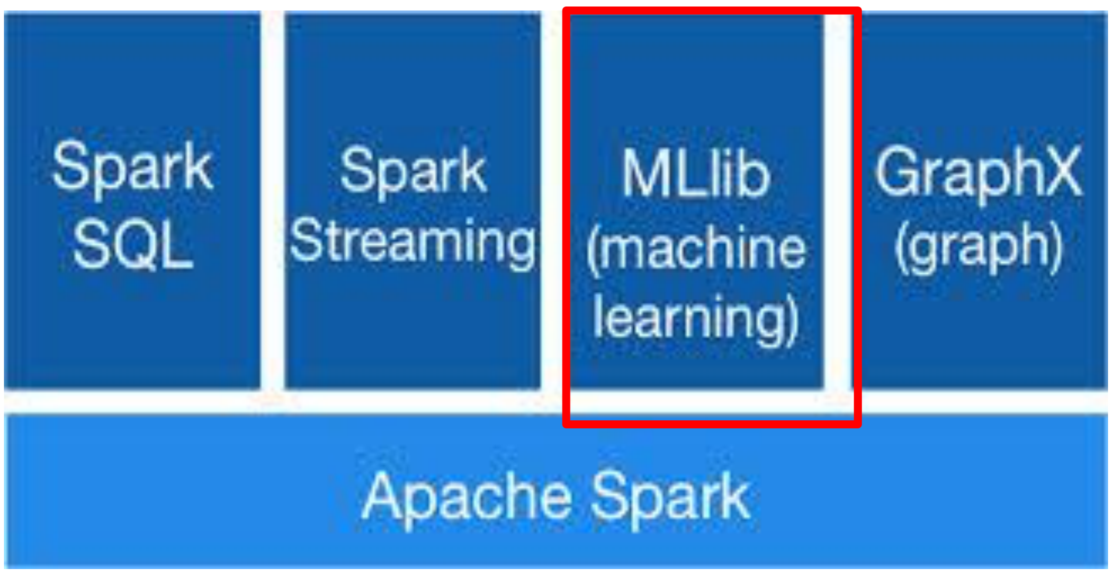
Software to Implement Distributed ML

History of Systems for Big ML

- Data-, model-parallel ML algorithms for optimization, MCMC
- One could write distributed implementations from scratch
- Perhaps better to use an existing open source platform?

Spark Overview [Zaharia et al., 2010]

- General-purpose system for Big Data processing
  - Shell/interpreter for Matlab/R-like analytics
- MLlib = Spark's ready-to-run ML library
  - Implemented on Spark's API

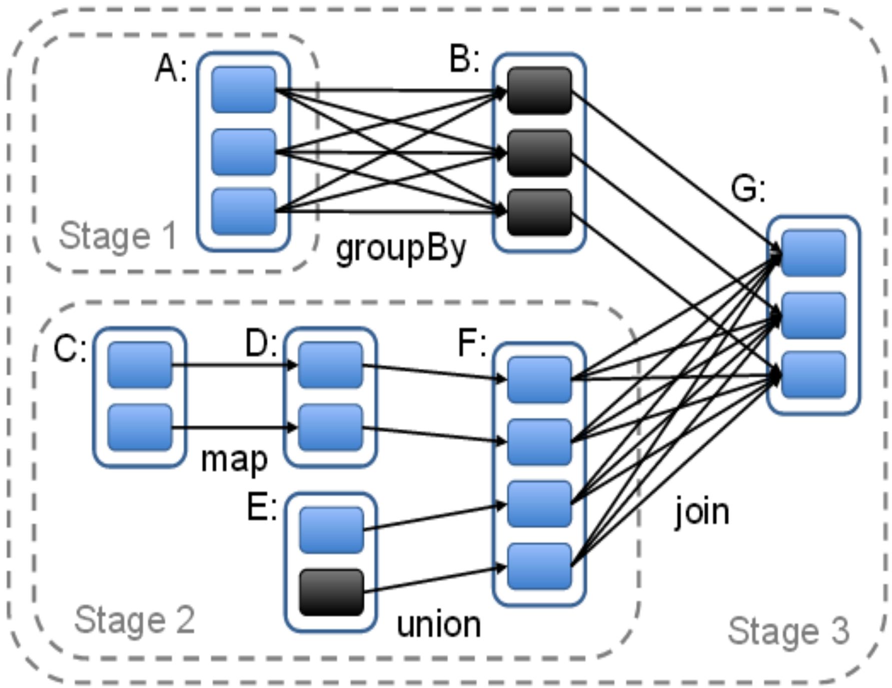


Spark Overview [Zaharia et al., 2010]

- MLlib algorithms (v1.4)
  - Classification and regression
    - linear models (SVMs, logistic regression, linear regression)
    - naive Bayes
    - decision trees
    - ensembles of trees (Random Forests and Gradient-Boosted Trees)
    - isotonic regression
  - Collaborative filtering
    - alternating least squares (ALS)
  - Clustering
    - k-means
    - Gaussian mixture
    - power iteration clustering (PIC)
    - latent Dirichlet allocation (LDA)
    - streaming k-means
  - Dimensionality reduction
    - singular value decomposition (SVD)
    - principal component analysis (PCA)

Spark Overview [Zaharia et al., 2010]

- Key feature: Resilient Distributed Datasets (RDDs)
  - Data processing = lineage graph of transforms
  - RDDs = nodes
  - Transforms = edges



Source: Zaharia et al. (2012)

Spark Overview [Zaharia et al., 2010]

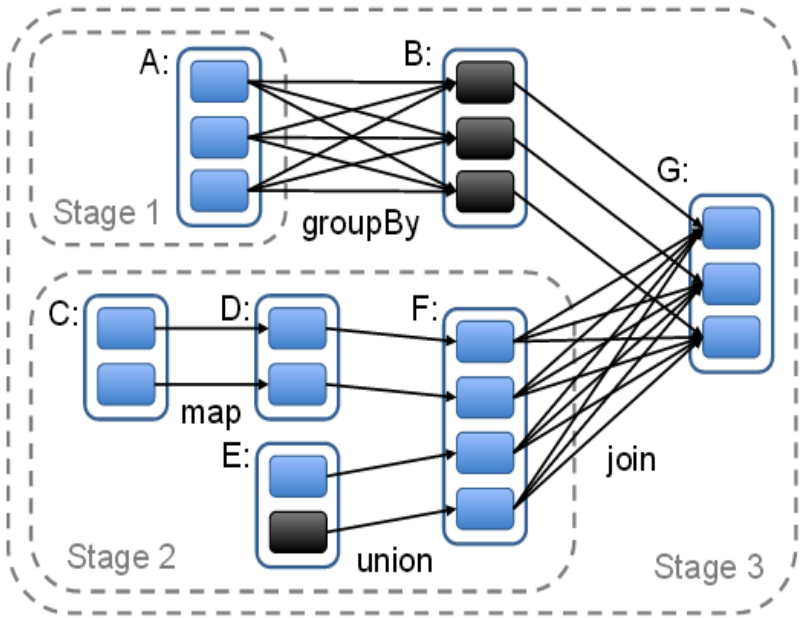
- RDD-based programming model
  - Similar in spirit to Hadoop Mapreduce
  - Functional style: manipulate RDDs via “transformations”, “actions”
    - E.g. map is a transformation, reduce is an action
  - Example: load file, count total number of characters

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

  - Other transformations and actions:
    - union(), intersection(), distinct()
    - count(), first(), take(), foreach()
    - …
  - Can specify if an RDD should be “persisted” to disk
    - Allows for faster recovery during cluster faults

Spark Overview [Zaharia et al., 2010]

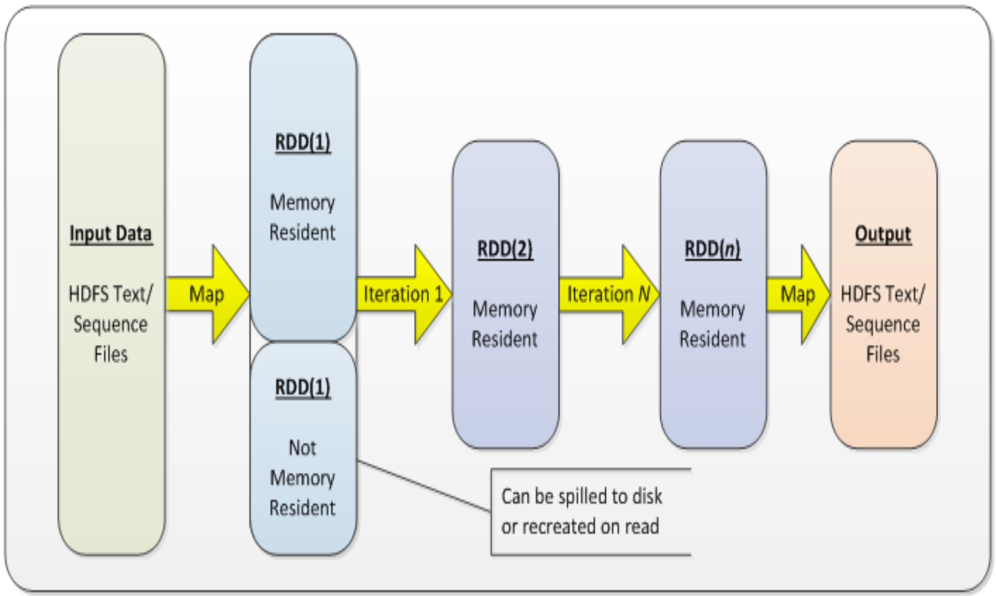
- Benefits of Spark:
  - Fault tolerant - RDDs immutable, just re-compute from lineage
  - Cacheable - keep some RDDs in RAM
    - Faster than Hadoop MR at iterative algorithms
  - Supports MapReduce as special case

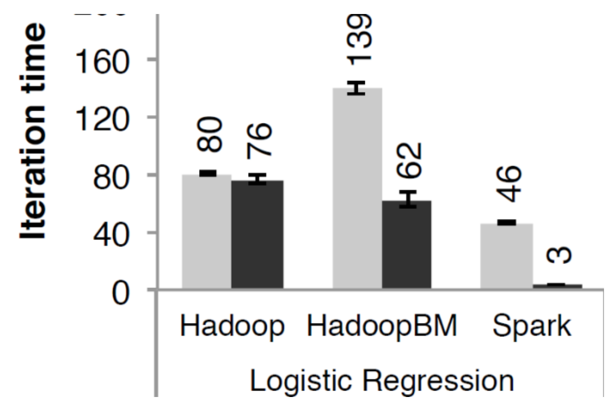


Source: Zaharia et al. (2012)

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



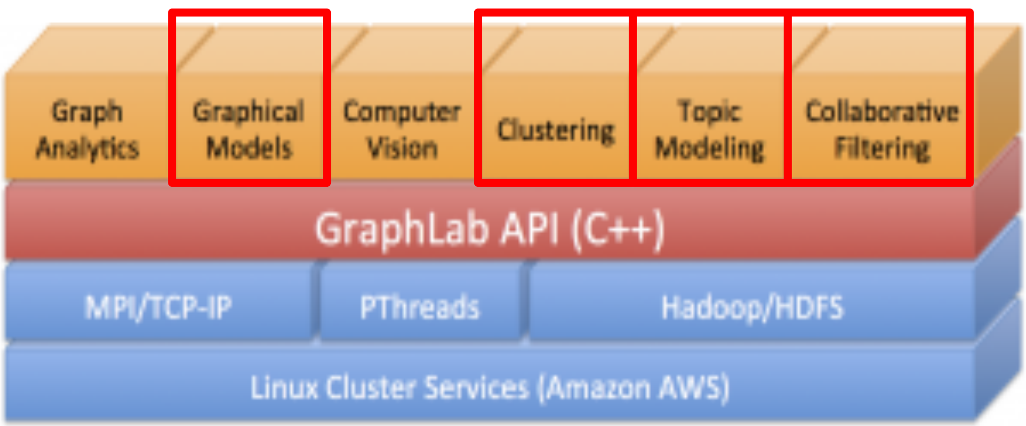


- **Spark ML library supports data-parallel ML algos, like Hadoop**
  - Spark and Hadoop: comparable first iter timings…
  - But Spark's later iters are much faster

Source: ebaytechblog.com

GraphLab Overview [Low et al., 2012]

- Known as “GraphLab PowerGraph v2.2”
  - Different from commercial software “GraphLab Create” by Dato.com, who formerly developed PowerGraph v2.2
- System for Graph Programming
  - Think of ML algos as graph algos
- Comes with ready-to-run “toolkits”
  - ML-centric toolkits: clustering, collaborative filtering, topic modeling, graphical models



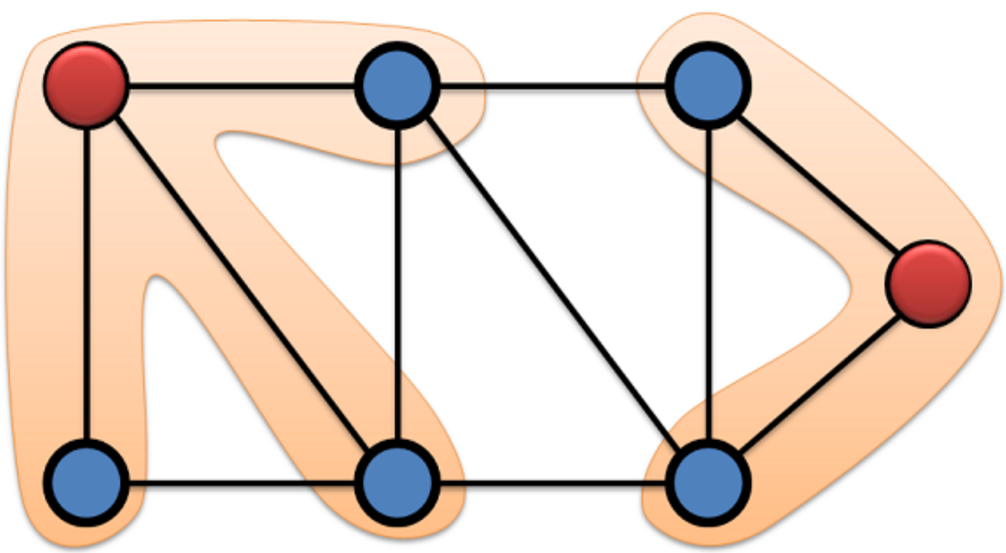
GraphLab Overview [Low et al., 2012]

- ML-related toolkits
  - Clustering
    - K-means
    - Spectral
  - Collaborative Filtering
    - Matrix Factorization (including Non-negative, L1/L2-regularized)
  - Graphical Models
    - Factor graphs
    - Belief propagation algorithm
  - Topic Modeling
    - LDA

- Other toolkits available for computer vision, graph analytics, linear systems

GraphLab Overview [Low et al., 2012]

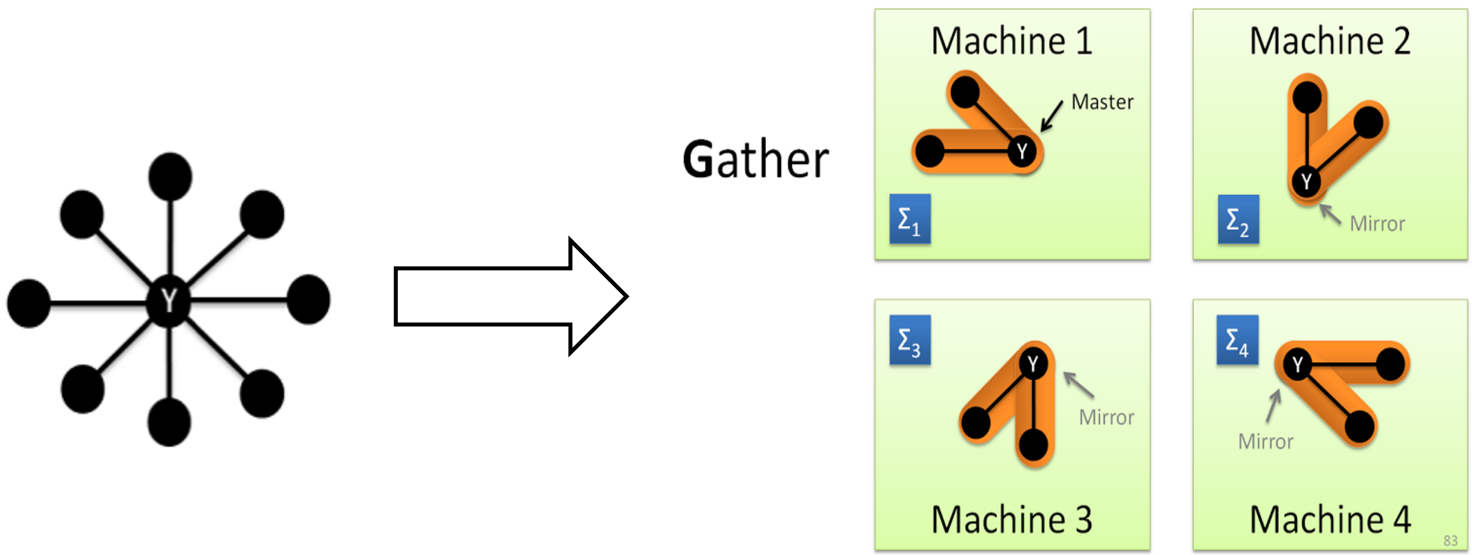
- Key feature: Gather-Apply-Scatter Programming Model
  - Write ML algos as vertex programs
  - Run vertex programs in parallel on each graph node
  - Graph nodes, edges can have data, parameters



Source: Gonzalez (2012)

GraphLab Overview [Low et al., 2012]

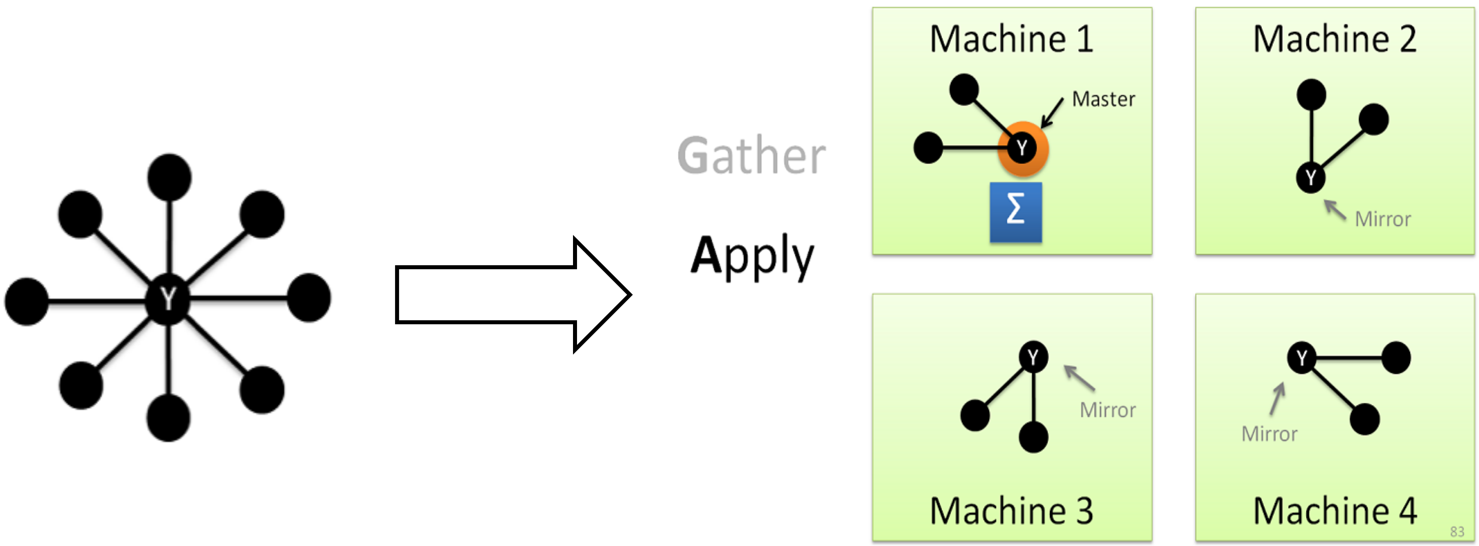
- Programming Model: GAS Vertex Programs
  - **1) Gather():** Accumulate data, params from my neighbors + edges
  - 2) Apply(): Transform output of Gather(), write to myself
  - 3) Scatter(): Transform output of Gather(), Apply(), write to my edges



Source: Gonzalez (2012)

GraphLab Overview [Low et al., 2012]

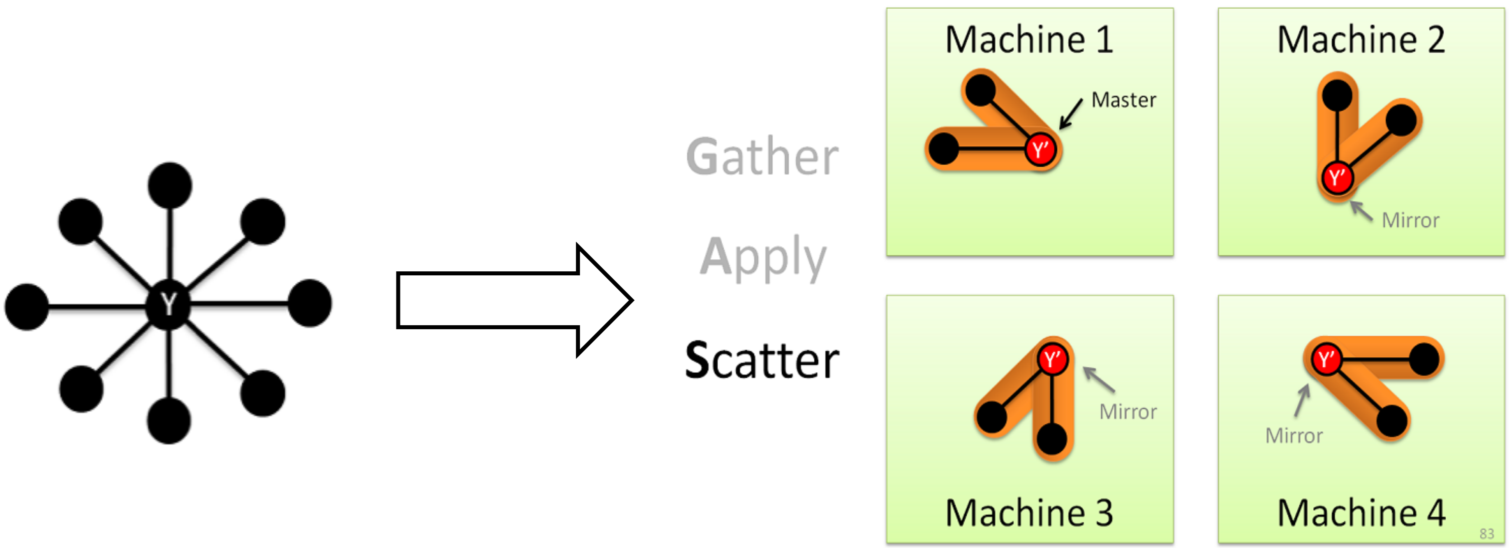
- Programming Model: GAS Vertex Programs
  - 1) Gather(): Accumulate data, params from my neighbors + edges
  - **2) Apply():** Transform output of Gather(), write to myself
  - 3) Scatter(): Transform output of Gather(), Apply(), write to my edges



Source: Gonzalez (2012)

GraphLab Overview [Low et al., 2012]

- Programming Model: GAS Vertex Programs
  - 1) Gather(): Accumulate data, params from my neighbors + edges
  - 2) Apply(): Transform output of Gather(), write to myself
  - 3) **Scatter():** Transform output of Gather(), Apply(), write to my edges



Source: Gonzalez (2012)

GraphLab Overview [Low et al., 2012]

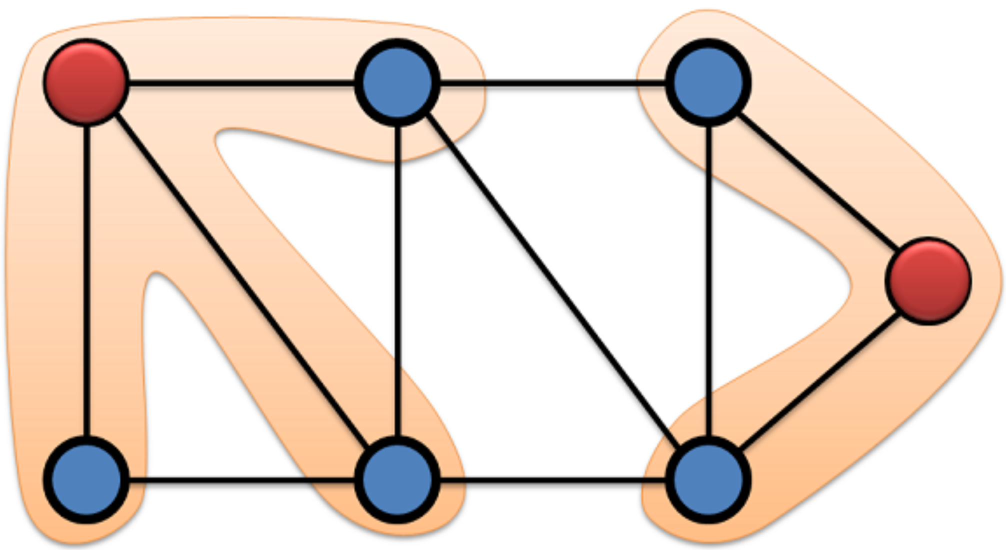
- Example GAS program: Pagerank
  - Programmer implements gather(), apply(), scatter() functions

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
   return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
   rnew = 0.15 + 0.85 * acc
   D_u.delta = (rnew - D_u.rank)/
               #outNbrs(u)
   D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u,D_(u,v),D_v):
   if(|D_u.delta|>ε) Activate(v)
   return delta
```

Source: Gonzalez et al. (OSDI 2012)

GraphLab Overview [Low et al., 2012]

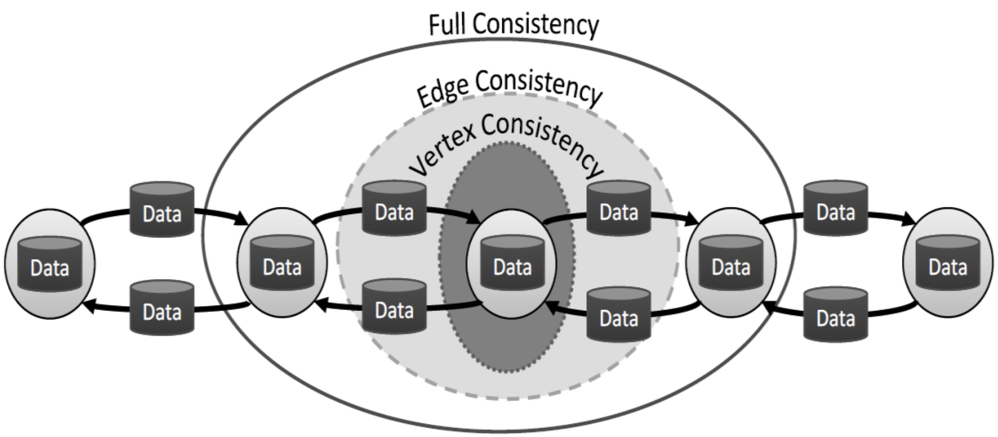
- Benefits of Graphlab
  - Supports asynchronous execution - fast, avoids straggler problems
  - Edge-cut partitioning - scales to large, power-law graphs
  - Graph-correctness - for ML, more fine-grained than MapR-correctness

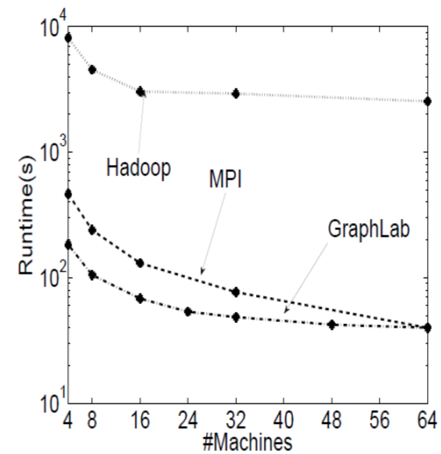


Source: Gonzalez (2012)

GraphLab: Model-Parallel via Graphs

- GraphLab **Graph consistency models**
  - Guide search for "ideal" model-parallel execution order
  - ML algo correct if input graph has all dependencies



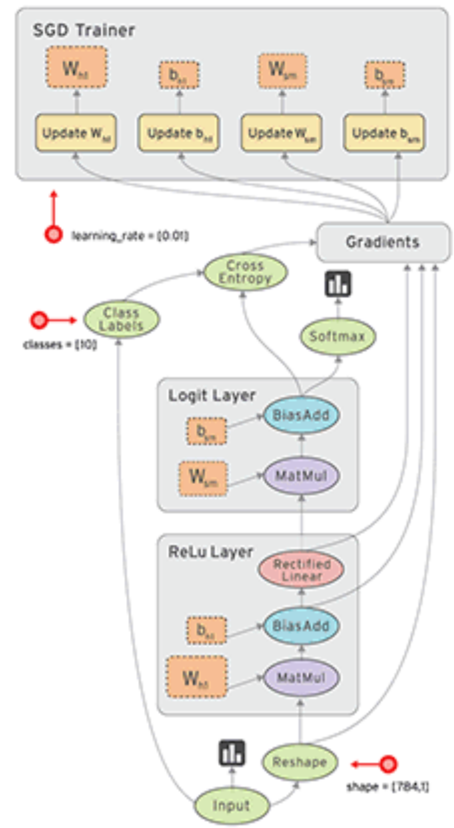


- GraphLab supports asynchronous (no-waiting) execution
  - Correctness enforced by graph consistency model
  - Result: GraphLab graph-parallel ML much faster than Hadoop

Source: Low et al. (2010)

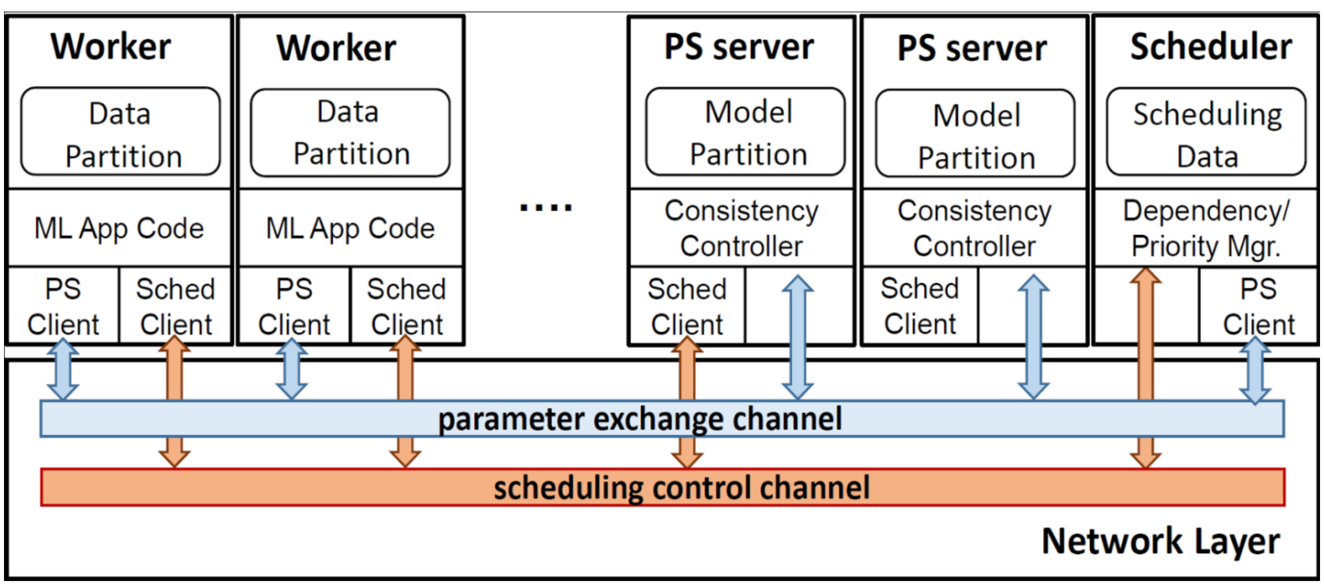
Google TensorFlow: Dataflow-style system

- First release Nov 2015
- Auto-differentiation + Dataflow system
  - Conceptually similar to Spark RDDs
  - Geared towards tensor/matrix computation
  - Asynchronous execution
- Distributed support is Work-in-Progress
  - Results are mixed, performance is OK for smaller models
  - Large models do not benefit from going distributed, e.g. VGG



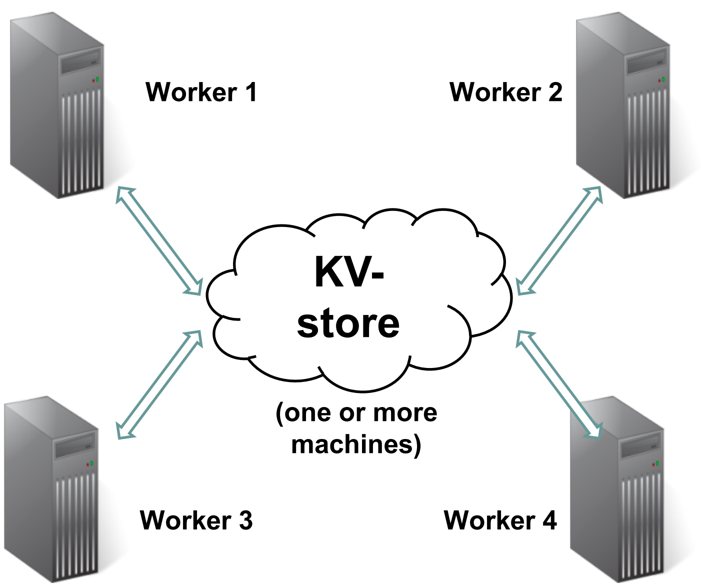
Parallel ML System Overview (Formerly Petuum) [Xing et al., 2015]

- Key modules
  - **Key-value store** (Parameter Server) for data-parallel ML algos
  - **Scheduler** for model-parallel ML algos
- Program ML algos in iterative-convergent style
  - ML algo = (1) write update equations + (2) iterate eqns via schedule



PMLS Overview [Xing et al., 2015]

- Key-Value store (Parameter Server)
  - Enables data-parallelism
  - A type of Distributed Shared Memory (DSM)
    - Model parameters globally shared across workers
  - Programming: replace local variables with PS calls



Single Machine

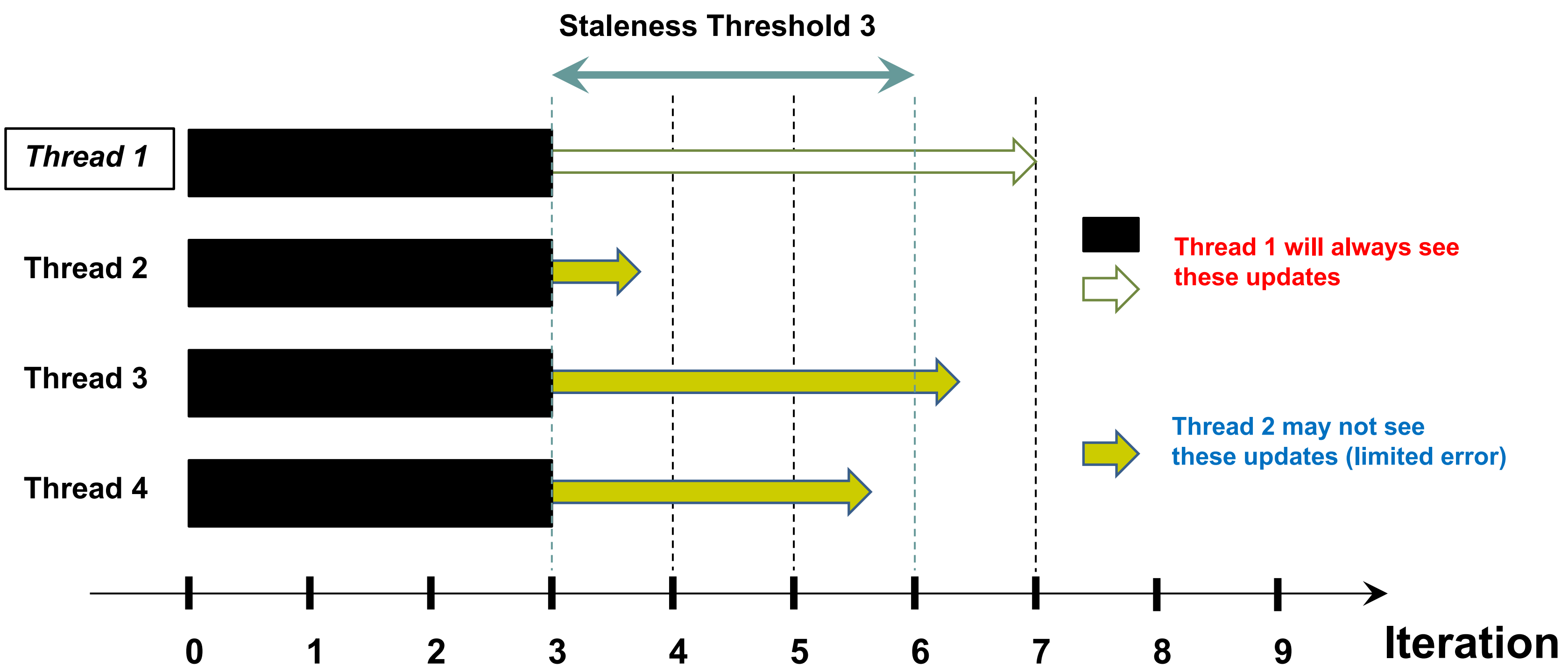
```
ProcessDataPoint(i) {
  for j = 1 to M {
    old = model[j]
    delta = f(model,data(i))
    model[j] += delta
  }
}
```

Distributed with PS

```
ProcessDataPoint(i) {
  for j = 1 to M {
    old = PS.read(model,j)
    delta = f(model,data(i))
    PS.inc(model,j,delta)
  }
}
```

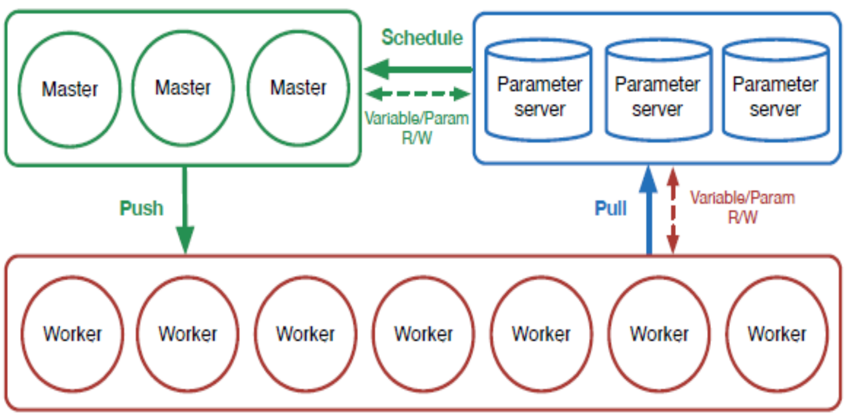
PMLS Overview [Xing et al., 2015]

- Key-Value store features:
  - ML-tailored consistency model: Stale Synchronous Parallel (SSP)
  - Asynchronous-like speed
  - Bulk Synchronous Parallel-like correctness guarantees for ML



PMLS Overview [Xing et al., 2015]

- Scheduler
  - Enables correct model-parallelism
  - Can analyze ML model structure for best execution order
  - Programming: schedule(), push(), pull() abstraction



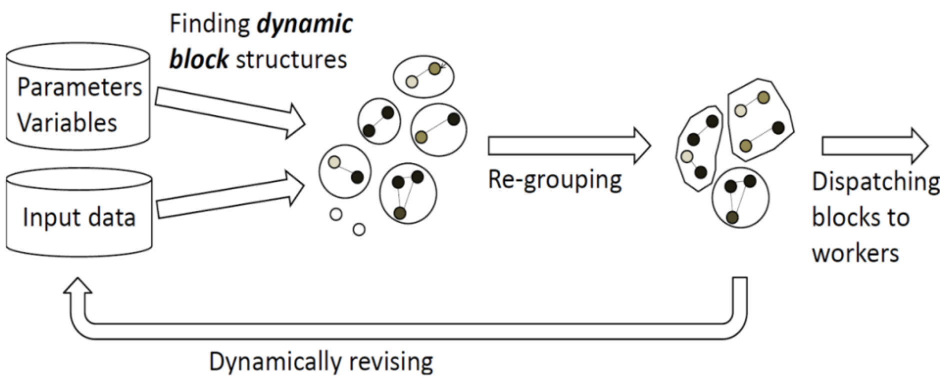

```
schedule() {
  // Select U vars x[j] to be sent
  // to the workers for updating
  ...
  return (x[j_1], ..., x[j_U])
}
```

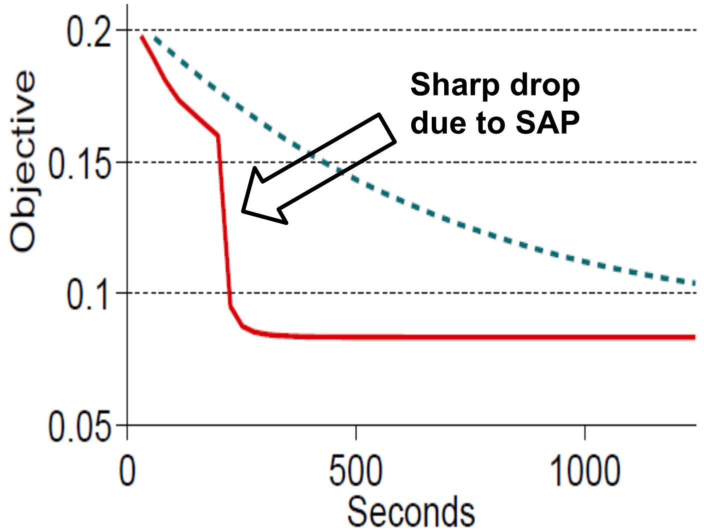
```
push(worker = p, vars = (x[j_1],...,x[j_U])) {
  // Compute partial update z for U vars x[j]
  // at worker p
  ...
  return z
}
```

```
pull(workers = [p], vars = (x[j_1],...,x[j_U]),
     updates = [z]) {
  // Use partial updates z from workers p to
  // update U vars x[j]. sync() is automatic.
  ...
}
```

PMLS Overview [Xing et al., 2015]

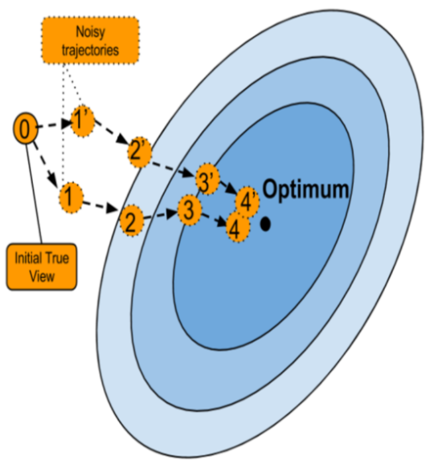
- Scheduler benefits:
  - ML scheduling engine: Structure-Aware Parallelization (SAP)
  - Scheduled ML algos require less computation to finish

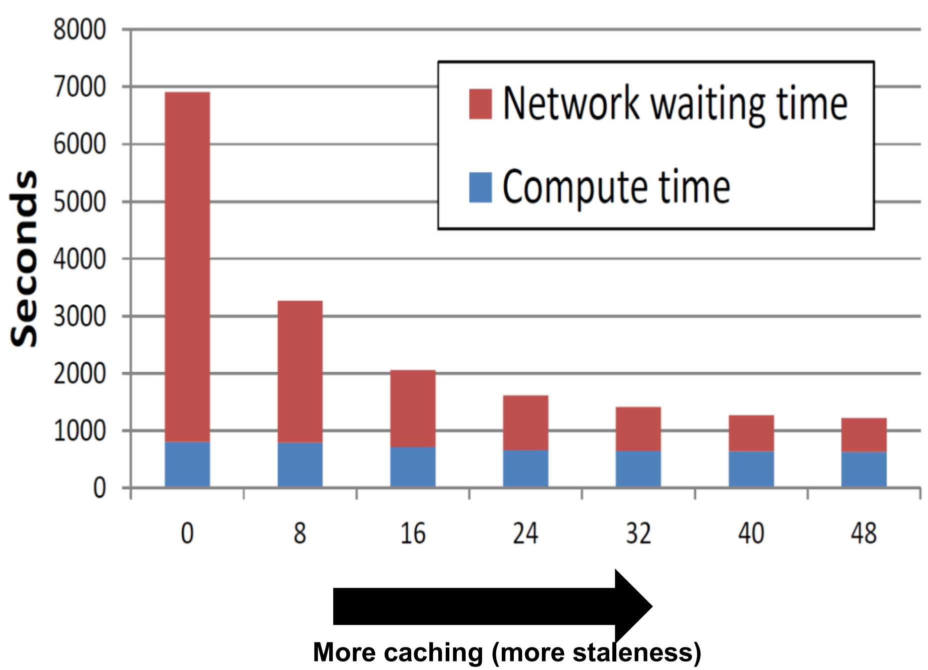




PMLS:
ML props = 1st-class citizen

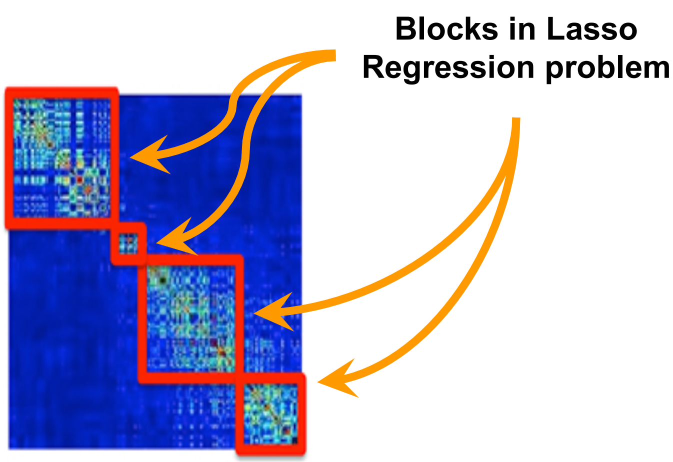
- Error tolerance via Stale Sync Parallel KV-store
  - System Insight 1: ML algos bottleneck on network comms
  - System Insight 2: More caching => less comms => faster execution

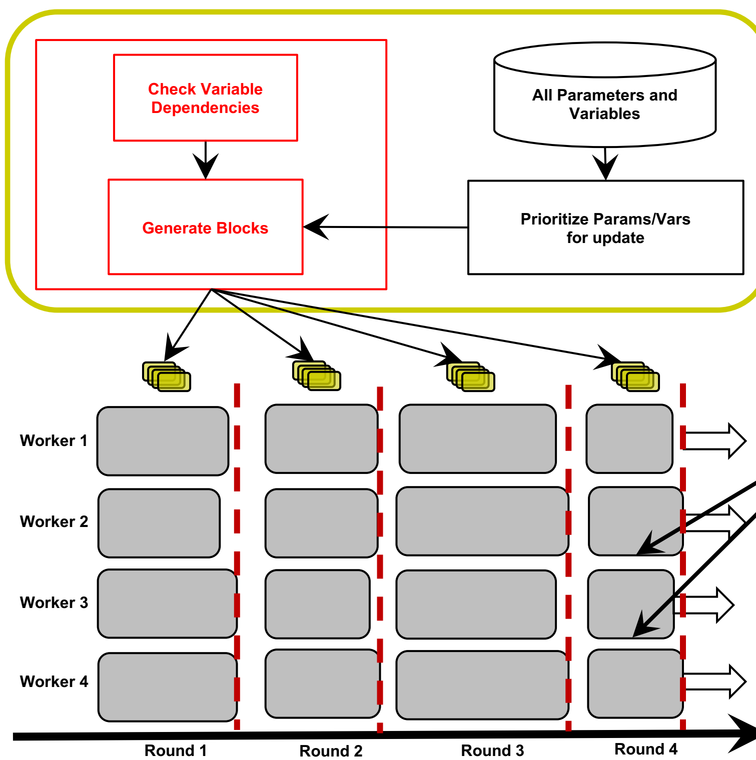




PMLS: ML props = 1st-class citizen

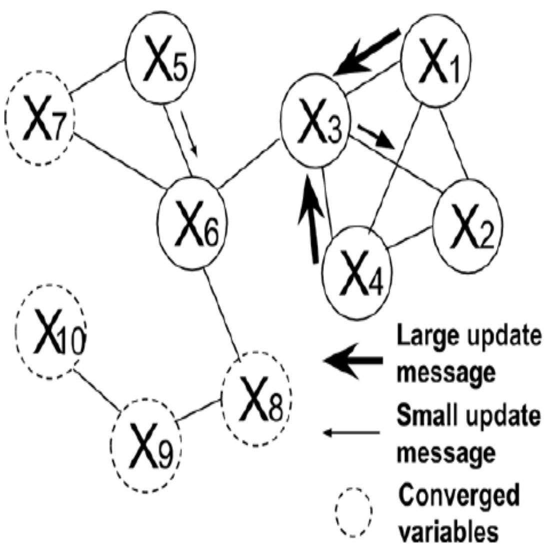
- Harness Block dependency structure via Scheduler
  - System Insight 1: Pipeline scheduler to hide latency
  - System Insight 2: Load-balance blocks to prevent stragglers

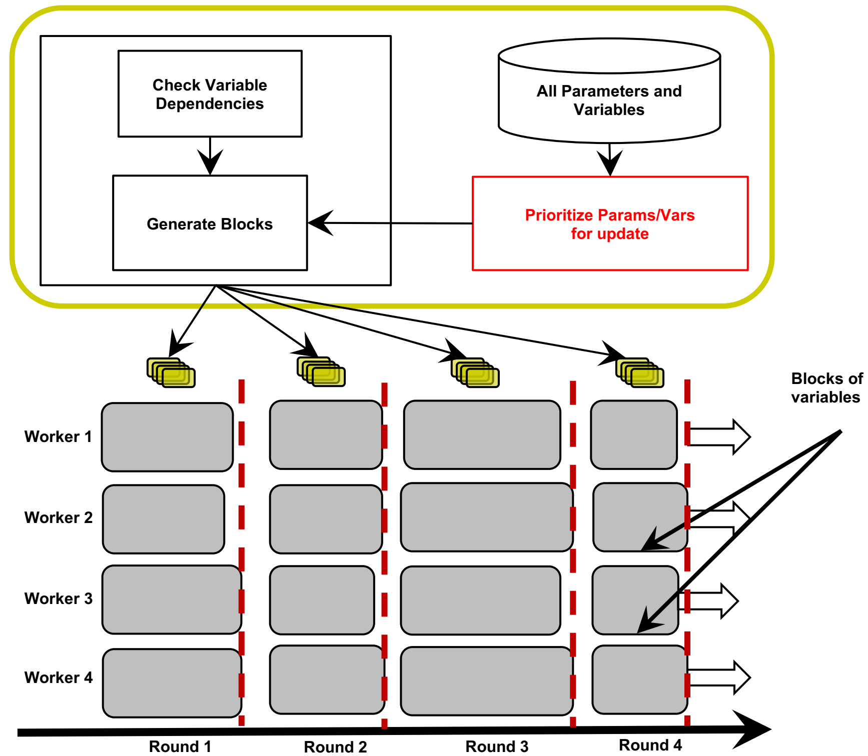




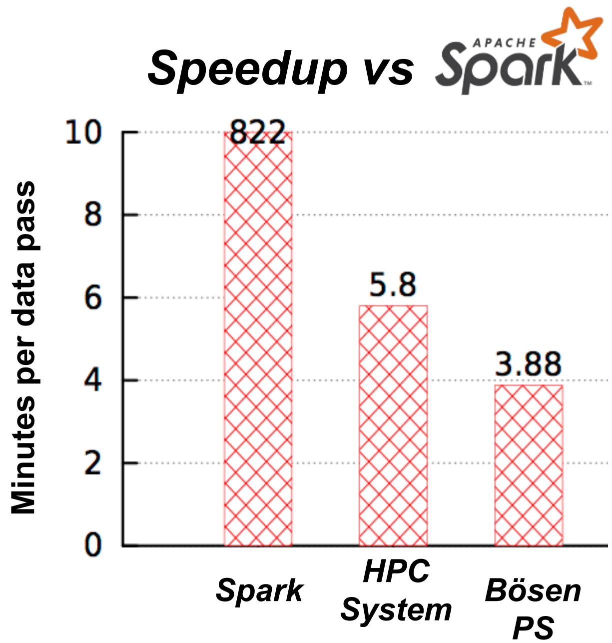
PMLS:
ML props = 1st-class citizen

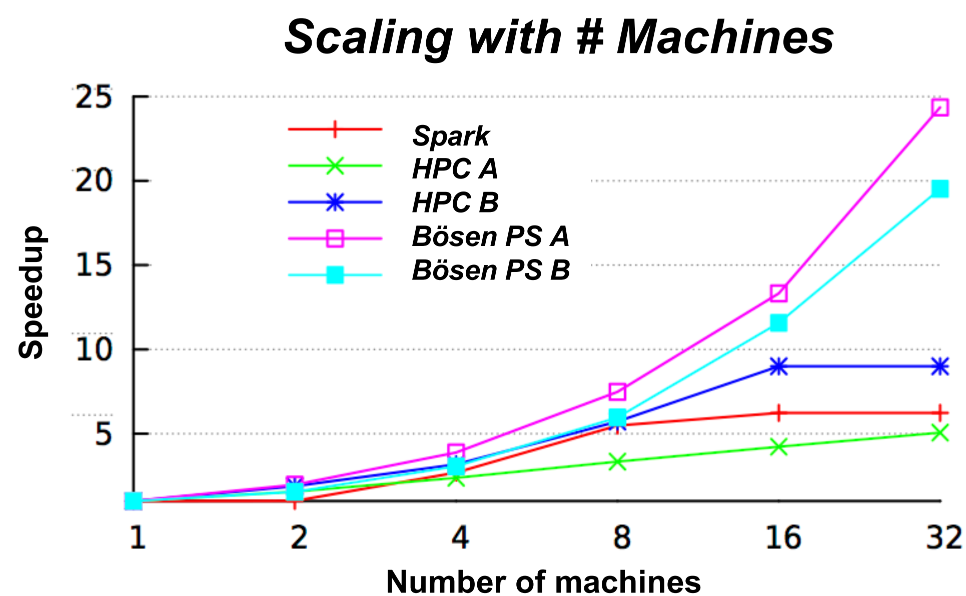
- Exploit Uneven Convergence via Prioritizer
  - System Insight 1: Prioritize small # of vars => fewer deps to check
  - System Insight 2: Lowers computational cost of Scheduling





PMLS Research in 2016: Parameter Servers



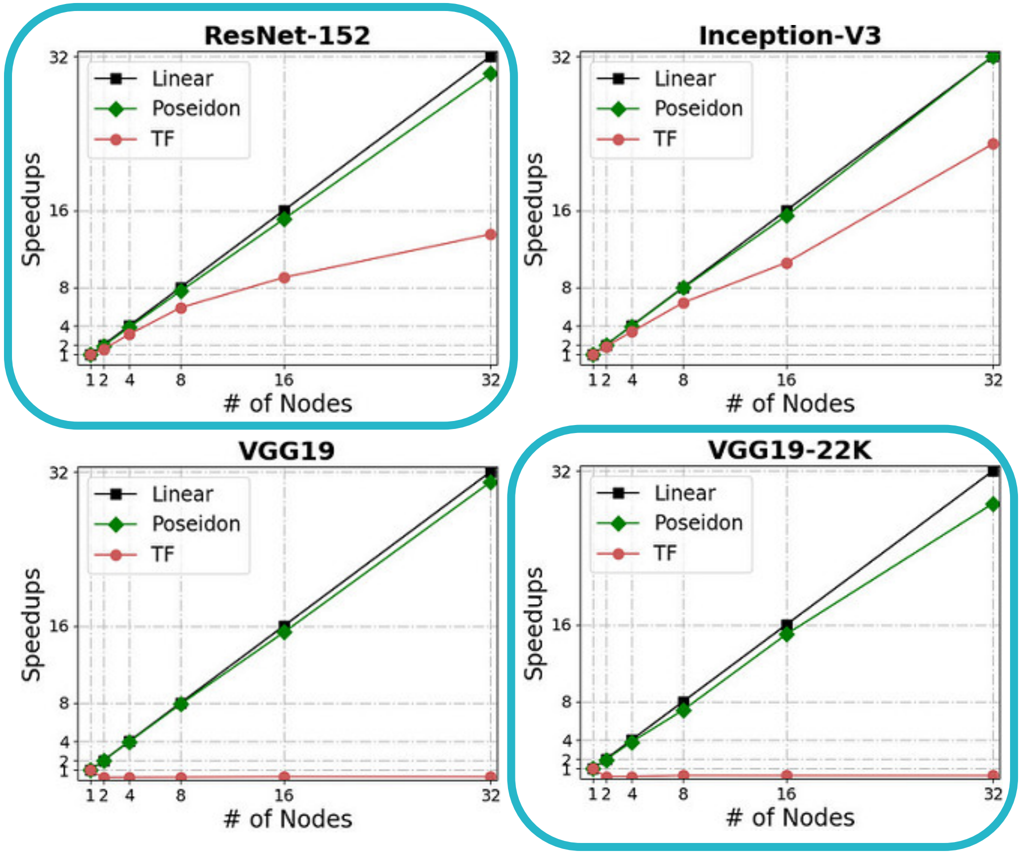


Task: SGD Matrix Factorization, 32 machines (16 cores, 32GB RAM), 250M parameters, 1.3GB data

PMLS Research in 2016: Deep Learning Systems

ILSVRC2015 winner
 # params: 60.2M

ILSVRC2013 winner
 # params: 60.2M



ILSVRC2013 winner
 # params: 143M
Most-adopted feature
Extraction network

ILSVRC2013 winner
 # params: 229M
Extended to 22K categories

ML Programming Interface: Needs and Considerations

- An ideal ML programming interface should make it easy to write correct data-parallel, model-parallel ML programs

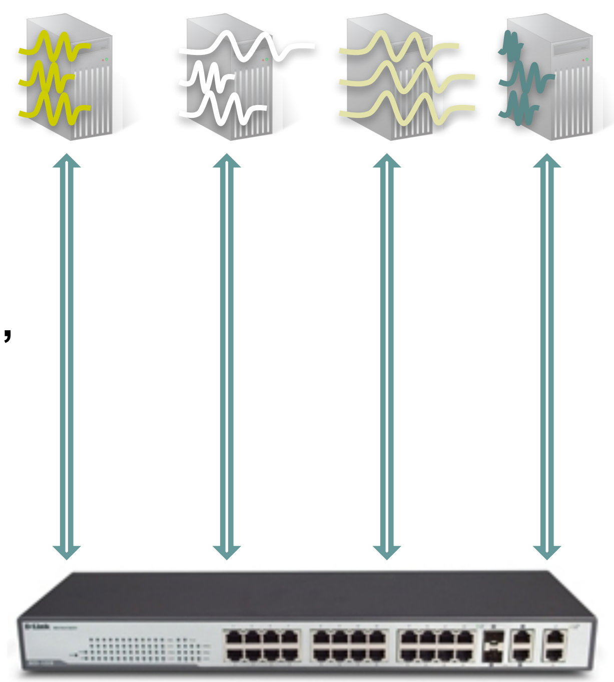
- What can be abstracted away?
  - Abstract away inter-worker communication/synchronization:
    - Automatic consistency models; bandwidth management through distributed shared memory
  - Abstract scheduling away from update equations:
    - Easy to change scheduling strategy, or use dynamic schedules
  - Abstract away worker management:
    - Let ML system decide optimal number and configuration of workers
  - Ideally, reduce programmer burden to just 3 things:
    - Declare model, write updates, write schedule

Systems, Architectures for Distributed ML

There Is No Ideal Distributed System!

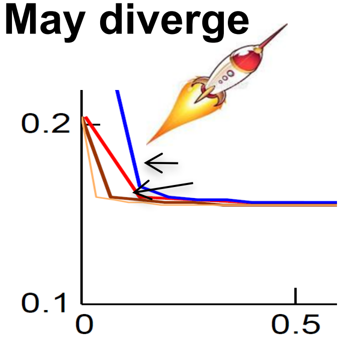
- Not quite that easy…
- **Two distributed challenges:**
  - Networks are slow
  - “Identical” machines rarely perform equally

Unequal performance

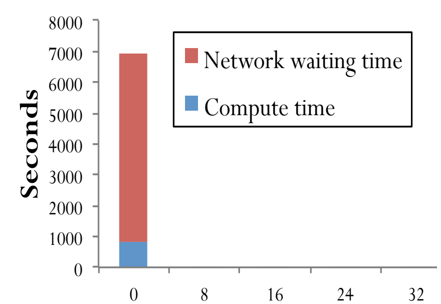


Low bandwidth, High delay

Async execution: May diverge



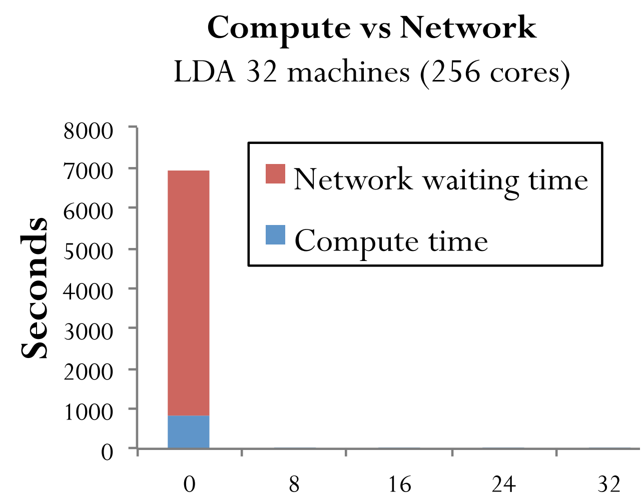
BSP execution: Long sync time



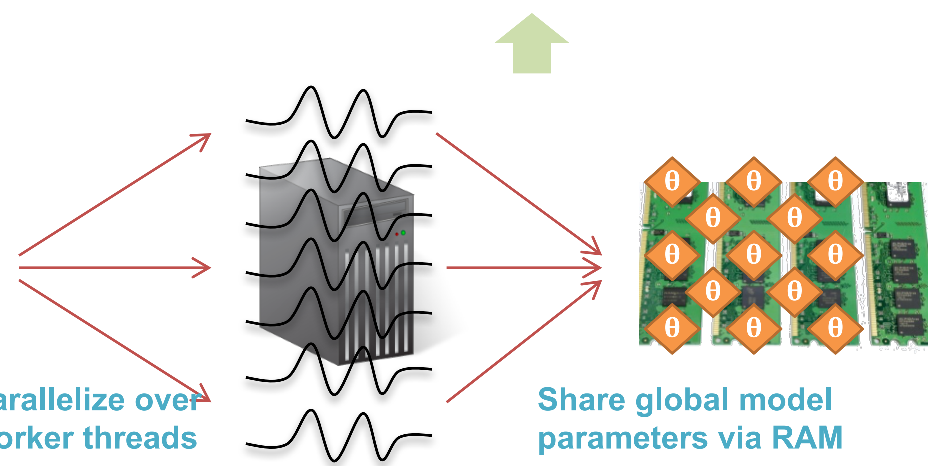
Why need new Big ML systems?

MLer's view

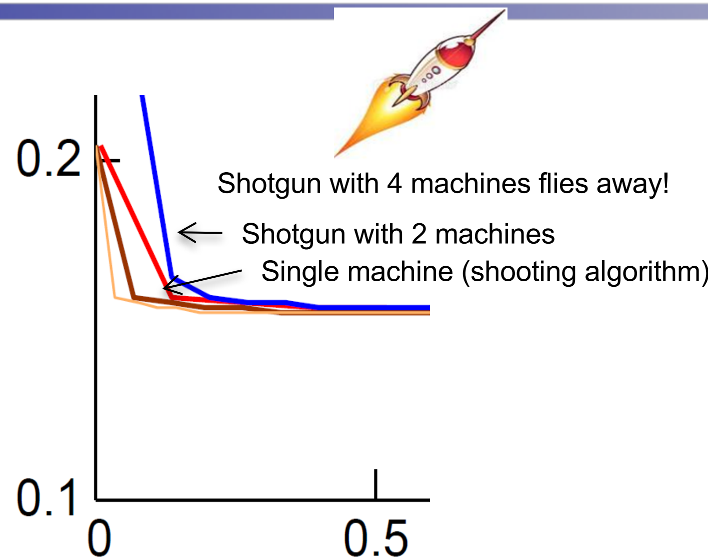
- Focus on
  - Correctness
  - fewer iteration to converge,
- but assuming an ideal system, e.g.,
  - zero-cost sync,
  - uniform local progress




```
for (t = 1 to T) {
  doThings()
  parallelUpdate(x,θ)
  doOtherThings()
}
```



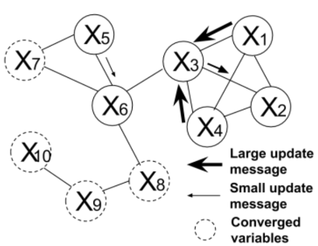
Why need new Big ML systems?



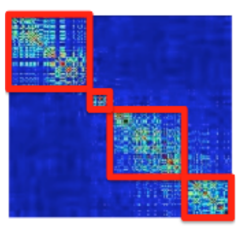
Systems View:

- Focus on
  - high iteration throughput (more iter per sec)
  - strong fault-tolerant atomic operations,
- but assume ML algo is a black box
  - ML algos "still work" under different execution models
  - "easy to rewrite" in chosen abstraction

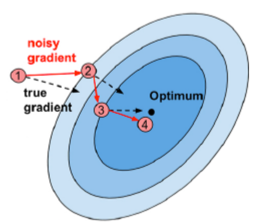
Agonistic of ML properties and objectives **in system design**



Non-uniform convergence

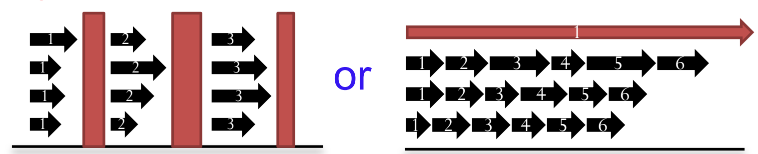


Dynamic structures

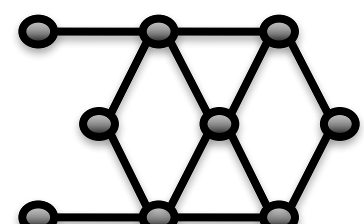


Error tolerance

Synchronization model



Programming model



Why need new Big ML systems?

MLer's view

- Focus on
  - Correctness
  - fewer iteration to converge,
- but assuming an ideal system, e.g.,
  - zero-cost sync,
  - uniform local progress

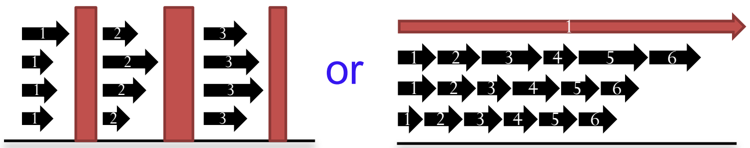
```
for (t = 1 to T) {
  doThings()
  parallelUpdate(x,θ)
  doOtherThings()
}
```

Oversimplify systems issues

- **need machines to perform consistently**
- **need lots of synchronization**
- **or even try not to communicate at all**

Systems View:

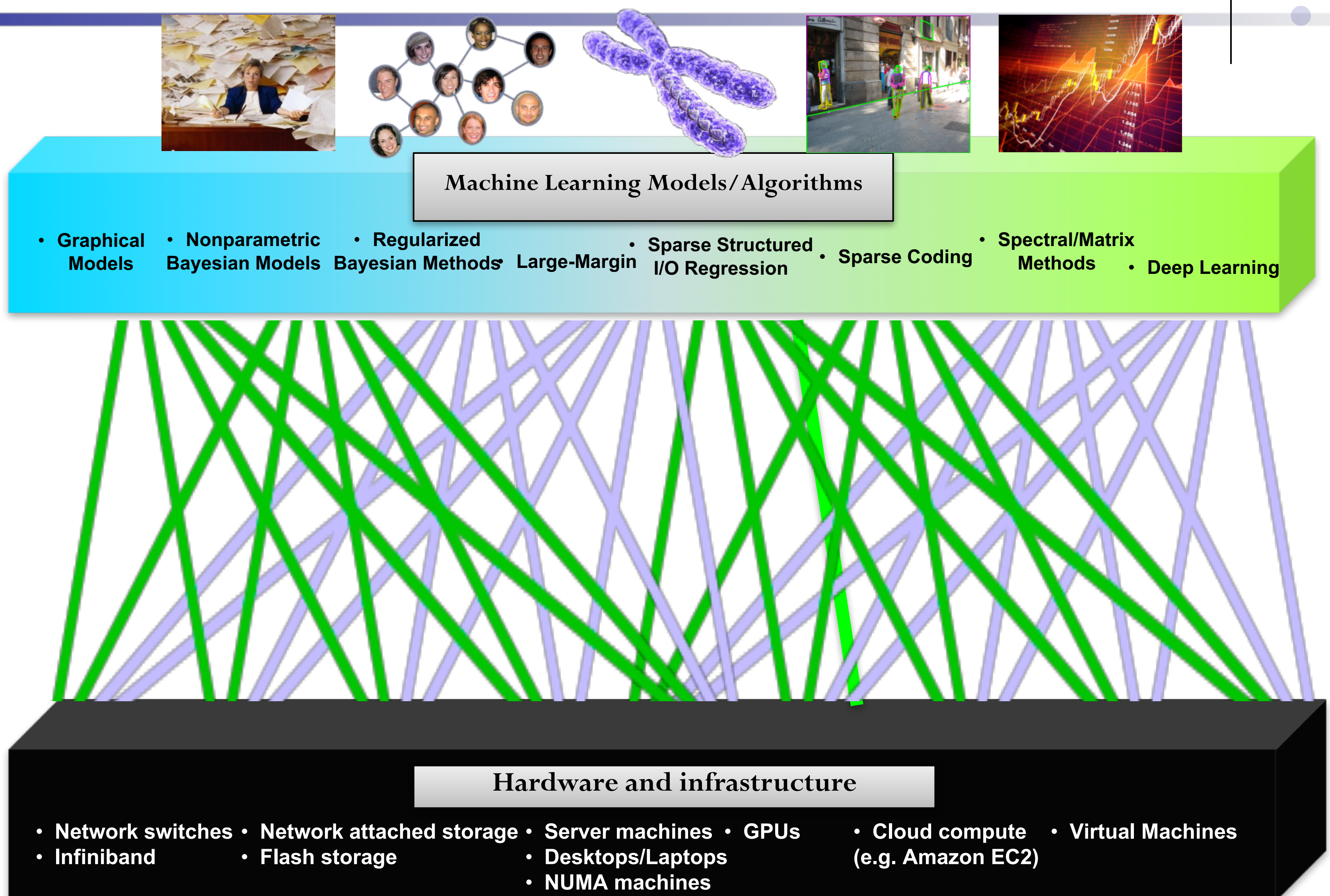
- Focus on
  - high iteration throughput (more iter per sec)
  - strong fault-tolerant atomic operations,
- but assume ML algo is a black box
  - ML algos "still work" under different execution models
  - "easy to rewrite" in chosen abstraction



Oversimplify ML issues and/or ignore ML opportunities

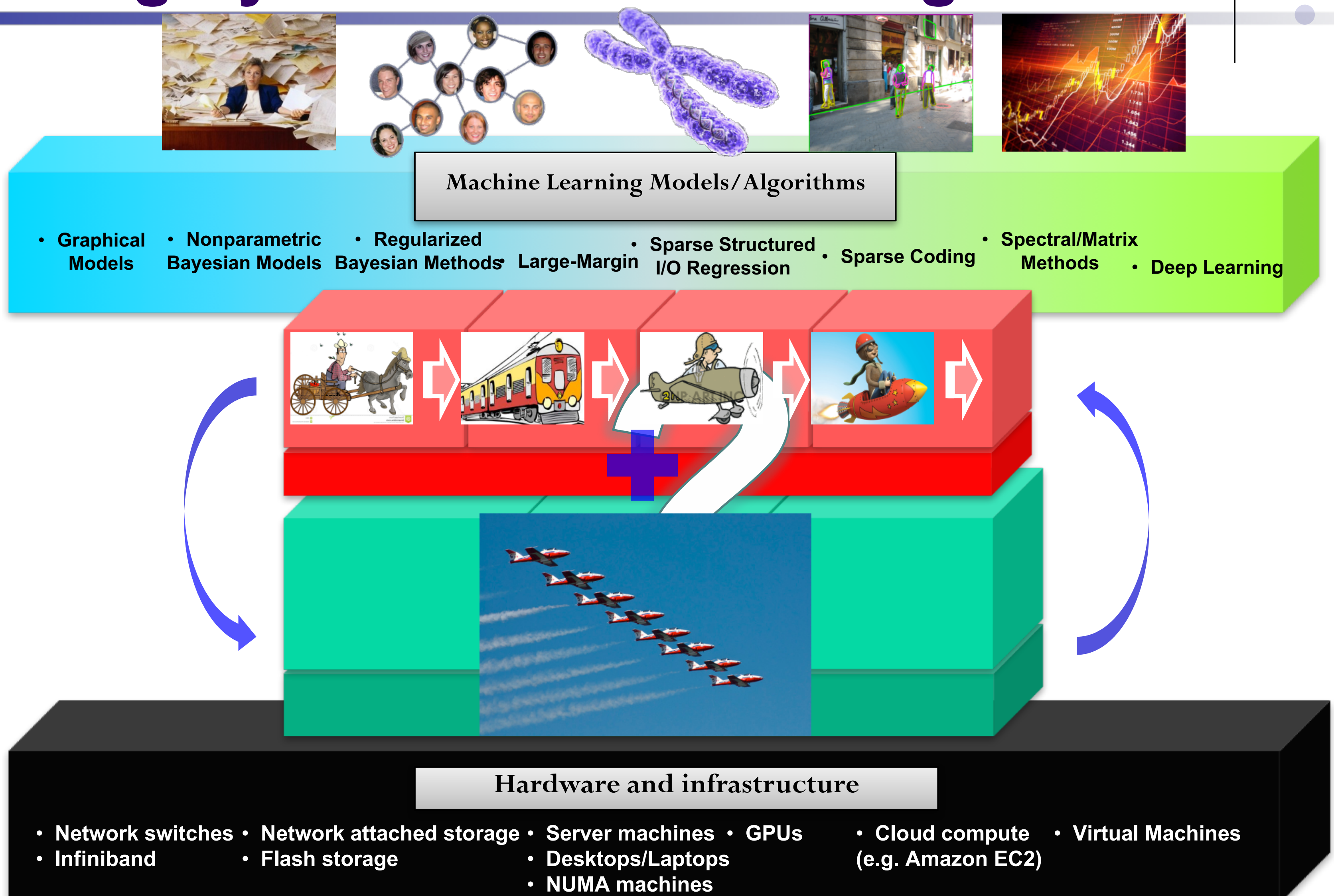
- **ML algos "just work" without proof**
- **Conversion of ML algos across different program models (graph programs, RDD) is easy**

Solution:

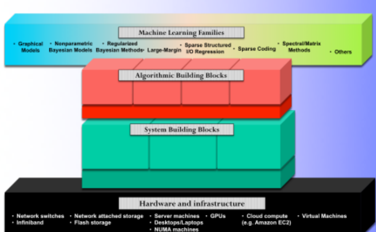


Machine Learning Models/Algorithms
• Graphical Models
• Nonparametric Bayesian Models
• Regularized Bayesian Methods
• Large-Margin
• Sparse Structured I/O Regression
• Sparse Coding
• Spectral/Matrix Methods
• Deep Learning
Hardware and infrastructure
• Network switches
• Infiniband
• Network attached storage
• Flash storage
• Server machines
• Desktops/Laptops
• NUMA machines
• GPUs
• Cloud compute (e.g. Amazon EC2)
• Virtual Machines

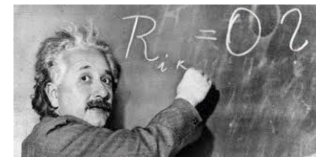
Solution:
An Alg/Sys INTERFACE for Big ML



The Big-ML "Stack" - More than just software



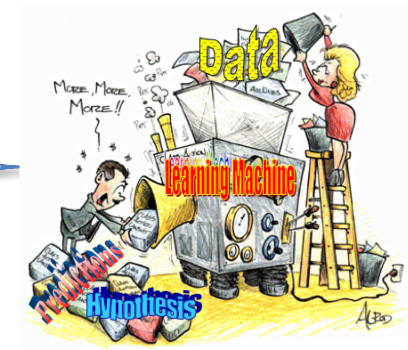
Theory: Degree of parallelism, convergence analysis, sub-sample complexity …



Representation: Compact and informative features

Model: Generic building blocks: loss functions, structures, constraints, priors …

Algorithm: Parallelizable and stochastic MCMC, VI, Opt, Spectrum …



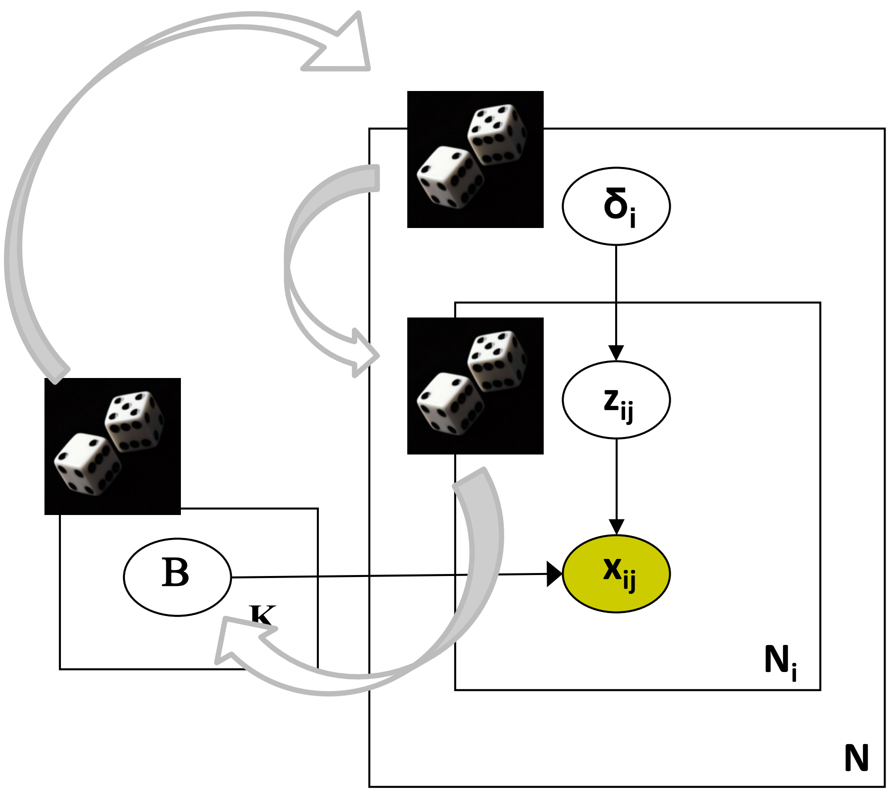
Programming model & Interface: High: Matlab/R
Medium: C/JAVA
Low: MPI

System: Distributed architecture: DFS, KV-store, task scheduler…

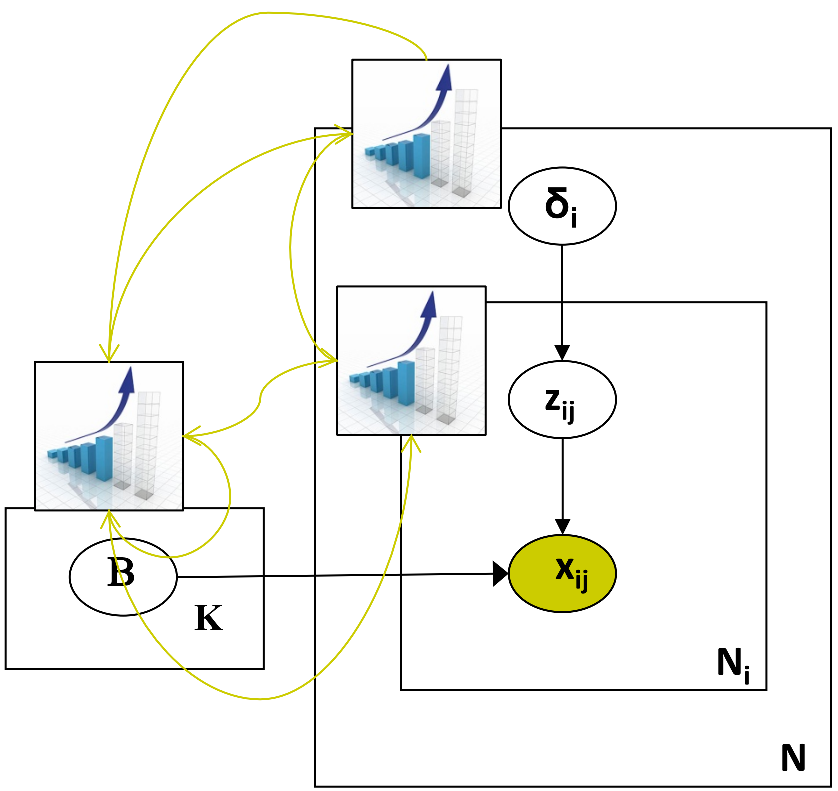
Hardware: GPU, flash storage, cloud …

ML algorithms are Iterative-Convergent

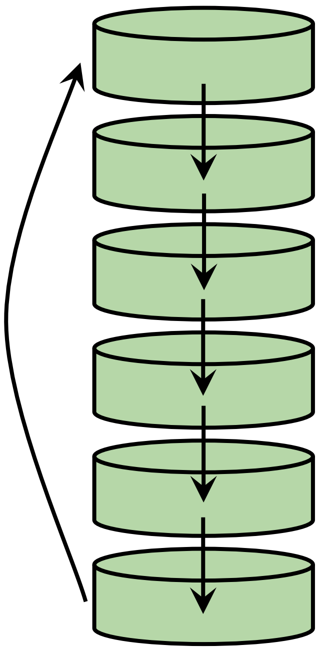
Markov Chain Monte Carlo

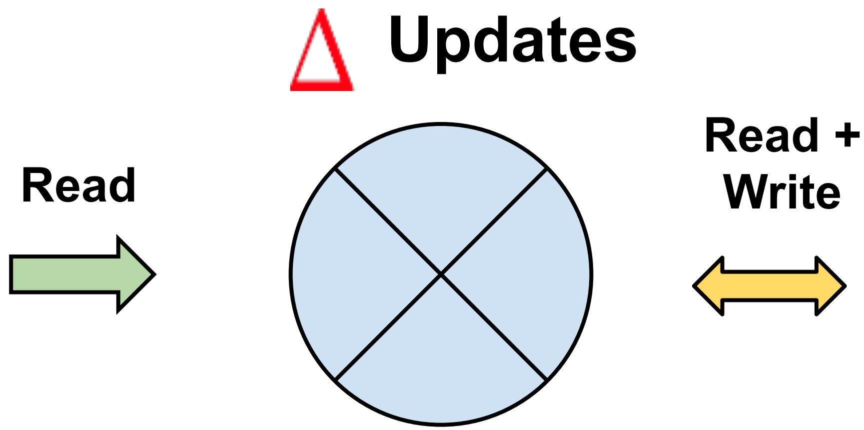


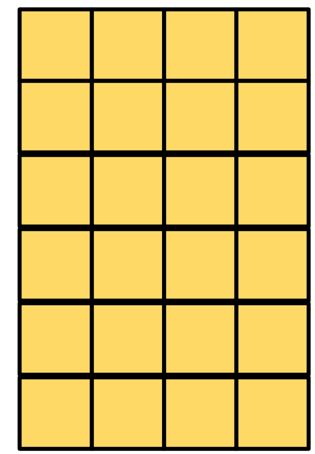
Optimization



A General Picture of ML Iterative-Convergent Algorithms







D

Data

Iterative Algorithm

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

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

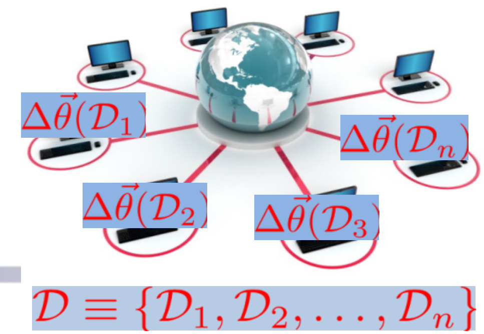
$F()$ Aggregate + Transform

Δ Intermediate Updates

$A^{(t-1)}$

Model Parameters at iteration (t-1)

Issues with Hadoop and I-C ML Algorithms?



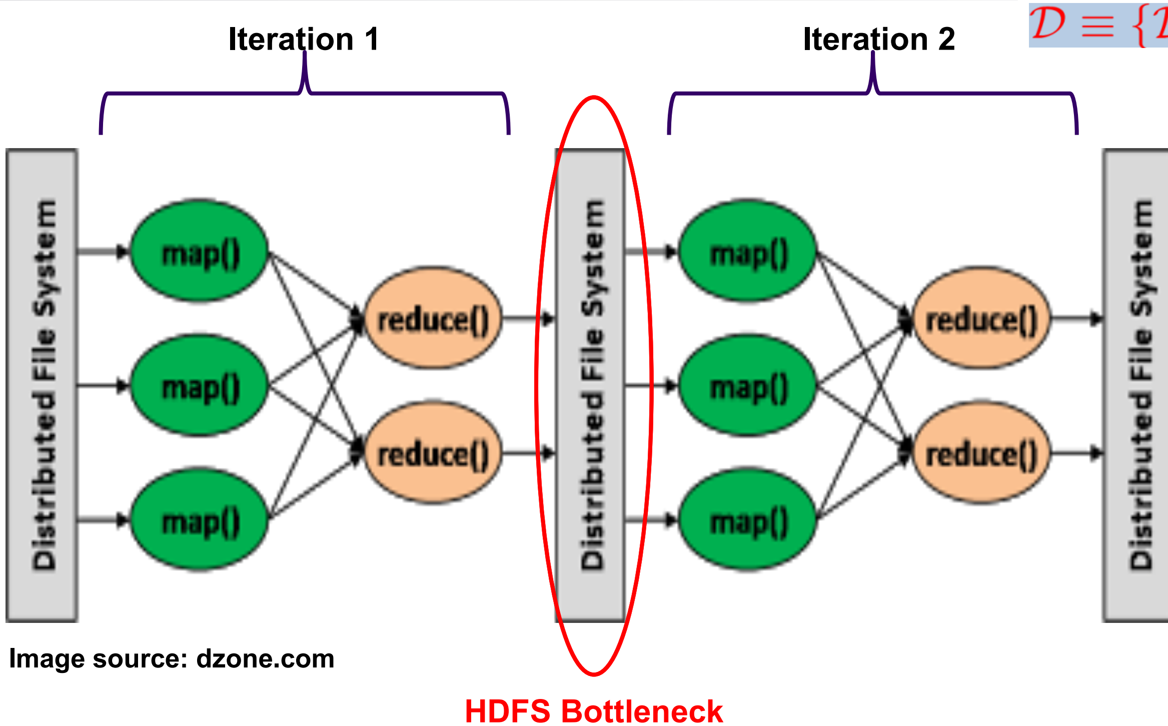
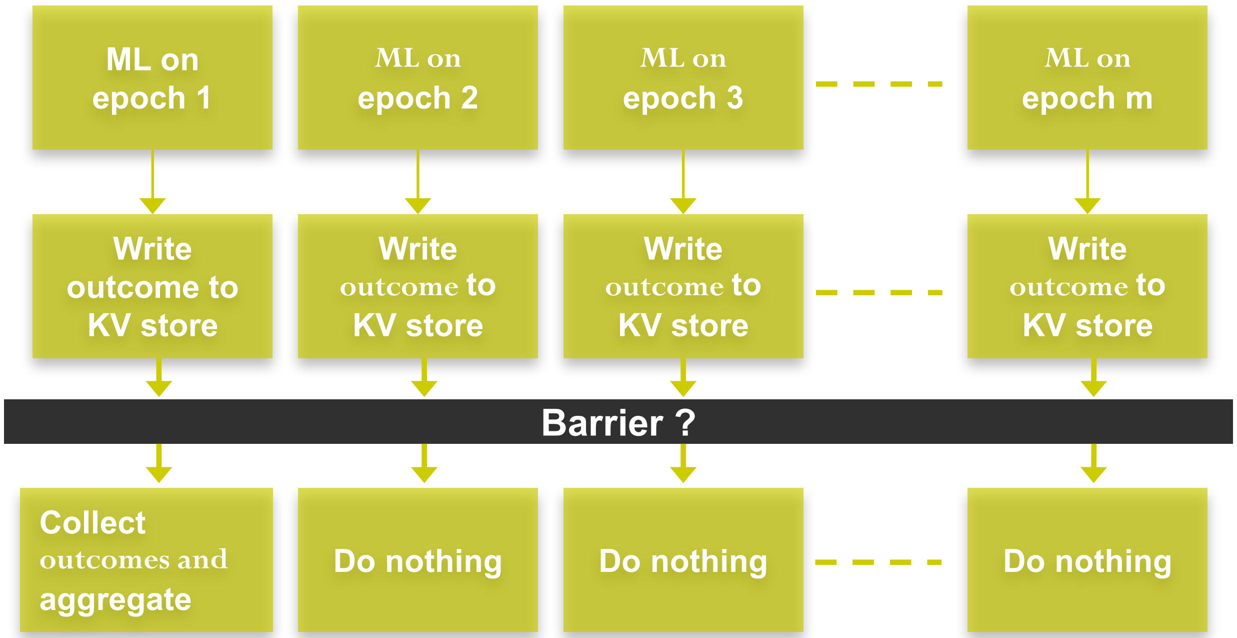


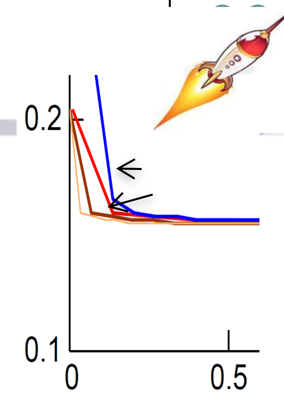
Image source: dzone.com

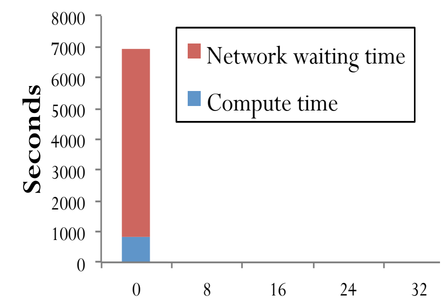
Naïve MapReduce not best for ML

- Hadoop can execute iterative-convergent, data-parallel ML...
  - map() to distribute data samples i, compute update $\Delta(D_i)$
  - reduce() to combine updates $\Delta(D_i)$
  - Iterative ML algo = repeat map()+reduce() again and again
- But reduce() writes to HDFS before starting next iteration's map() - very slow iterations!

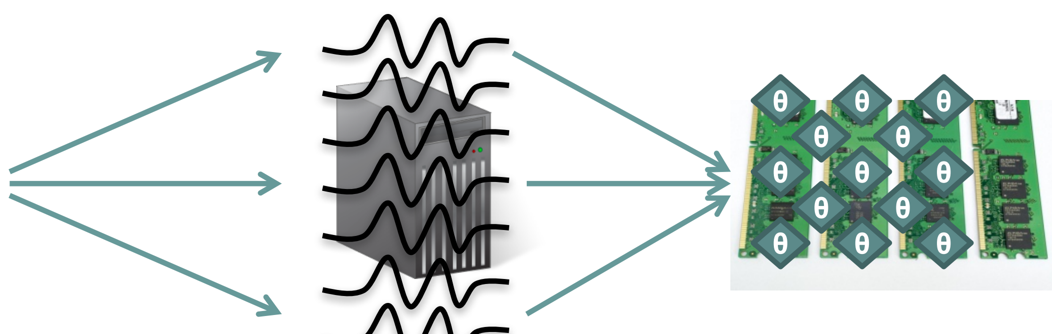
Good Parallelization Strategy is important



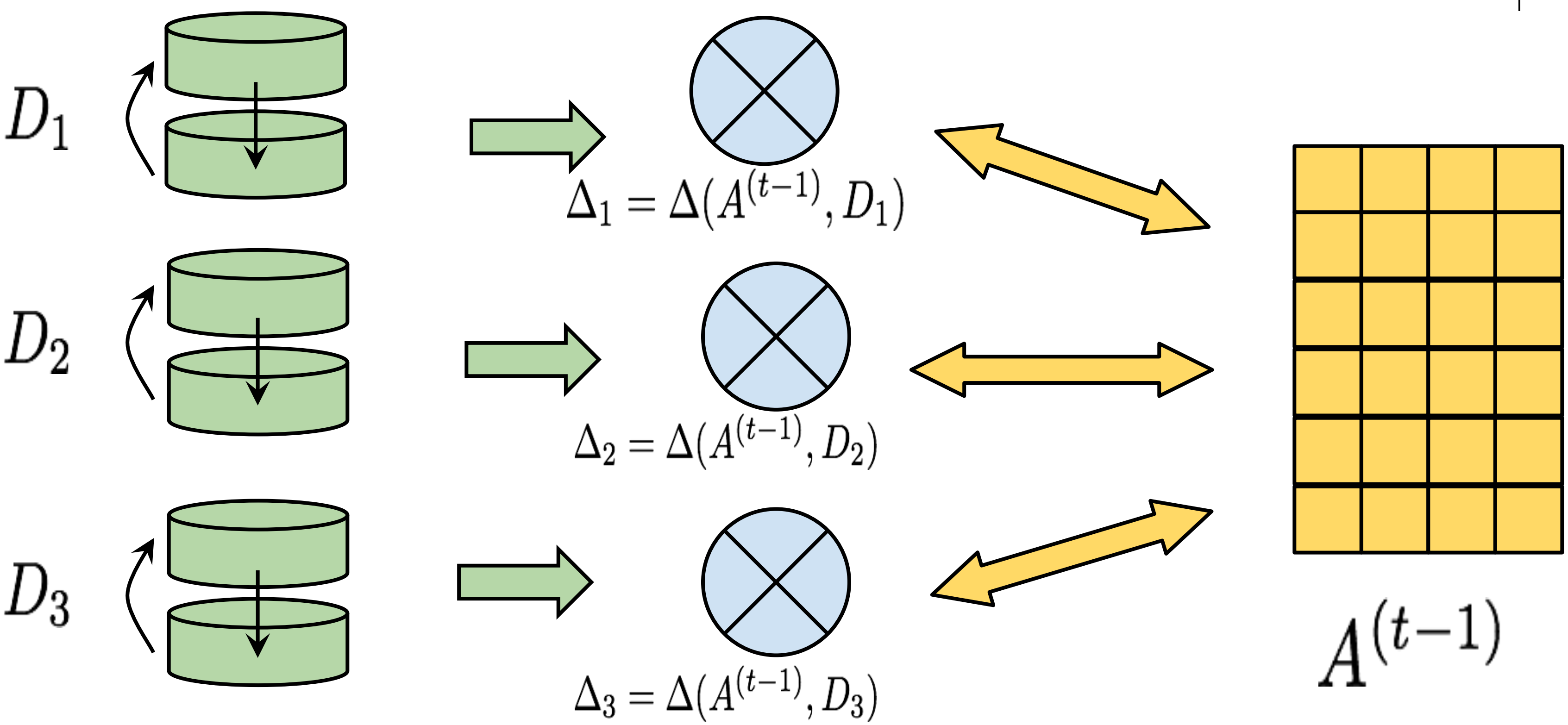





```
for (t = 1 to T) {
  doThings()
  parallelUpdate(x,θ)
  doOtherThings()
}
```



Data Parallelism



Additive Updates

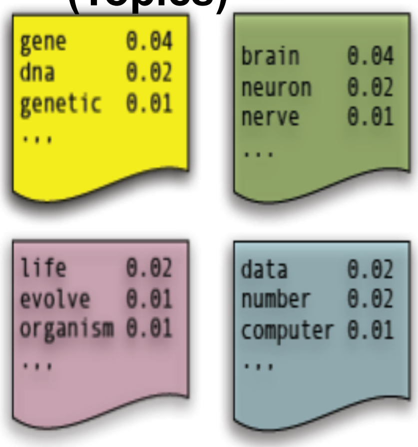
$$\Delta = \sum_{p=1}^{3} \Delta_p$$

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

Example Data Parallel: Topic Models

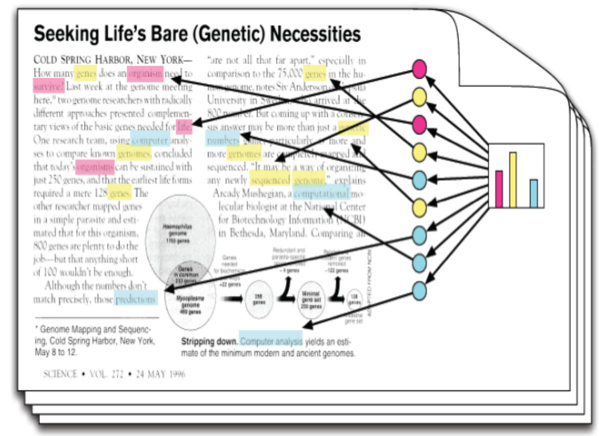
BIG DATA (billions of docs)

Model (Topics)



Update (MCMC algo)

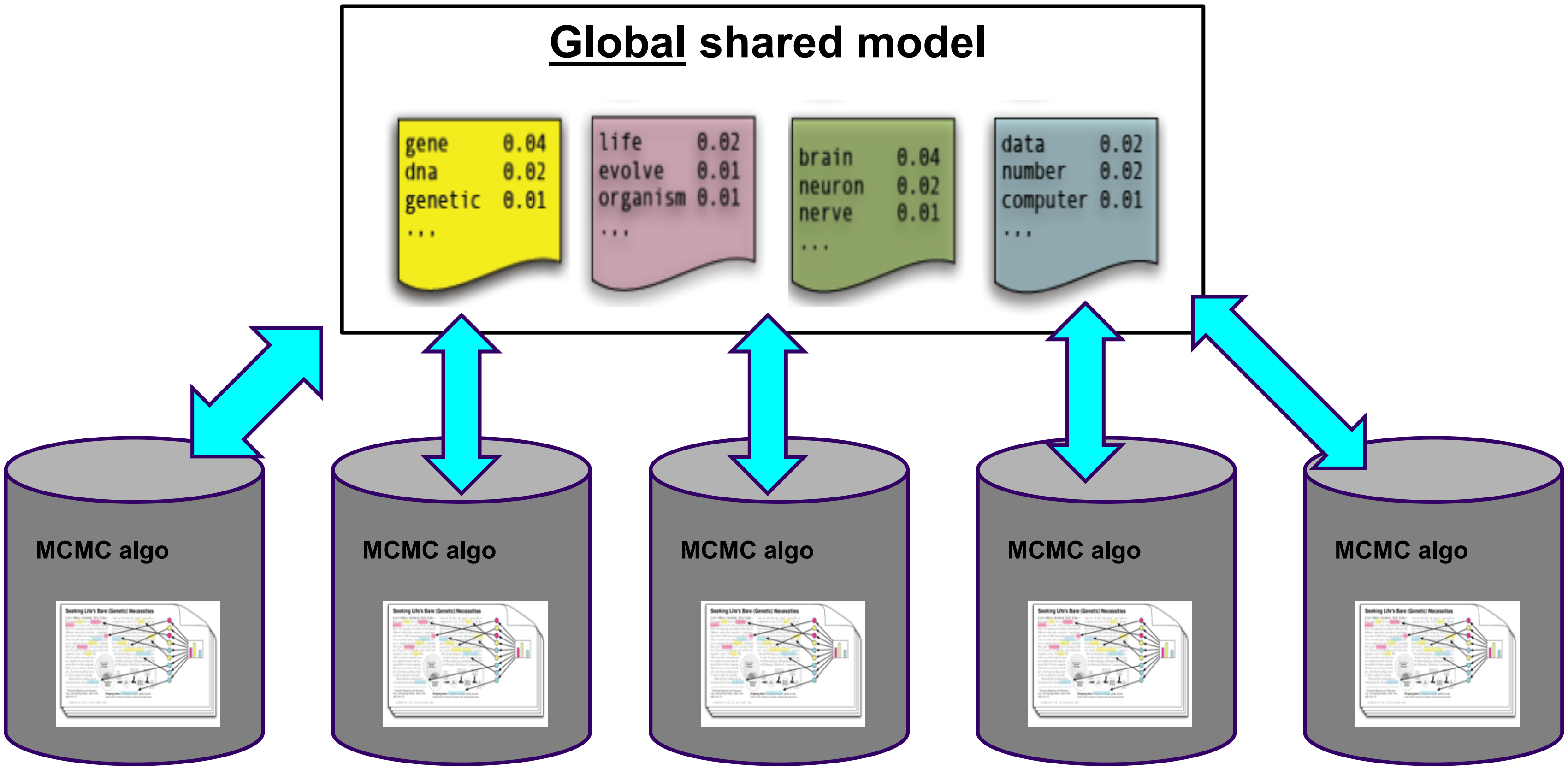
Data (Docs)



$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

Example Data Parallel: Topic Models

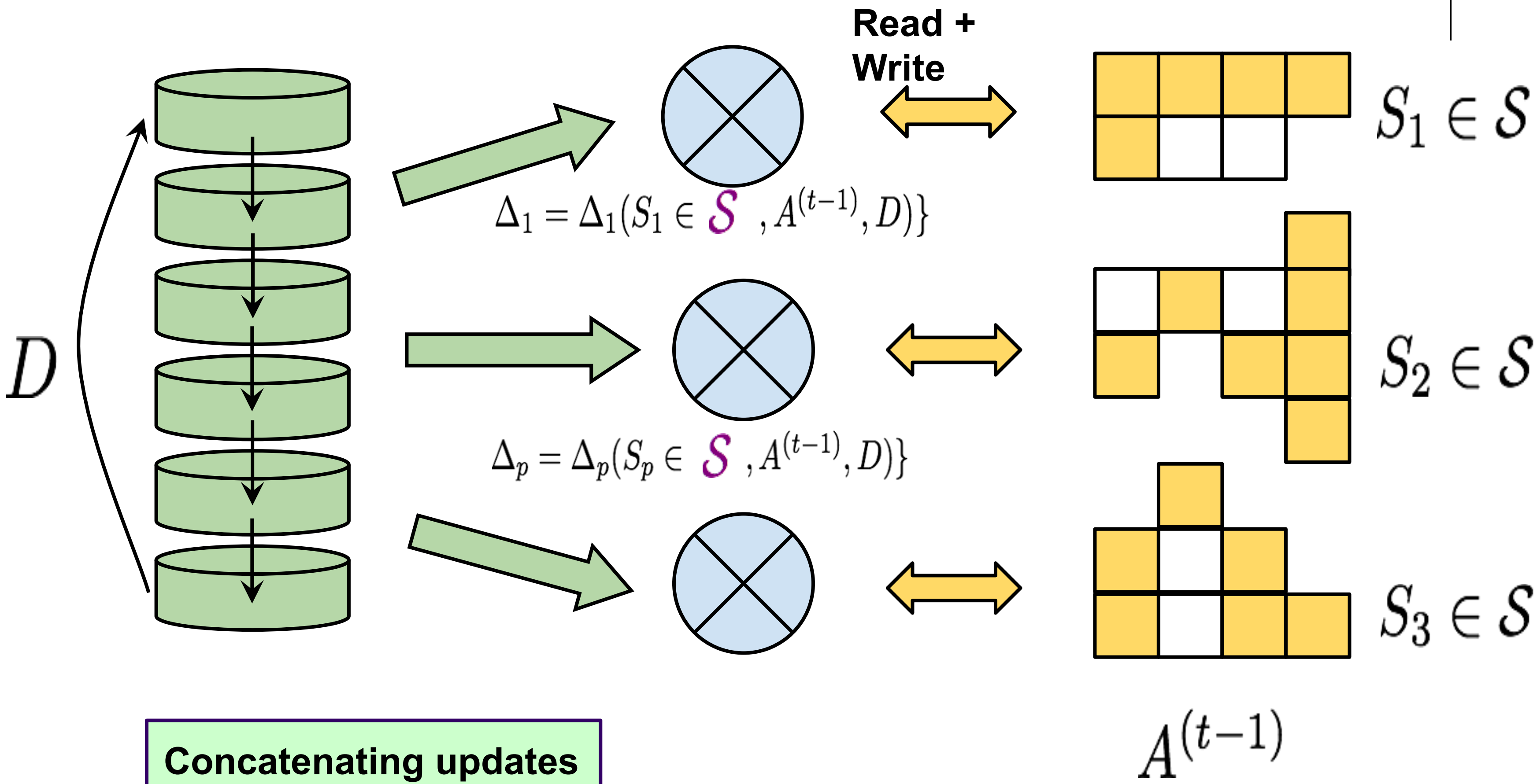
$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$



Model Parallelism

Scheduling Function

$\mathcal{S} = \mathcal{S}(A^{(t-1)}, D)$



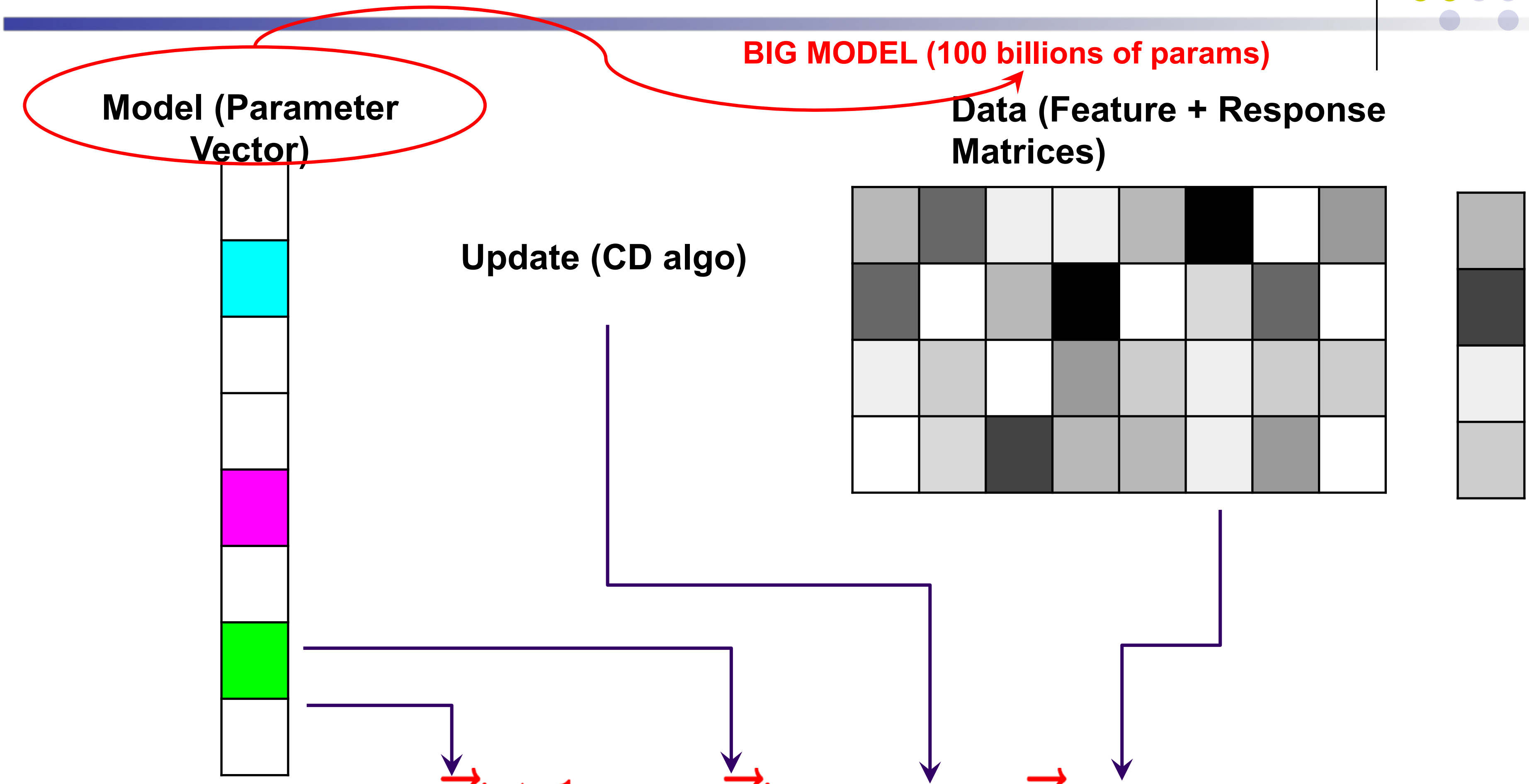
Concatenating updates

$$\Delta = \{\Delta_p\}$$

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

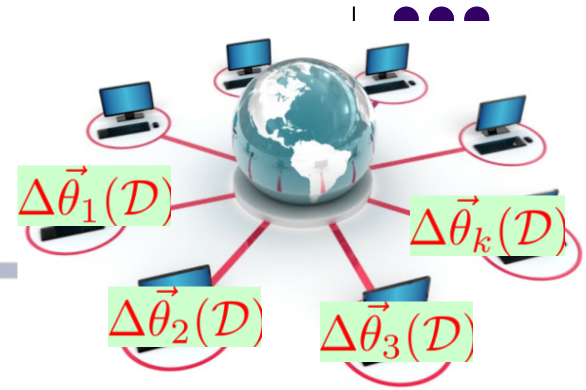
model parameters not updated in this iteration

Example Model Parallel: Lasso Regression



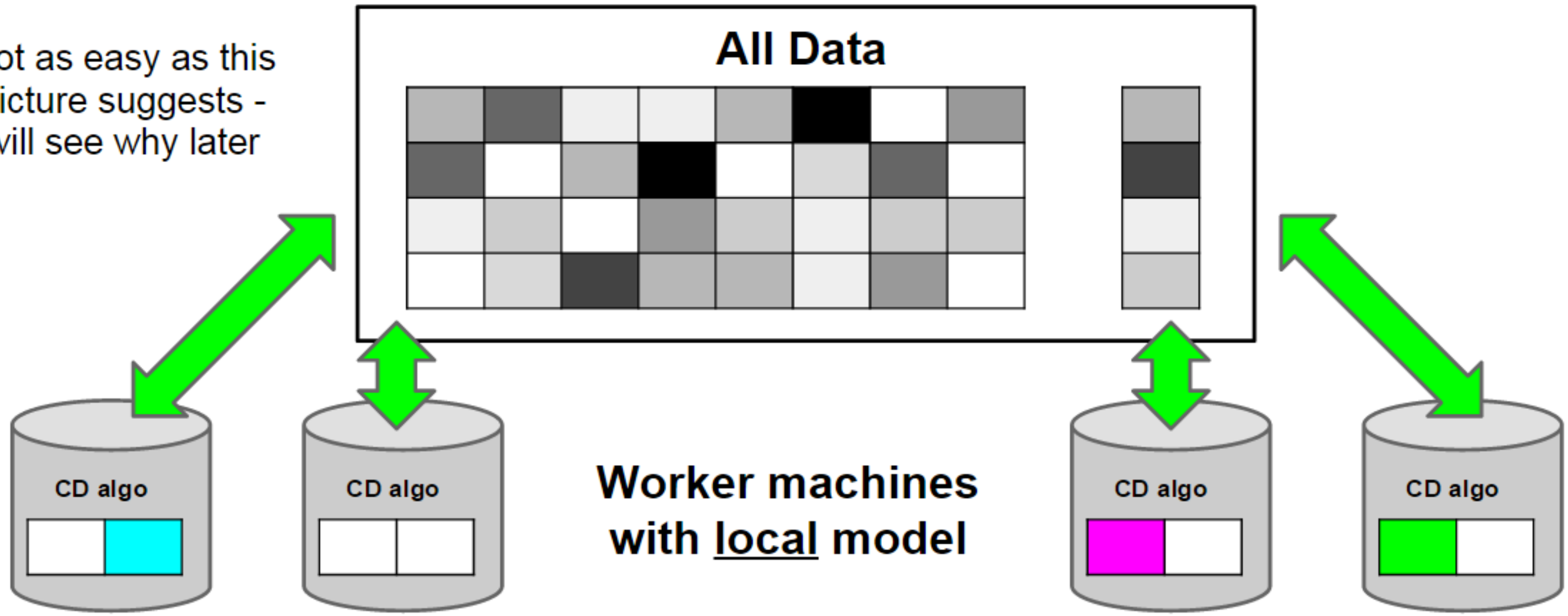
$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

Example Model Parallel: Lasso Regression

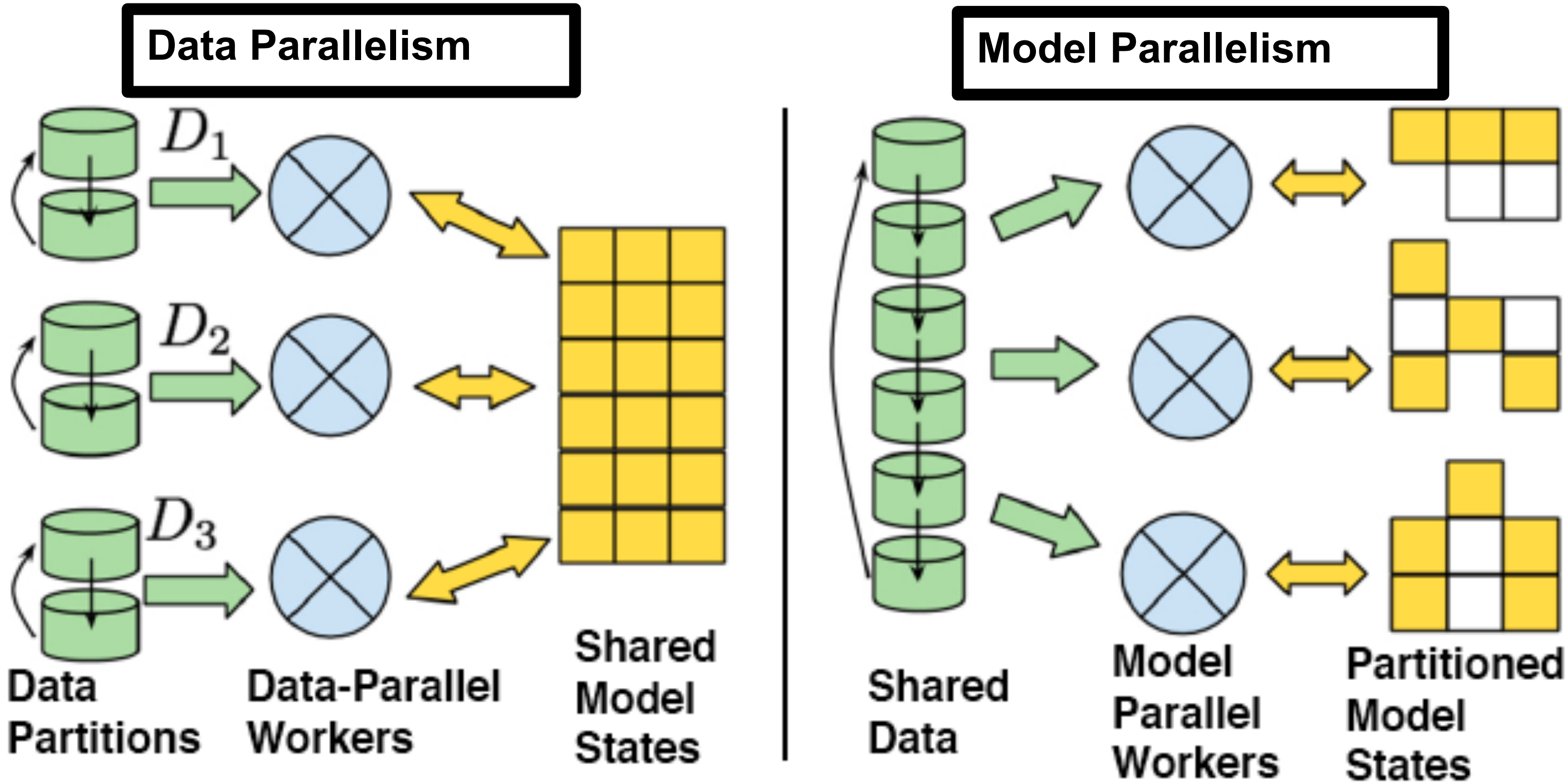


$$\vec{\theta} \equiv [\vec{\theta}_1^{\mathrm{T}}, \vec{\theta}_2^{\mathrm{T}}, \ldots, \vec{\theta}_k^{\mathrm{T}}\}^{\mathrm{T}}$$

Not as easy as this picture suggests - will see why later



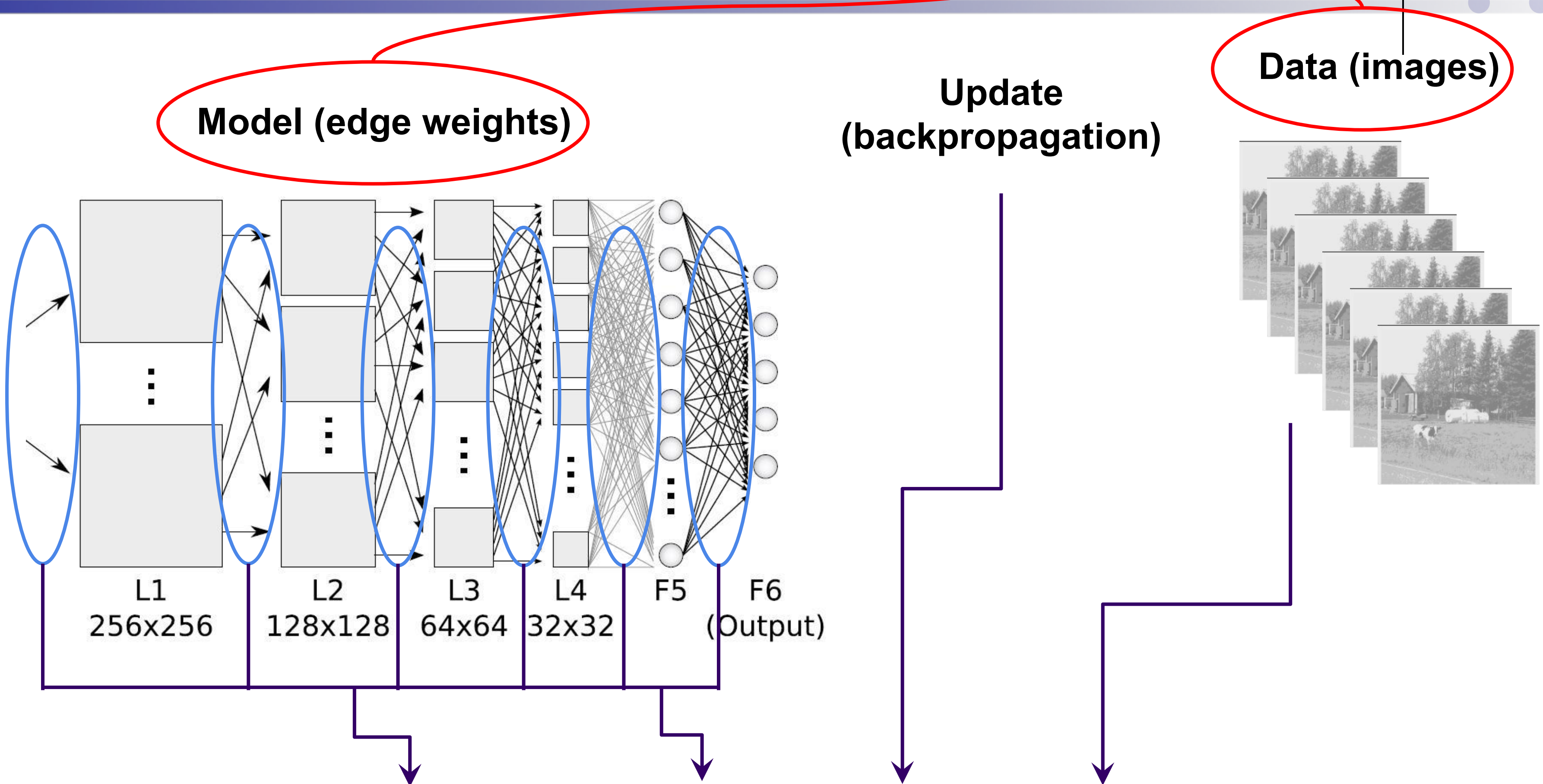
A Dichotomy of Data and Model in ML Programs



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

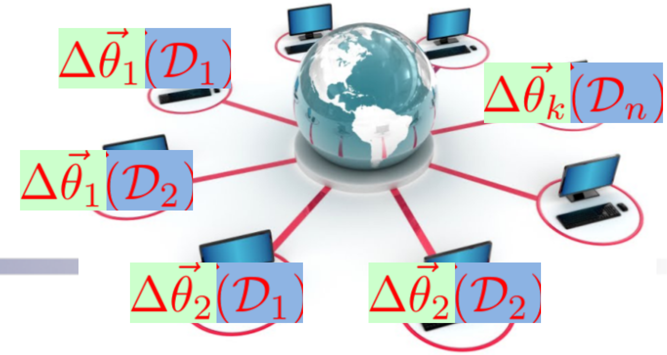
Data+Model Parallel: Solving Big Data+Model

Data & Model both big!
Millions of images,
Billions of weights
What to do?



$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

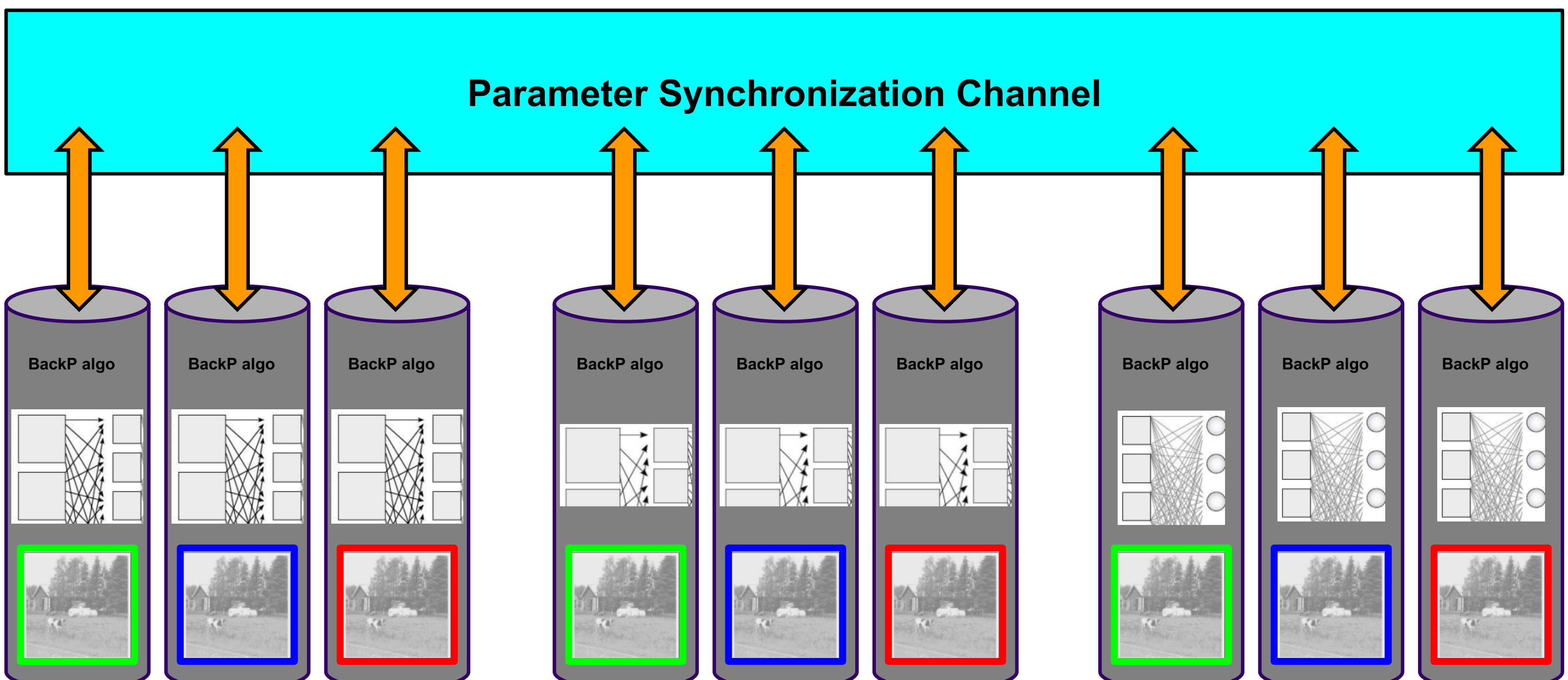
Data+Model Parallel: Solving Big Data+Model



Tackle Deep Learning scalability challenges by combining data+model parallelism

$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$

$$\vec{\theta} \equiv [\vec{\theta}_1^{\mathrm{T}}, \vec{\theta}_2^{\mathrm{T}}, \ldots, \vec{\theta}_k^{\mathrm{T}}\}^{\mathrm{T}}$$

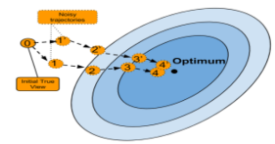


How difficult is data/model-parallelism?

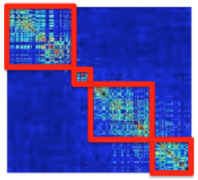
- Certain mathematical conditions must be met
- Data-parallelism generally OK when data IID (independent, identically distributed)
  - Very close to serial execution, in most cases
- Naive Model-parallelism doesn't work
  - NOT equivalent to serial execution of ML algo
  - Need carefully designed schedule

Intrinsic Properties of ML Programs

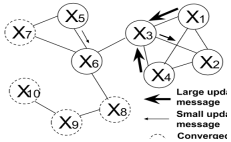
- ML is **optimization-centric**, and admits an **iterative convergent** algorithmic solution rather than a one-step closed form solution
  - **Error tolerance**: often robust against limited errors in intermediate calculations



  - **Dynamic structural dependency**: changing correlations between model parameters critical to efficient parallelization



  - **Non-uniform convergence**: parameters can converge in very different number of steps

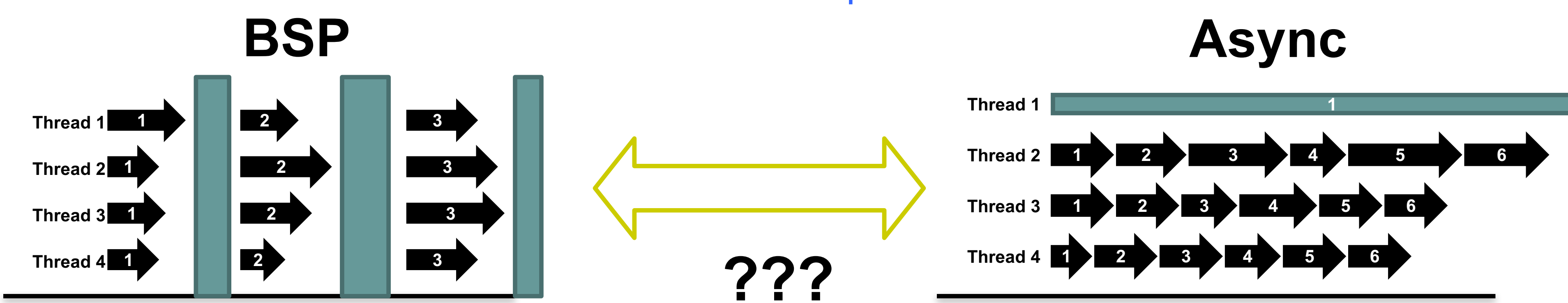


- Whereas traditional programs are **transaction-centric**, thus only guaranteed by **atomic correctness** at every step

- Most existing platforms (e.g., Spark, GraphLab) have not yet systematically explore and exploit above properties

Challenges in Data Parallelism

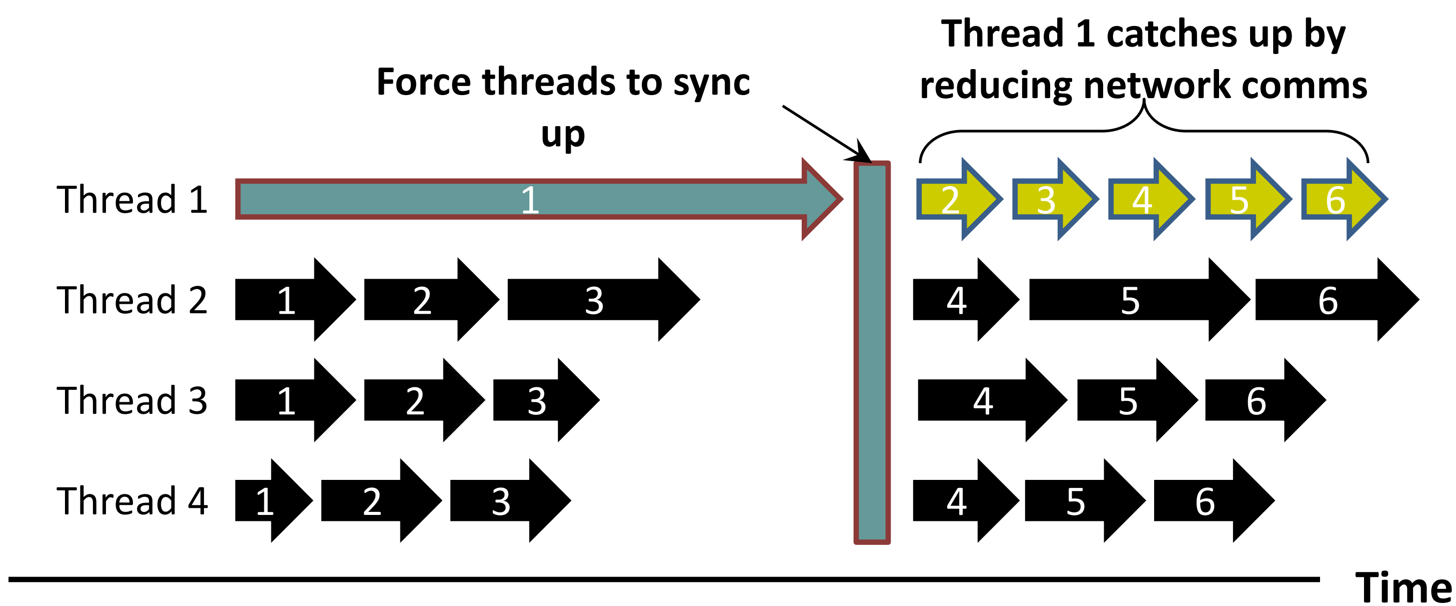
- **Existing ways are either safe/slow (BSP), or fast/risky (Async)**
- **Challenge 1: Need "Partial" synchronicity**
  - Spread network comms evenly (don't sync unless needed)
  - Threads usually shouldn't wait – but mustn't drift too far apart!
- **Challenge 2: Need straggler tolerance**
  - Slow threads must somehow catch up



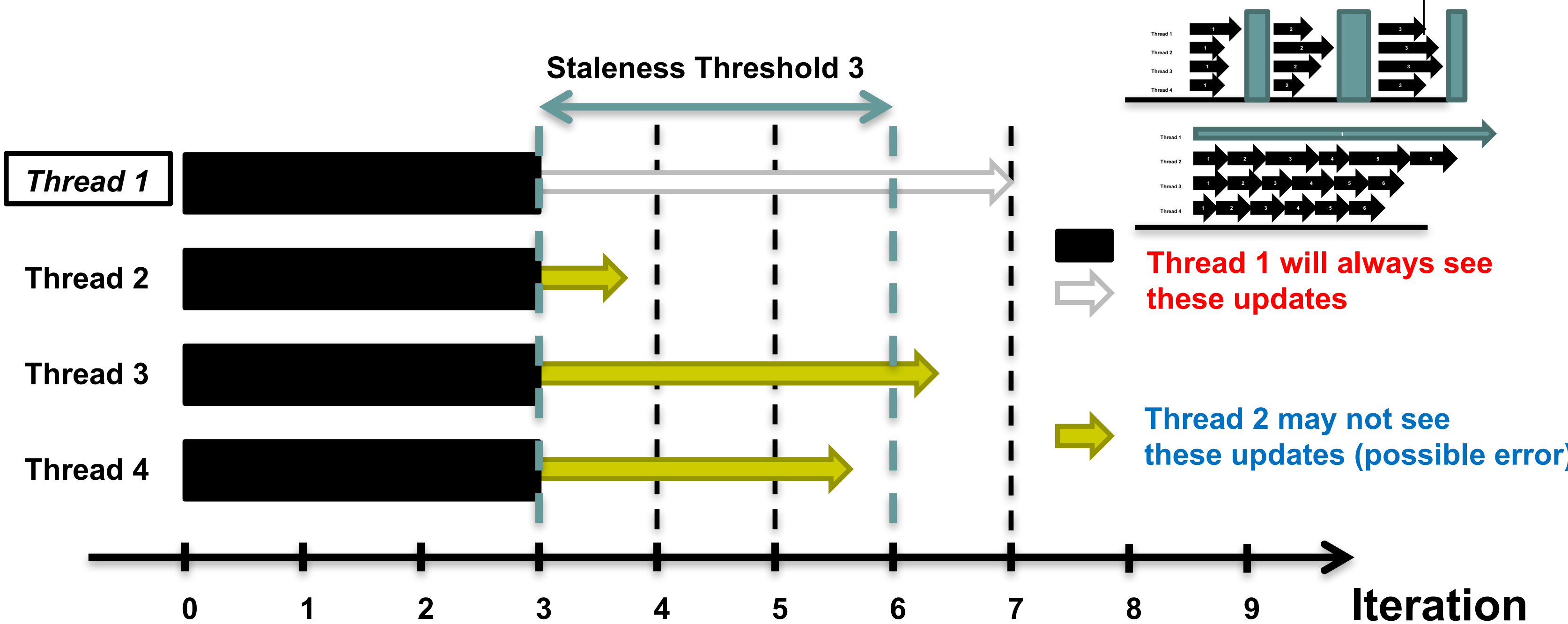
Is persistent memory really necessary for ML?

Is there a middle ground for data-parallel consistency?

- **Challenge 1: "Partial" synchronicity**
  - Spread network comms evenly (don't sync unless needed)
  - Threads usually shouldn't wait – but mustn't drift too far apart!
- **Challenge 2: Straggler tolerance**
  - Slow threads must somehow catch up



High-Performance Consistency Models for Fast Data-Parallelism [Ho et al., 2013]



Stale Synchronous Parallel (SSP), a “bounded-asycnhronous” model

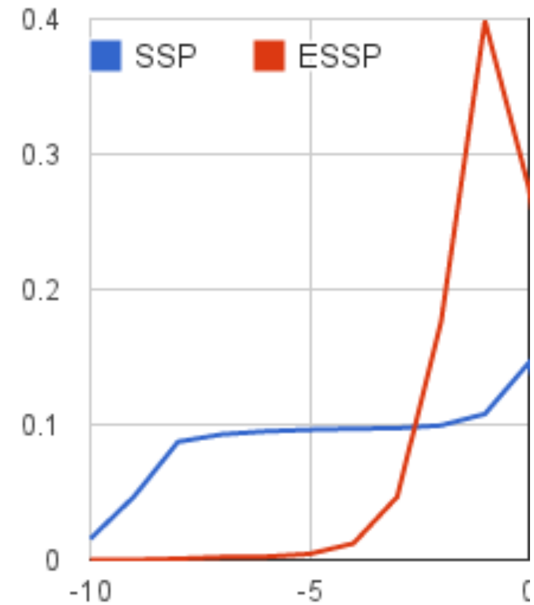
- Allow threads to run at their own pace, without synchronization
- Fastest/slowest threads not allowed to drift >S iterations apart
- Threads cache local (stale) versions of the parameters, to reduce network syncing

Consequence:

- Asynchronous-like speed, BSP-like ML correctness guarantees
- Guaranteed age bound (staleness) on reads
- Contrast: no-age-guarantee Eventual Consistency seen in Cassandra, Memcached

Improving Bounded-Async via Eager Updates [Dai et al., 2015]

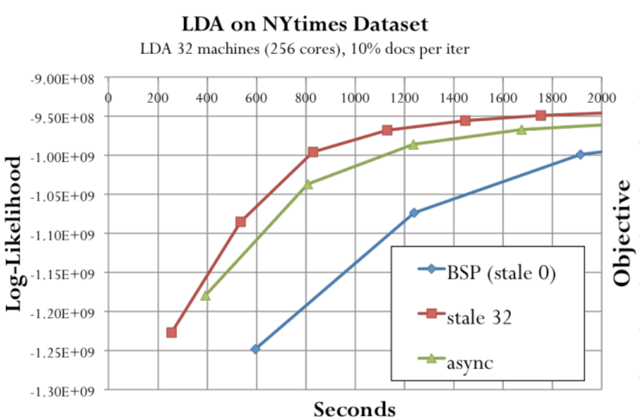
- Eager SSP (ESSP) protocol
  - Use spare bandwidth to push fresh parameters sooner
- Figure: difference in stale reads between SSP and ESSP
  - ESSP has fewer stale reads; lower staleness variance
  - Faster, more stable convergence (theorems later)



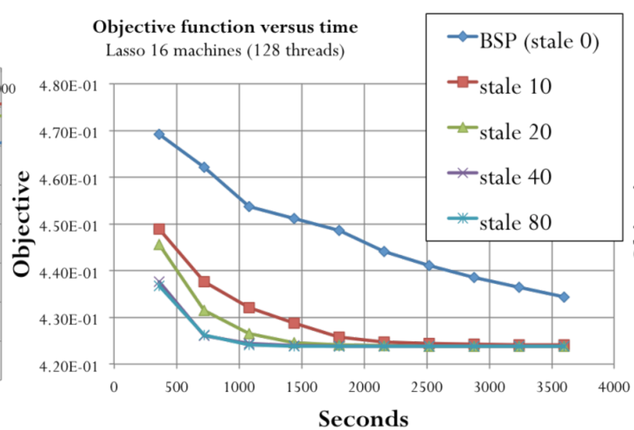
Enjoys Async Speed, yet BSP Guarantee, across algorithms

- Scale up Data Parallelism without being limited by long BSP synchronization time

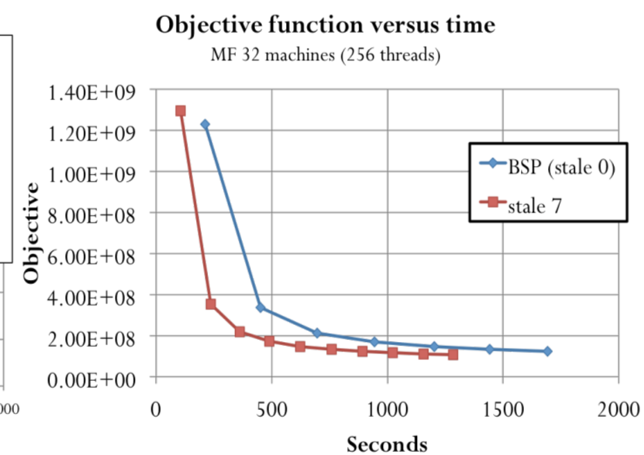
- Effective across different algorithms, e.g. LDA, Lasso, Matrix Factorization:



LDA



LASSO

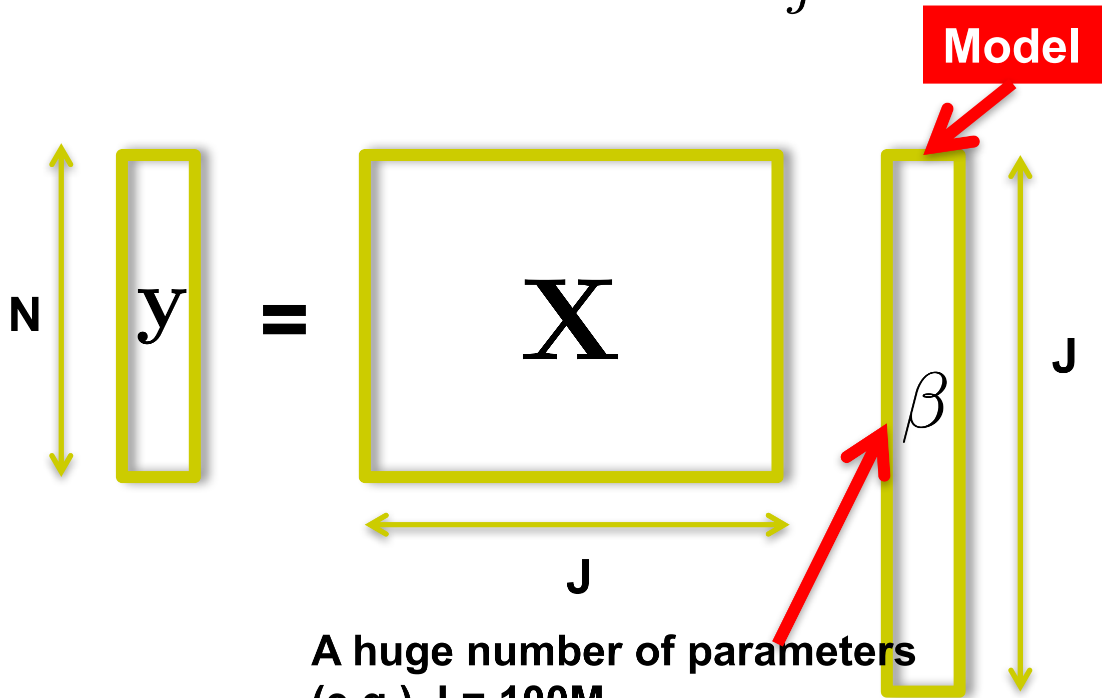


Matrix Fact.

Challenges in Model Parallelism

- Recall Lasso regression:

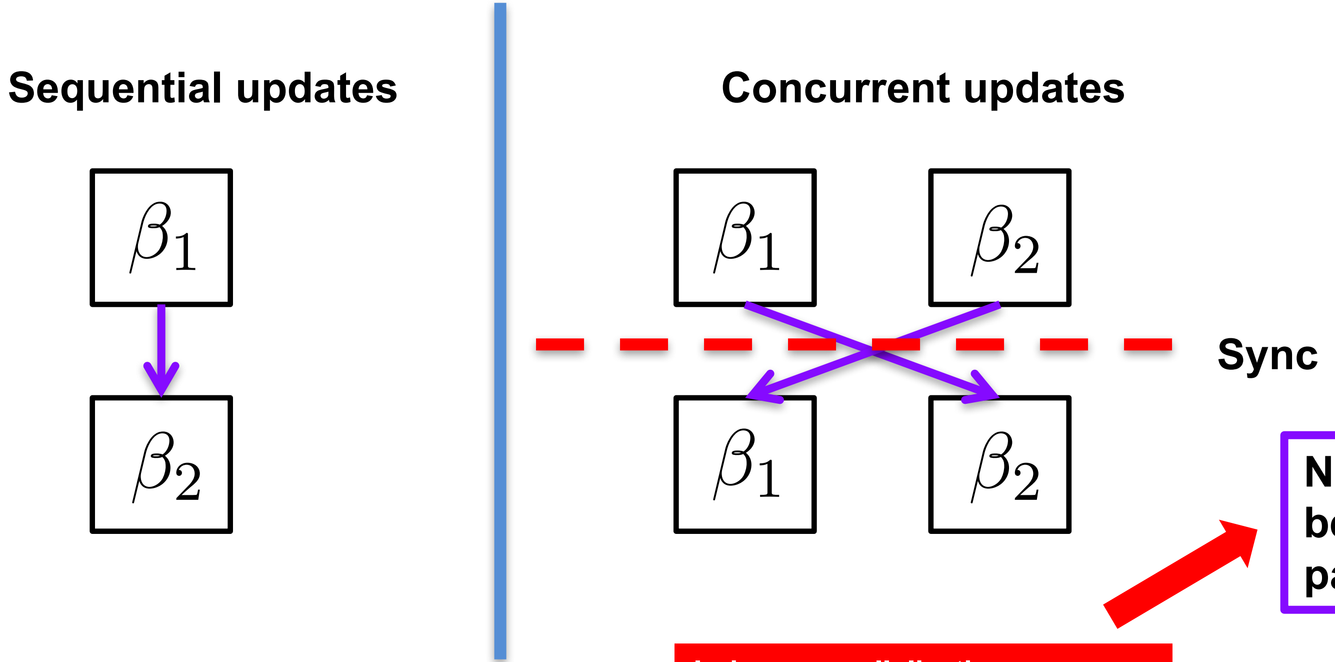
$$\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda \sum_j |\beta_j|$$



A huge number of parameters
(e.g.) J = 100M

Challenge 1: Model Dependencies

- Concurrent updates of β may induce errors

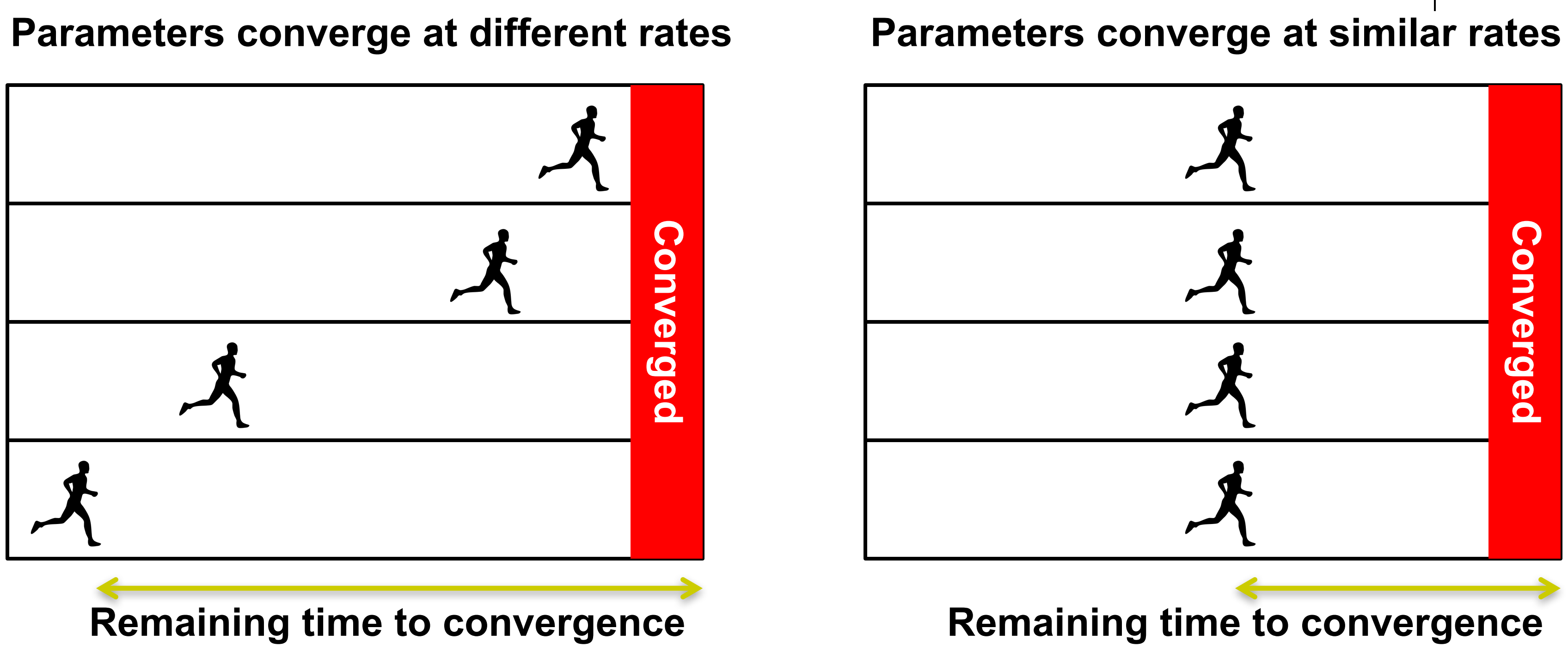


Need to check $x_1^T x_2$ before updating parameters

Induces parallelization error

$$\beta_1^{(t)} \leftarrow S(\mathbf{x}_1^T \mathbf{y} - \boxed{\mathbf{x}_1^T \mathbf{x}_2 \beta_2^{(t-1)}}, \lambda)$$

Challenge 2: Uneven Convergence Rate on Parameters



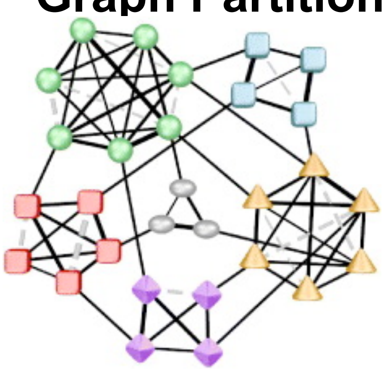
- Convergence time determined by slowest parameters
- How to make slowest parameters converge more quickly?

Is there a middle ground for model-parallel consistency?

- **Existing ways are either safe but slow, or fast but risky**
- **Challenge 1: need approximate but fast model partition**
  - Full representation of data/model, and explicitly compute all dependencies via graph cut is not feasible
- **Challenge 2: need dynamic load balancing**
  - Capture and explore transient model dependencies
  - Explore uneven parameter convergence

Graph Partition

GraphLab

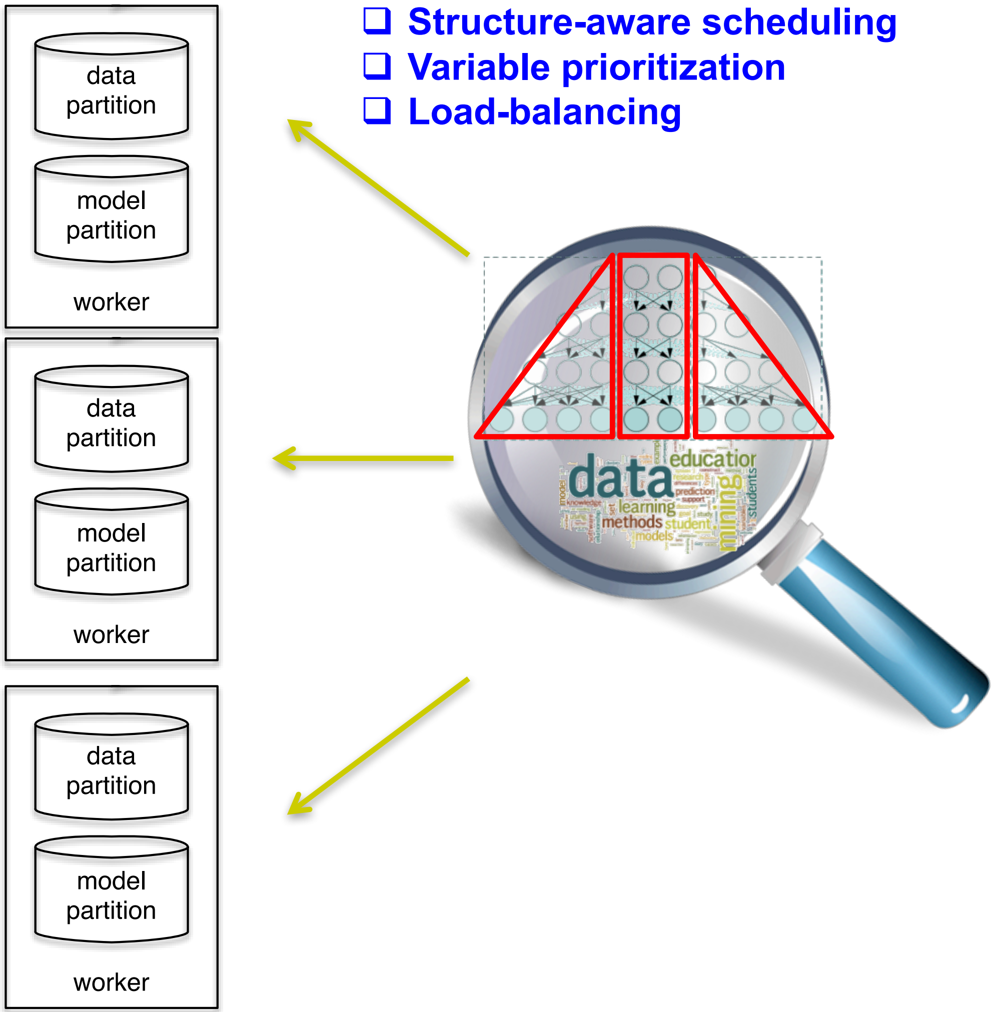


???

Is full consistency really necessary for ML?

Random Partition

Structure-Aware Parallelization (SAP) [Lee et al., 2014; Kumar et al., 2014]



- ❑ **Careful model-parallel execution:**
  - ❑ **Structure-aware scheduling**
  - ❑ **Variable prioritization**
  - ❑ **Load-balancing**
- ❑ **Simple programming:**
  - ❑ **Schedule()**
  - ❑ **Push()**
  - ❑ **Pull()**

```
schedule() {
  // Select U vars x[j] to be sent
  // to the workers for updating
  ...
  return (x[j_1], ..., x[j_U])
}
```

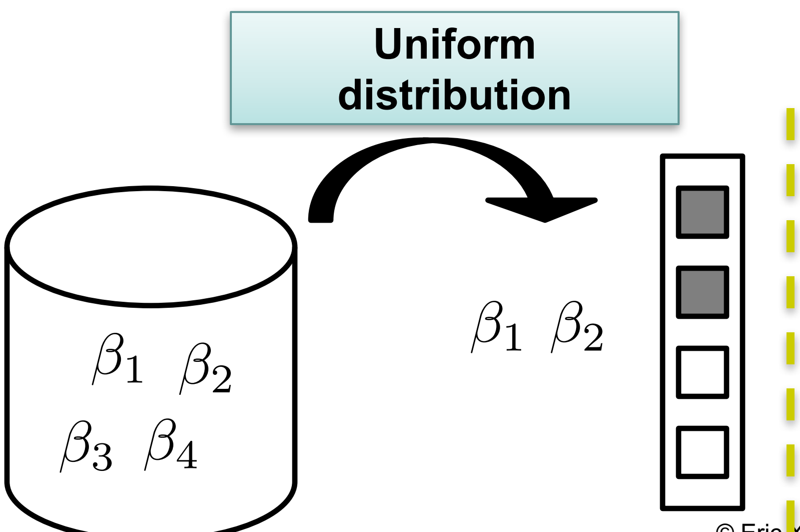
```
push(worker = p, vars = (x[j_1],...,x[j_U])) {
  // Compute partial update z for U vars x[j]
  // at worker p
  ...
  return z
}
```

```
pull(workers = [p], vars = (x[j_1],...,x[j_U]),
     updates = [z]) {
  // Use partial updates z from workers p to
  // update U vars x[j]. sync() is automatic.
  ...
}
```

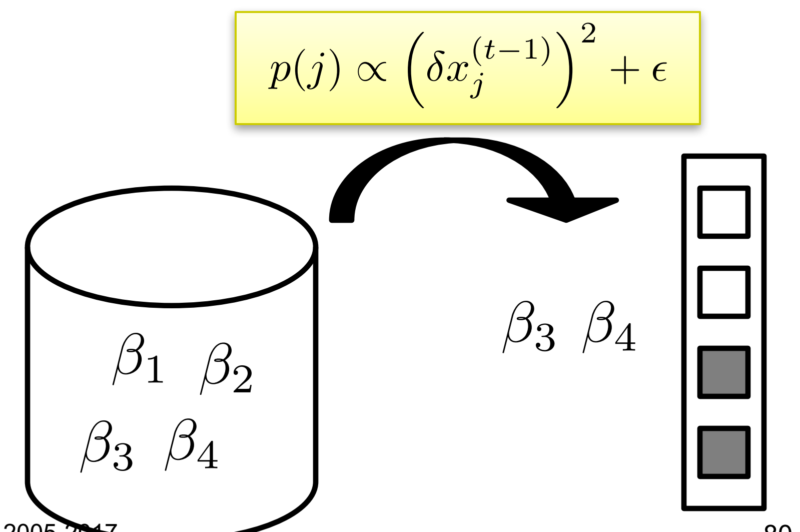
Schedule 1: Priority-based [Lee et al., 2014]

- Choose params to update based on convergence progress
  - Example: sample params with probability proportional to their recent change
  - Approximately maximizes the convergence progress per round

Shotgun [Bradley et al. 2011]

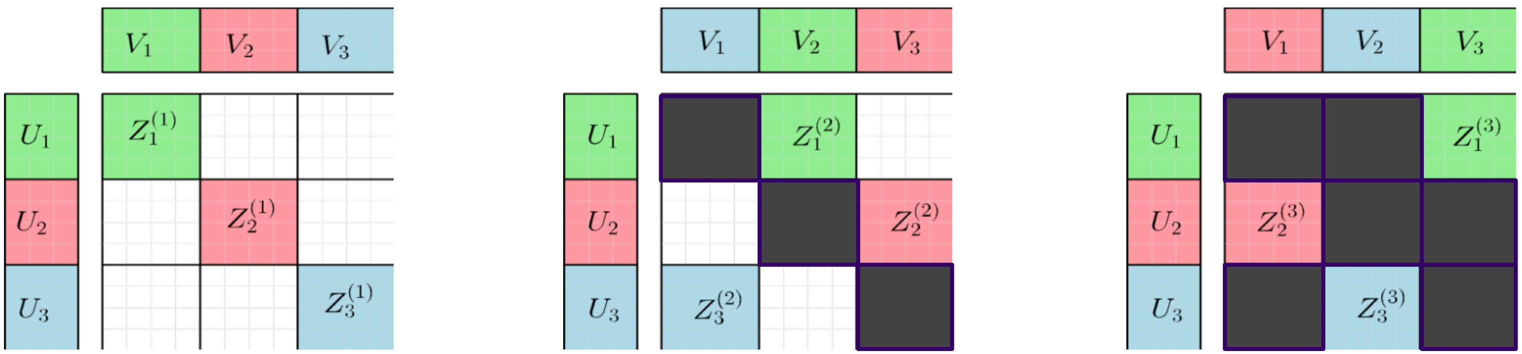


Priority-based scheduling

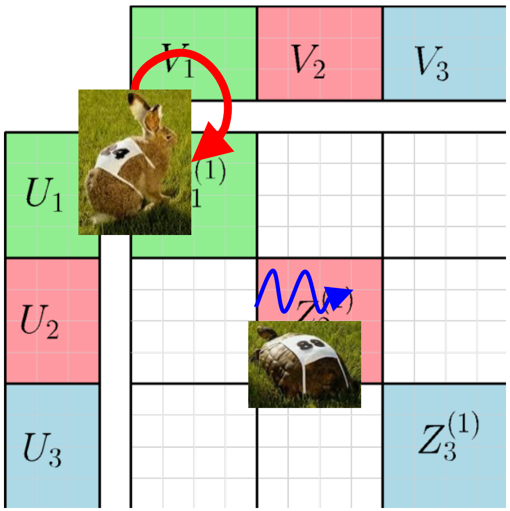


Schedule 2: Block-based (with load balancing) [Kumar et al., 2014]

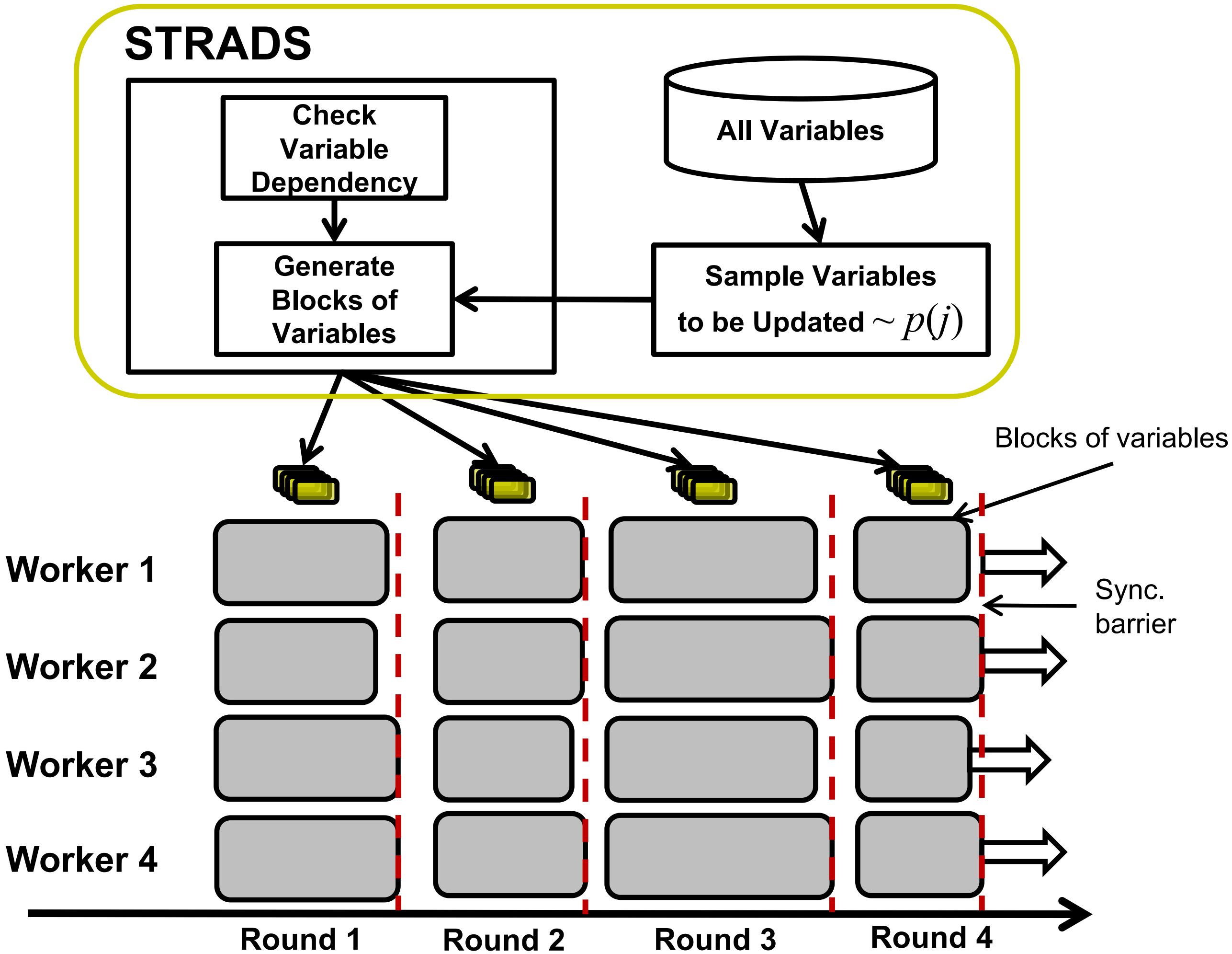
Partition data & model into *d* × *d* blocks
Run different-colored blocks in parallel



Blocks with less data/para or experience less straggling run more iterations
Automatic load-balancing + better convergence



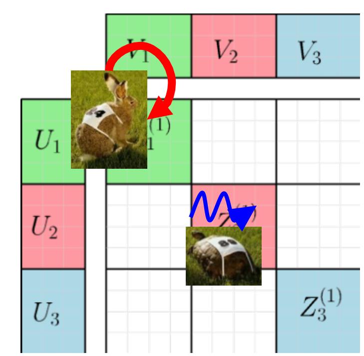
Structure-aware Dynamic Scheduler (STRADS) [Lee et al., 2014, Kumar et al., 2014]



- **Priority Scheduling**

$$\{\beta_j\} \sim \left(\delta\beta_j^{(t-1)}\right)^2 + \eta$$

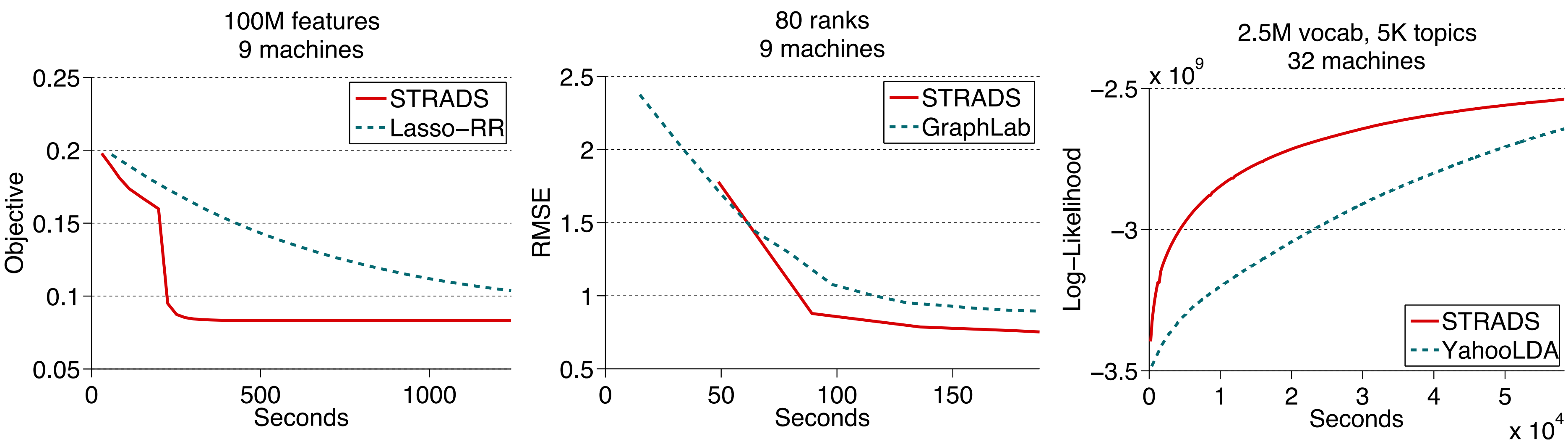
- **Block scheduling**



[Kumar, Beutel, Ho and Xing, ***Fugue: Slow-worker agnostic distributed learning***, *AISTATS 2014*]

Avoids dependent parallel updates, attains near-ideal convergence speed

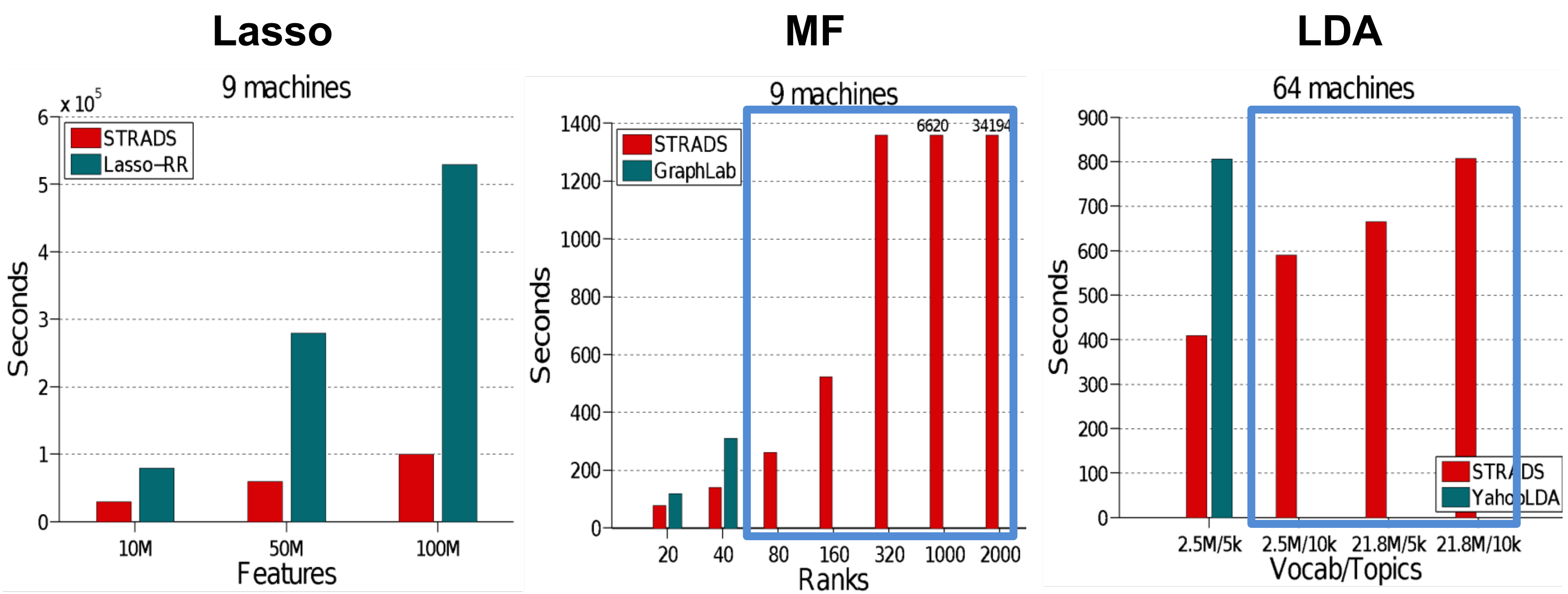
- STRADS+SAP achieves better speed and objective



Lasso **MF** **LDA**

Efficient for large models

- Model is partitioned => can run larger models on same hardware



Theory of Real Distributed ML Systems

Why study parallel ML theory?

- What sequential guarantees still hold in parallel setting?
  - Under what conditions?
- Growing body of literature for “ideal” parallel systems
  - Serializable– equivalent to single-machine execution in some sense
  - Focused on per-iteration analysis
    - Abstract away computational/comms cost
    - Predicting real-world running time requires these costs to be put back
- “Real-world” parallel systems a work in progress
  - Asynchronous or bounded-async approaches can empirically work better than synchronous approaches
    - Need additional theoretical analysis to understand why
    - Async => no serializability… why does it still work?
  - Parallelization requires data and/or model partitioning… many strategies exist
    - Want partitioning strategies that are provably correct
    - Need to determine when/where independence is violated, and what impact such violation has on algorithm correctness

Challenges in real-world distributed systems

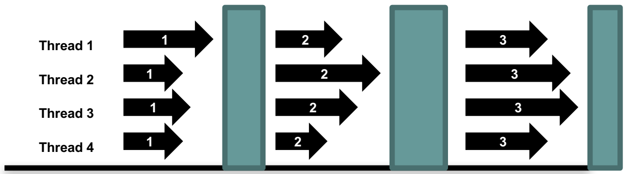
- Real-world systems need asynchronous execution and load balancing
  - Synchronous system: load imbalances => slow workers => waiting at barriers
  - Need load balancing to reduce load at slow workers
  - Need asynchronous execution so faster workers can proceed without waiting
- Solution 1: key-value stores
  - Automatically manages communication with bounded asynchronous guarantees
- Solution 2: scheduling systems
  - Automatically balances workload across workers; also performs prioritization and dependency checking

Communication strategies

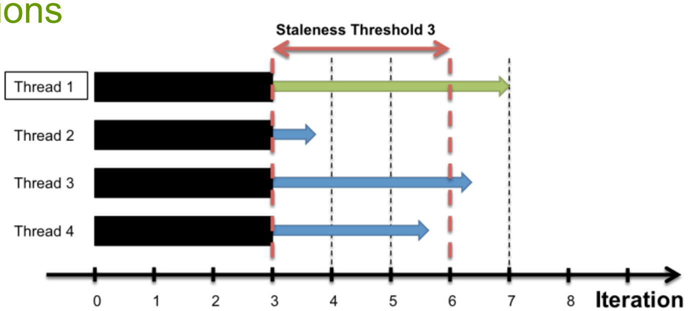
- Data parallel
  - Partition data across workers
    - Or fetch small batches of data in an online/streaming fashion
  - Communicate model as needed to workers
    - e.g. key-value store with bounded asynchronous model – theoretical consequences?
- Model parallel
  - Partition model across workers
    - Model partitions can change dynamically during execution – theoretical consequences?
  - Send data to workers as needed (e.g. from shared database)
    - Or place full copy of data on each worker (since data is immutable)
- Data + Model parallel?
  - Partition both data and model across workers
  - Wide space of strategies; need to reduce model and data communication
    - Reduce model communication by exploiting independence between variables
    - Reduce data and model communication via broadcast strategies, e.g. Halton sequence

Bridging Models for Parallel Programming

- Bulk Synchronous Parallel **[Valiant, 1990]** is a bridging model
  - Bridging model specifies how/when parallel workers should compute, and how/when workers should communicate
  - Key concept: barriers
    - No communication before barrier, only computation
    - No computation inside barrier, only communication



  - Computation is "serializable" – many sequential theoretical guarantees can be applied with no modification
- Bounded Asynchronous Parallel (BAP) bridging model
  - Key concept: bounded staleness **[Ho et al., 2013; Dai et al., 2015]**
    - Workers re-use old version of parameters, up to s iterations old – no need to barrier
    - Workers wait if parameter version older than s iterations

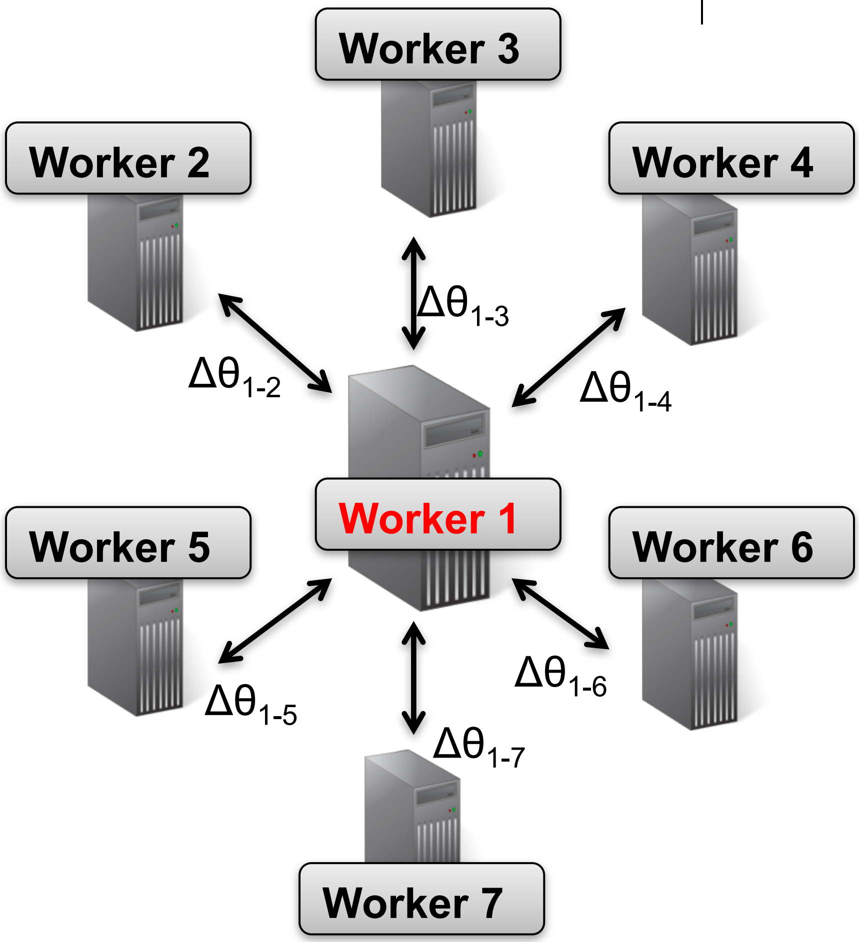


Types of Convegence Guarantees

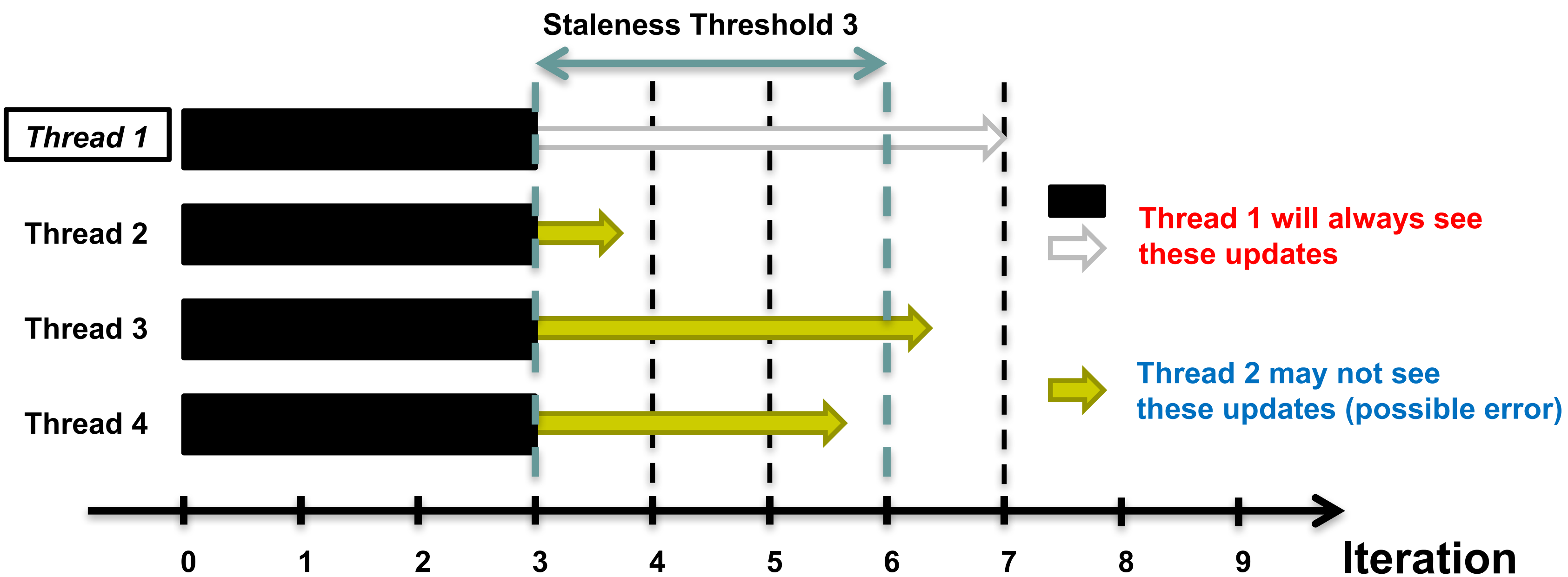
- Regret/Expectation bounds on parameters
  - Better bounds => better convergence progress per iteration
- Probabilistic bounds on parameters
  - Similar meaning to regret/expectation bounds, usually stronger in guarantee
- Variance bounds on parameters
  - Lower variance => higher stability near optimum => easier to determine convergence
- For data parallel?
- For Model parallel?
- For Data + model parallel?

BAP Data Parallel: Can we do value-bounding?

- **Idea:** limit model parameter difference $\Delta\theta_{i\text{-}j} = ||\theta_i - \theta_j||$ between machines i,j to < a threshold
- Does not work in practice!
  - To guarantee that $\Delta\theta_{i\text{-}j}$ has not exceeded the threshold, machines must wait to communicate with each other
  - No improvement over synchronous execution!
- Rather than controlling parameter difference via magnitude, what about via iteration count?
  - This is the (E)SSP communication model…



BAP Data Parallel: (E)SSP model [Ho et al., 2013; Dai et al., 2015]



Stale Synchronous Parallel (SSP)

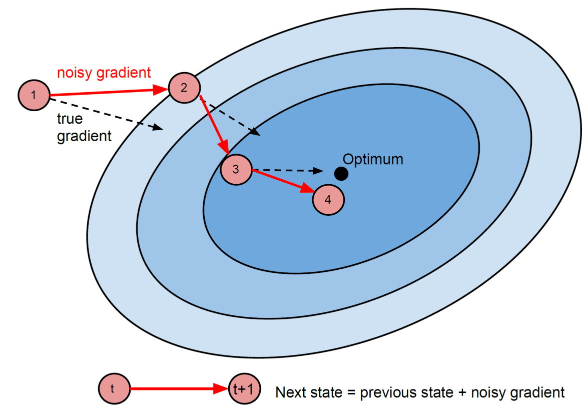
- Allow threads to run at their own pace, without synchronization
- Fastest/slowest threads not allowed to drift >S iterations apart
- Threads cache local (stale) versions of the parameters, to reduce network syncing

Consequence:

- Asynchronous-like speed, BSP-like ML correctness guarantees
- Guaranteed age bound (staleness) on reads
- Contrast: no-age-guarantee Eventual Consistency seen in Cassandra, Memcached

BAP Data Parallel: (E)SSP Regret Bound [Ho et al., 2013]

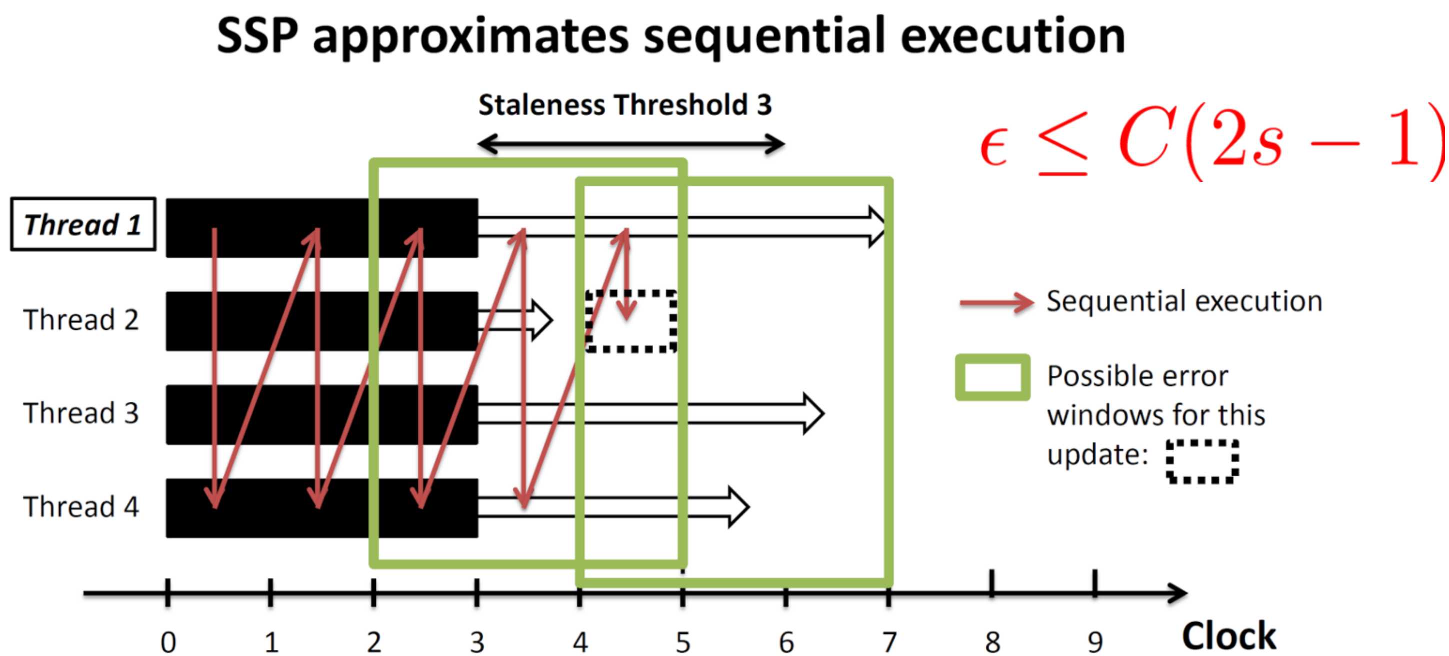
- **Goal:** minimize convex $f(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x})$
  (Example: Stochastic Gradient)
  - L-Lipschitz, problem diameter bounded by F^2
  - Staleness s, using P threads across all machines
  - Use step size $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$
- **(E)SSP converges according to**
  - Where T is the number of iterations



$$R[\mathbf{X}] := \overbrace{\left[\frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{\mathbf{x}}_t)\right] - f(\mathbf{x}^*)}^{\textbf{Difference between SSP estimate and true optimum}} \leq 4FL\sqrt{\frac{2(s+1)P}{T}}$$

- Note the RHS interrelation between (L, F) and (s, P)
  - An interaction between model and systems parameters
- Stronger guarantees on means and variances can also be proven

Intuition:
Why does (E)SSP converge?



- Number of missing updates bounded
  - Partial, but bounded, loss of serializability
- Hence numeric error in parameter also bounded
- Later in this tutorial – formal theorem

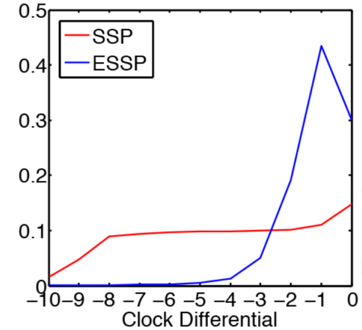
SSP versus ESSP: What is the difference?

- ESSP is a systems improvement over SSP communication
  - Same maximum staleness guarantee as SSP
  - Whereas SSP waits until the last second to communicate…
  - … ESSP communicates updates as early as possible

- What impact does ESSP have on convergence speed and stability?

BAP Data Parallel: (E)SSP Probability Bound

[Dai et al., 2015]



Let real staleness observed by system be γ_t

Let its mean, variance be $\mu_\gamma = \mathbb{E}[\gamma_t]$, $\sigma_\gamma = var(\gamma_t)$

Theorem: Given L-Lipschitz objective f_t and stepsize h_t,

$$P\left[\underbrace{\frac{R[X]}{T}}_{\text{Gap between current estimate and optimum}} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + \underbrace{2\eta L^2 \mu_\gamma}_{\text{Penalty due to high avg. staleness } u_{stale}}\right) \geq \tau\right] \leq \exp\left\{\frac{-T\tau^2}{\underbrace{2\bar{\eta}_T \sigma_\gamma}_{\text{Penalty due to high staleness var. } \sigma_{stale}} + \frac{2}{3}\eta L^2(2s+1)P\tau}\right\}$$

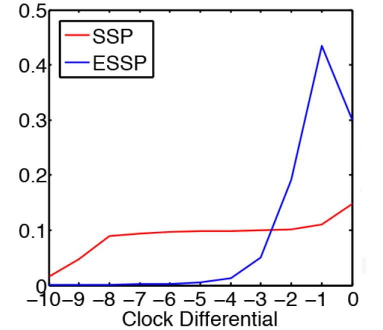
$$R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*) \qquad \bar{\eta}_T = \frac{\eta^2 L^4 (\ln T + 1)}{T} = o(T)$$

Explanation: the (E)SSP distance between true optima and current estimate decreases exponentially with more iterations. *Lower staleness mean, variance μ_γ, σ_γ improve the convergence rate.*

Take-away: controlling staleness mean μ_γ, variance σ_γ (on top of max staleness s) is needed for faster ML convergence, which ESSP does.

BAP Data Parallel: (E)SSP Variance Bound

[Dai et al., 2015]



Theorem: the variance in the (E)SSP estimate is

$$\mathrm{Var}_{t+1} = \mathrm{Var}_{t} - 2\eta_t cov(\boldsymbol{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) + \mathcal{O}(\eta_t \xi_t) + \mathcal{O}(\eta_t^2 \rho_t^2) + \mathcal{O}^*_{\gamma_t}$$

where

$$cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T \boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$$

and $\mathcal{O}^*_{\gamma_t}$ represents 5th order or higher terms in γ_t

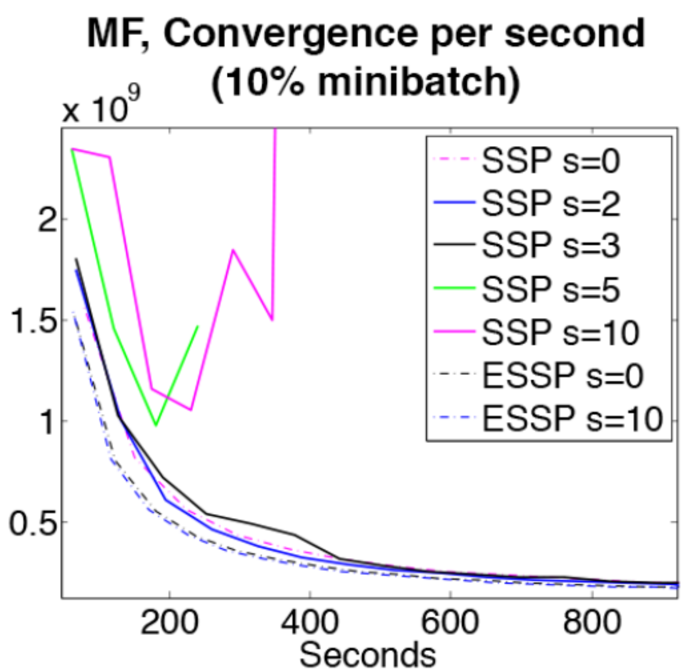
Explanation: The variance in the (E)SSP parameter estimate monotonically decreases when close to an optimum.

Lower (E)SSP staleness γ_t *=> Lower variance in parameter => Less oscillation in parameter => More confidence in estimate quality and stopping criterion.*

Take-away: Lower average staleness (via ESSP) not only improves convergence speed, but also yields better parameter estimates

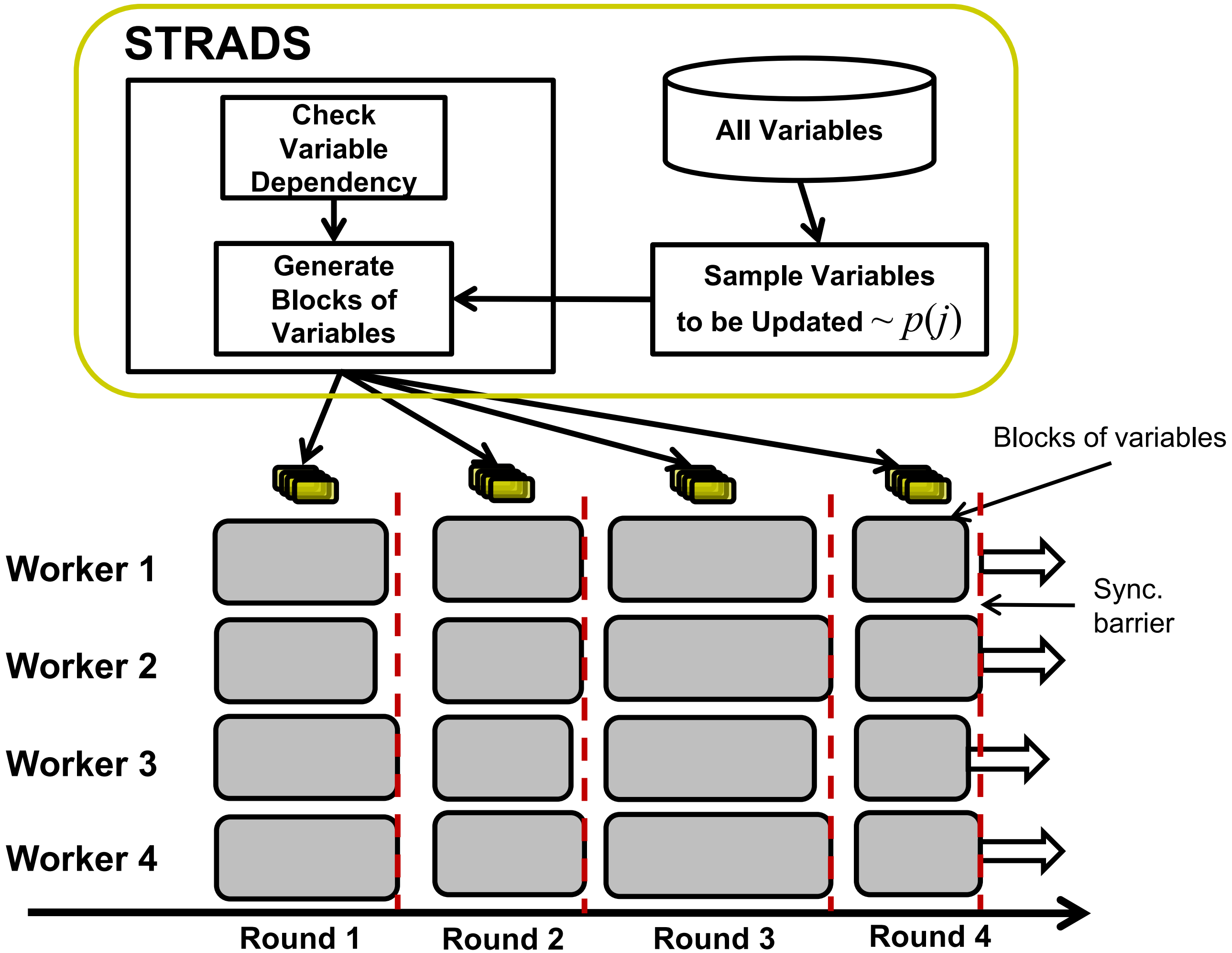
ESSP vs SSP: Increased stability helps empirical performance

- Low-staleness SSP and ESSP converge equally well
- But at higher staleness, ESSP is more stable than SSP
  - ESSP communicates updates early, whereas SSP waits until the last second
  - ESSP better suited to real-world clusters, with straggler and multi-user issues



Scheduled Model Parallel: Dynamic/Block Scheduling

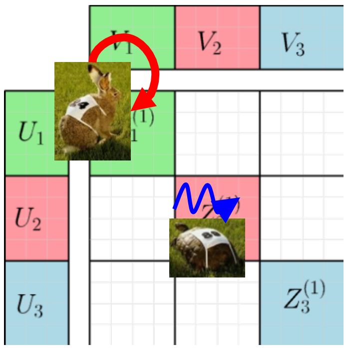
[Lee et al. 2014, Kumar et al. 2014]



- **Priority Scheduling**

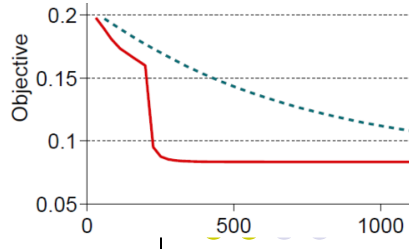
$$\{\beta_j\} \sim \left(\delta\beta_j^{(t-1)}\right)^2 + \eta$$

- **Block scheduling**



Scheduled Model Parallel:
Dynamic Scheduling Expectation Bound

[Lee et al. 2014]



- **Goal:** solve sparse regression problem $\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda \sum_j |\beta_j|$
  - Via coordinate descent over "SAP blocks" $X^{(1)}, X^{(2)}, \ldots, X^{(B)}$
    - $X^{(b)}$ are the data columns (features) in block (b)
  - P parallel workers, M-dimensional data
  - ρ = Spectral Radius[BlockDiag[$(X^{(1)})^T X^{(1)}, \ldots, (X^{(t)})^T X^{(t)}$]]; this block-diagonal matrix quantifies the maximum level of correlation (and hence problem difficulty) within all the SAP blocks $X^{(1)}, X^{(2)}, \ldots, X^{(t)}$
- **SAP converges according to**
  - Where t is # of iterations

$$\mathbb{E}\left[\overbrace{f(X^{(t)}) - f(X^*)}^{\text{Gap between current parameter estimate and optimum}}\right] \leq \frac{\overbrace{\mathcal{O}(M)}^{\text{SAP explicitly minimizes } \rho\text{, ensuring as close to } 1/P \text{ convergence as possible}}}{P - \boxed{\frac{\mathcal{O}(P^2\rho)}{M}}} \frac{1}{t} = \mathcal{O}\left(\frac{1}{Pt}\right)$$

- **Take-away:** SAP minimizes ρ by searching for feature subsets $X^{(1)}, X^{(2)}, \ldots, X^{(B)}$ without cross-correlation => as close to P-fold speedup as possible

Scheduled Model Parallel:

Dynamic Scheduling Expectation Bound is near-ideal

[Xing et al. 2015]

Let $S^{ideal}()$ be an ideal model-parallel schedule

Let $\beta^{(t)}_{ideal}$ be the parameter trajectory due to ideal scheduling

Let $\beta^{(t)}_{dyn}$ be the parameter trajectory due to SAP scheduling

Theorem: After *t* iterations, we have

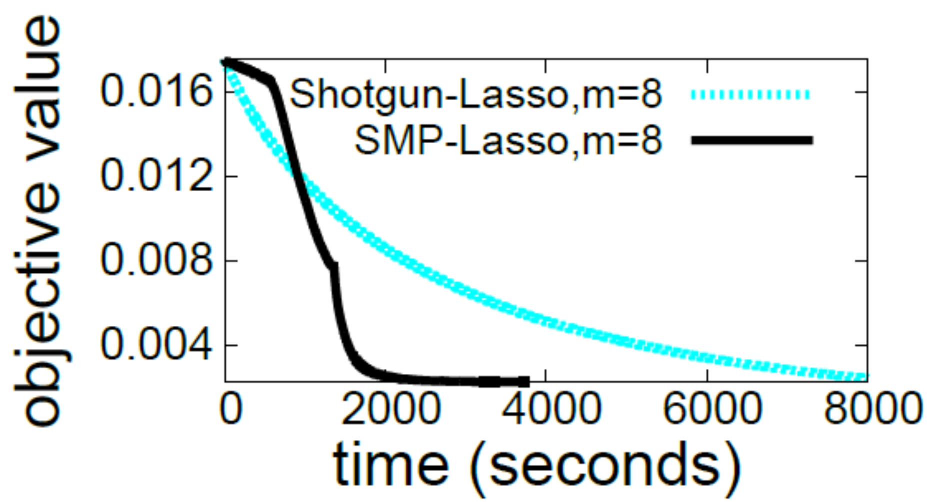
$$E[||\beta^{(t)}_{ideal} - \beta^{(t)}_{dyn}||] \leq C \frac{2M}{(t+1)^2} \mathbf{X}^\top \mathbf{X}$$

Explanation: *Under dynamic scheduling, algorithmic progress is nearly as good as ideal model-parallelism.*

Intuitively, this is because both ideal and SAP model-parallelism minimize the parameter dependencies between parallel workers.

Scheduled Model Parallel:
Dynamic Scheduling Empirical Performance

- Dynamic Scheduling for Lasso regression (SMP-Lasso): almost-ideal convergence rate, much faster than random scheduling (Shotgun-Lasso)

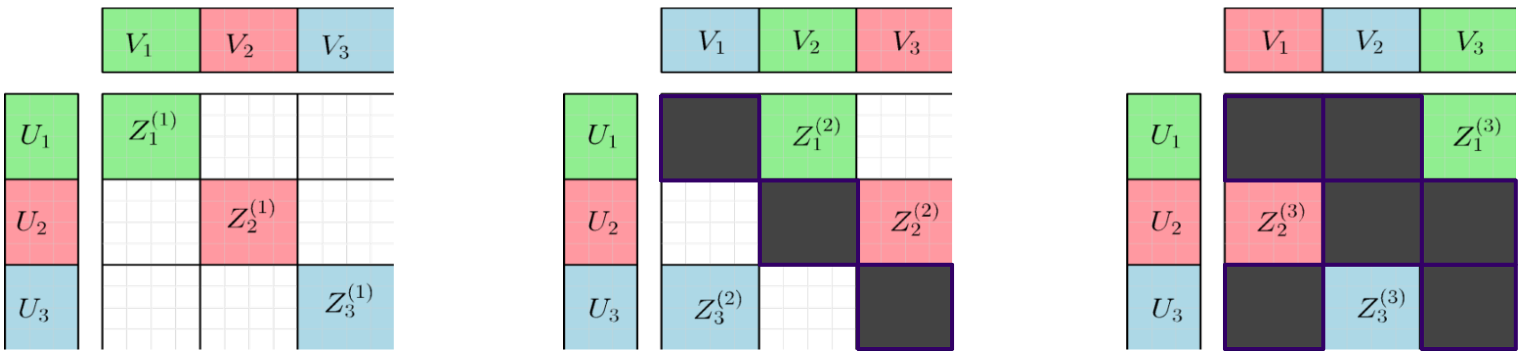


Scheduled Data+Model Parallel:

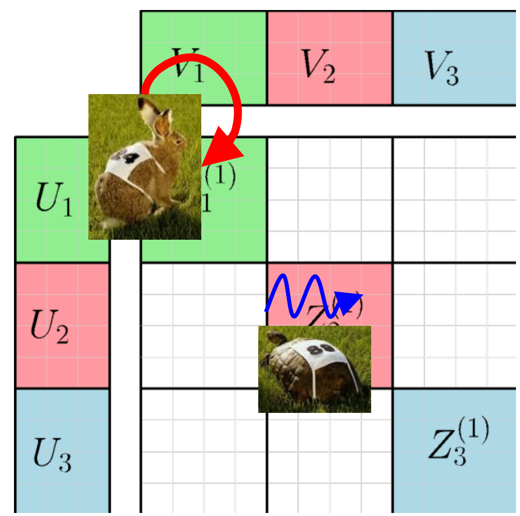
Block-based Scheduling (with load balancing)

[Kumar et al. 2014]

Partition data & model into *d* × *d* blocks
Run different-colored blocks in parallel



Blocks with less data/para or experience less straggling run more iterations
Automatic load-balancing + better convergence



Scheduled Data+Model Parallel:
Block-based Scheduling Variance Bound 1
[Kumar et al. 2014]

- Variance between iterations S_n+1 and S_n is:

$$
\begin{aligned}
&Var(\Psi_{S_{n+1}}) \\
&= Var(\Psi_{S_n}) - \boxed{2\eta_{S_n}} \sum_{i=1}^{w} n_i \Omega_0^i Var(\psi_{S_n}^i) \\
&\quad - \boxed{2\eta_{S_n}} \sum_{i=1}^{w} n_i \Omega_0^i CoVar(\psi_{S_n}^i, \bar{\delta}_{S_n}^i) + \boxed{\eta_{S_n}^2} \sum_{i=1}^{w} n_i \Omega_1^i + \boxed{\mathcal{O}(\Delta_{S_n})}
\end{aligned}
$$

- Explanation:
  - higher order terms (red) are negligible
  - => parameter variance decreases every iteration
- Every iteration, the parameter estimates become more stable

Scheduled Data+Model Parallel:
Block-based Scheduling Variance Bound 2
[Kumar et al. 2014]

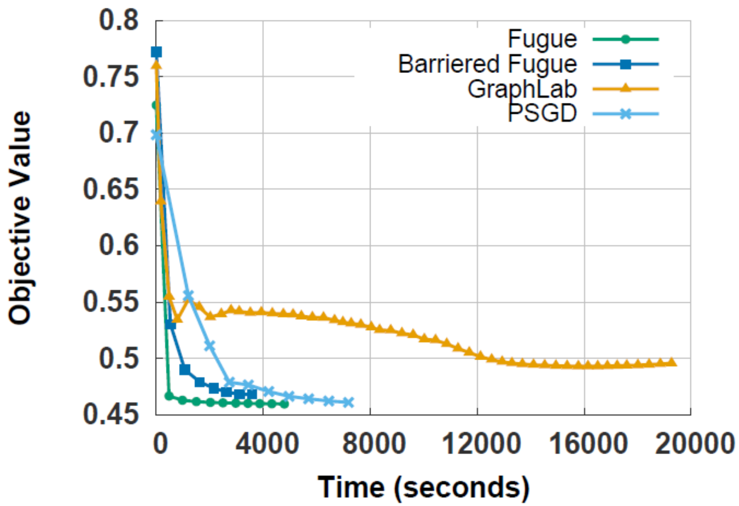
- Intra-block variance: Within blocks, suppose we update the parameters ψ using n_i data points. Then, variance of ψ after those n_i updates is:

$$
\begin{aligned}
Var(\psi^{t+n_i}) =& Var(\psi^t) - 2\eta_t n_i \Omega_0 (Var(\psi^t)) \\
& - 2\eta_t n_i \Omega_0 CoVar(\psi_t, \bar{\delta}_t) + \boxed{\eta_t^2 n_i \Omega_1} \\
& \boxed{+ \underbrace{\mathcal{O}(\eta_t^2 \rho_t) + \mathcal{O}(\eta_t \rho_t^2) + \mathcal{O}(\eta_t^3) + \mathcal{O}(\eta_t^2 \rho_t^2)}_{\Delta_t}}
\end{aligned}
$$

- Explanation:
  - Higher order terms (red) are negligible
  - => doing more updates within each block decreases parameter variance, leading to more stable convergence
- Load balancing by doing extra updates is effective

Scheduled Data+Model Parallel:
Block-Scheduling Empirical Performance

- Slow-worker Agnostic Block-Scheduling (Fugue) faster than:
  - Embarrassingly Parallel SGD (PSGD)
  - Non slow-worker Agnostic Block-Scheduling (Barriered Fugue)
- Slow-worker Agnostic Block-Scheduling converges to a better optimum than asynchronous GraphLab
  - Reason: more stable convergence due to block-scheduling
- Task: Imagenet Dictionary Learning
  - 630k images, 1k features



BAP Model-Parallel Guarantees

- Model-parallel under synchronous setting:
  - Dynamic scheduling
  - Slow-worker block-based scheduling
- Synchronous slow-worker problem solved by:
  - Load balancing (for dynamic scheduling)
  - Allow additional iters while waiting for other workers (slow-worker scheduling)
- Work in progress: theoretical guarantees for bounded-async model-parallel execution
  - Intuition: model-parallel sub-problems are nearly independent (thanks to scheduling)
  - Perhaps better per-iteration convergence than bounded-async data-parallel learning?

Summary

- ML Programs different from Operational Programs
  - Error tolerant – allows bounded asynchronous execution
  - Dependency structures – will slow down convergence if ignored
  - Non-uniform convergence – can allocate resources more efficiently
- Distributed Systems are Challenging
  - Uneven machine performance – must deal with slow workers/stragglers
  - Communication bottlenecks – due to iterative algo updates on Big Models
- Data, Model-parallelism to understand ML algorithms, and build distributed ML systems
  - How to distribute ML computation?
  - How to bridge ML computation and communication?
  - How to perform ML communication?
- Theory to understand why/how distributed ML systems work