Big ML Software for Modern ML Algorithms

Qirong Ho^ and Eric P. Xing*

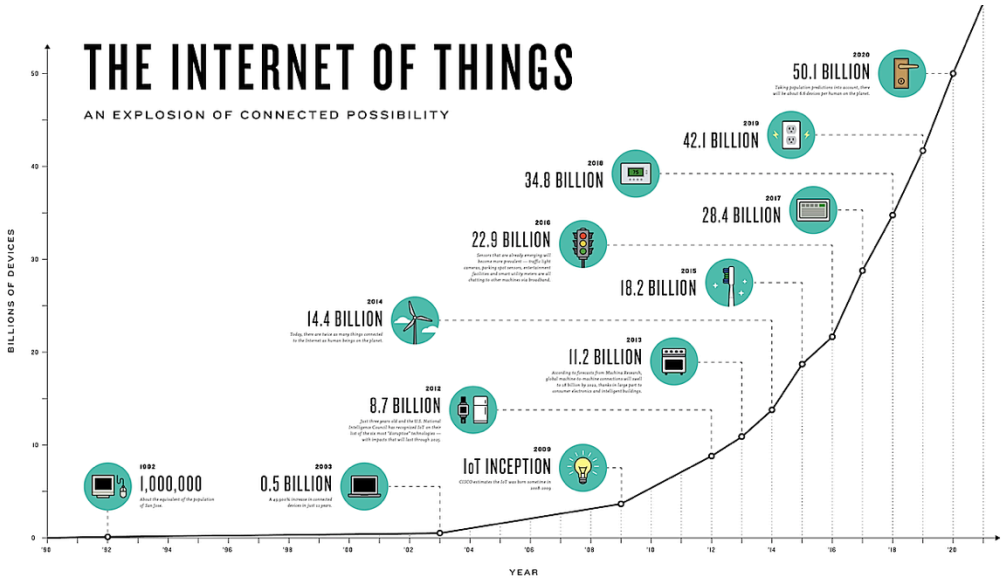
^Institute for Infocomm Research, A*STAR
*Carnegie Mellon University

The First Encounter of Science with Big Data

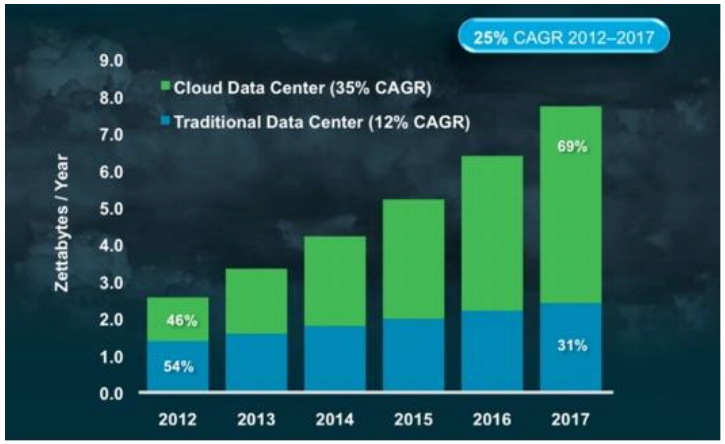
Trees Falling in the Forest

- Nobody knows what's in data unless it has been processed and analyzed
  - Need a scalable way to automatically search, digest, index, and understand contents

Challenge #1 – Massive Data Scale



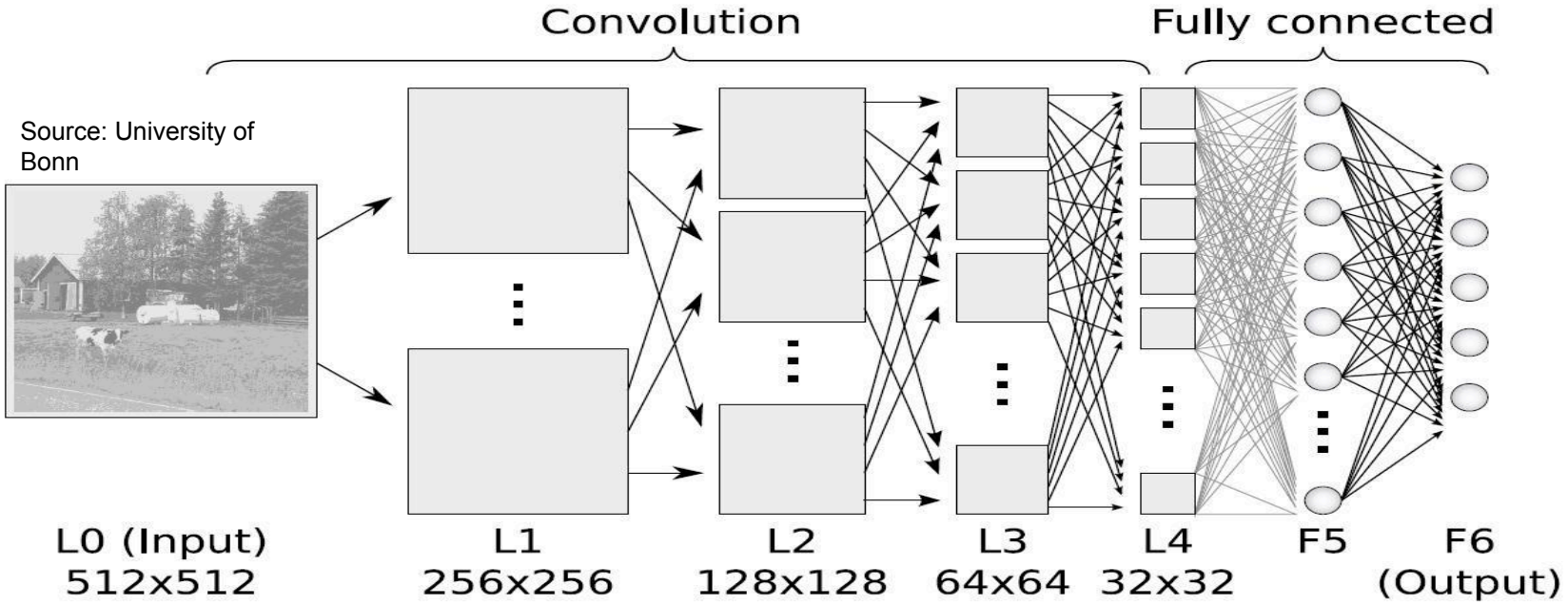
Source: The Connectivist



Source: Cisco Global Cloud Index

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

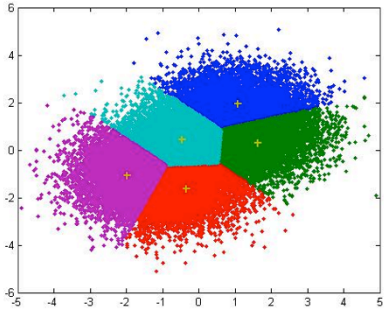
Challenge #2 – Gigantic Model Size



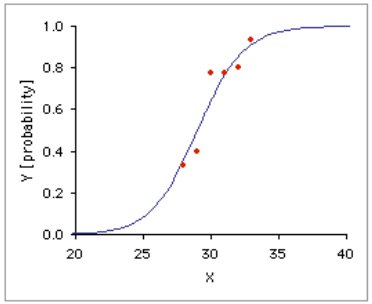
Big Data needs Big Models to extract understanding
But ML models with >1 trillion params also won't fit!

Challenge #3 – Inadequate ML library

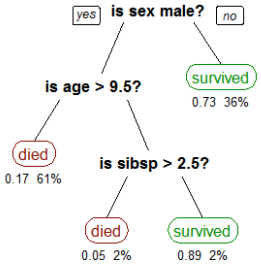
Classic ML algorithms used for decades



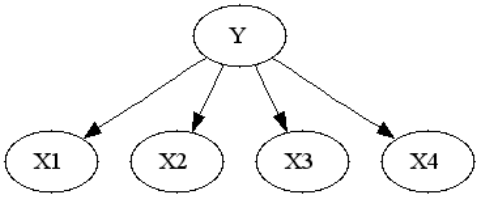
K-means



Logistic regression



Decision trees

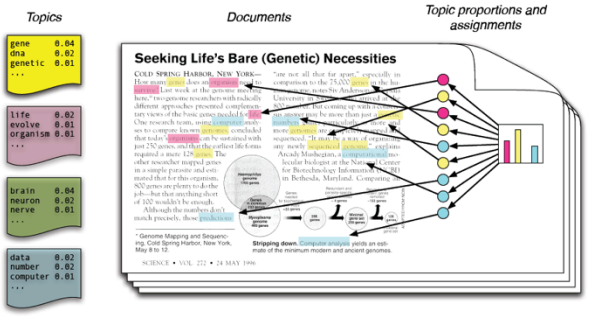


Naive Bayes

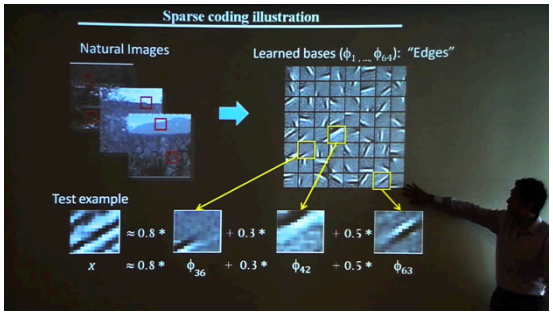
And many more...

Challenge #3 – Inadequate ML library

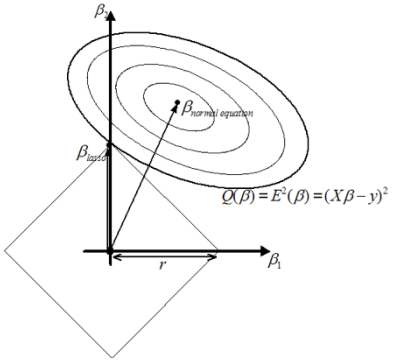
But new tasks have emerged; demand today's ML algos



Topic models
make sense of documents



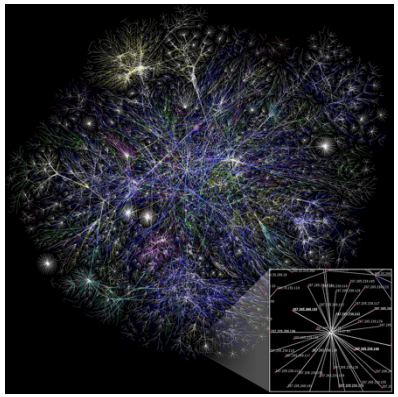
Deep learning
make sense of images, audio



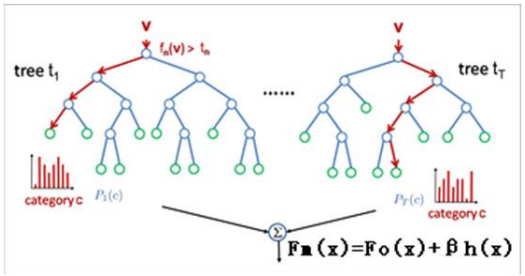
Lasso regression
find significant genes,
predict stock market

Challenge #3 – Inadequate ML library

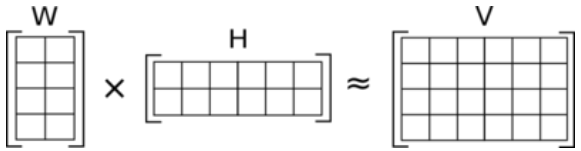
But new tasks have emerged; demand today's ML algos



Latent space network models
find communities in networks



Tree ensembles
better than decision trees



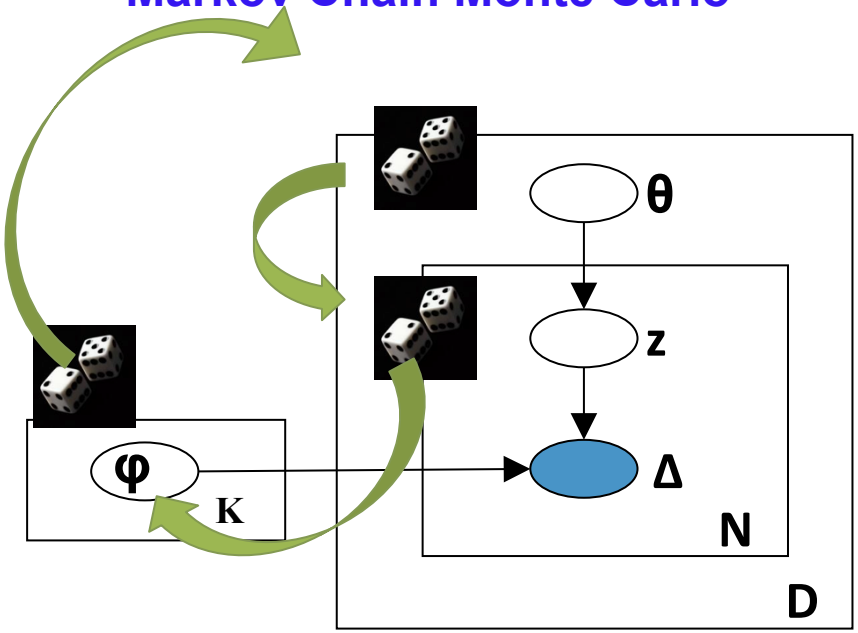
Constrained matrix factorization
collaborative filtering when negative values just don't make sense

Where are these new algos in today's Big Data tools?

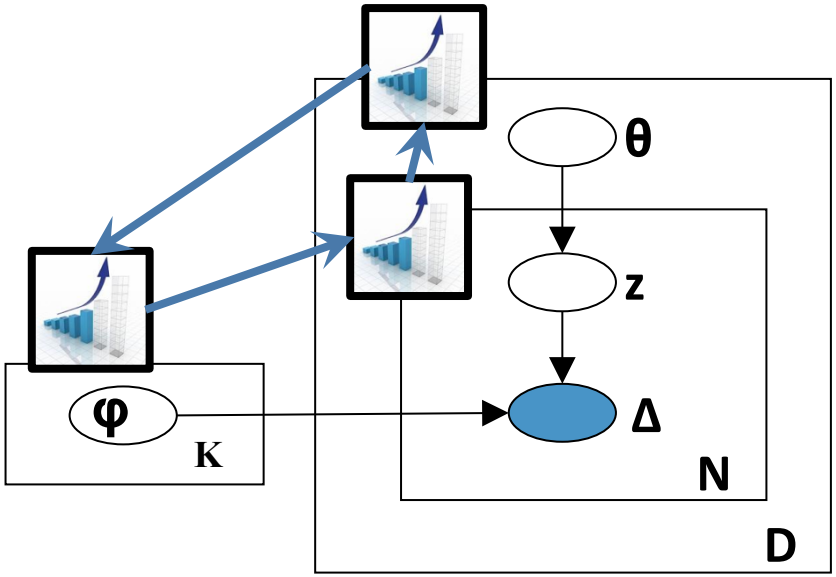
Challenge #4 – ML algos iterative-convergent

ML algorithms = "engine" to solve ML models

Markov Chain Monte Carlo



Optimization



Hadoop not suited to iterative ML

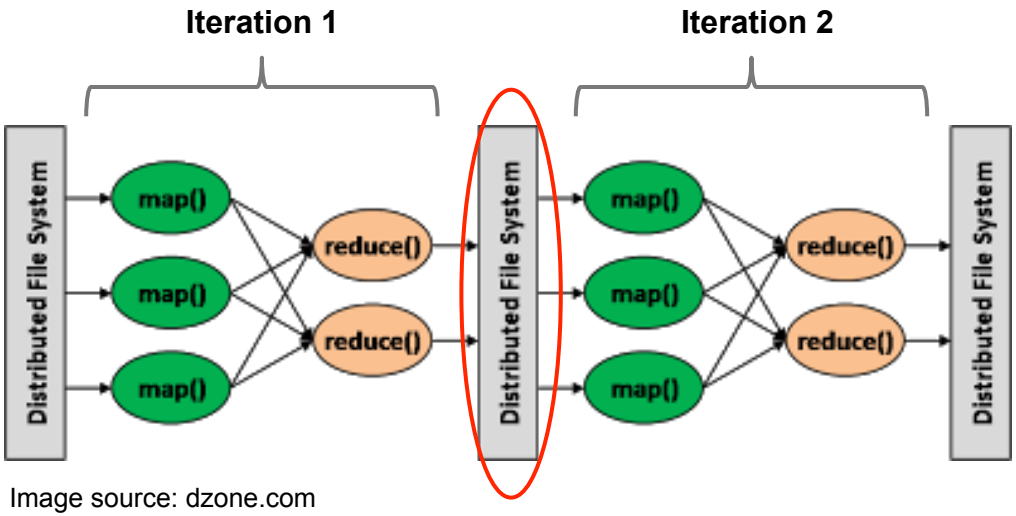


Image source: dzone.com

HDFS Bottleneck

ML algos iterative-convergent, but Hadoop not efficient at iterative programs
Iterative program => need many map-reduce phases => HDFS disk I/O becomes bottleneck
Alternatives to Hadoop later in this tutorial...

Why need new Big ML systems?

ML practitioner's view

- Want correctness, fewer iters to converge

Why need new Big ML systems?

ML practitioner's view

- Want correctness, fewer iters to converge
- … but assume an ideal system

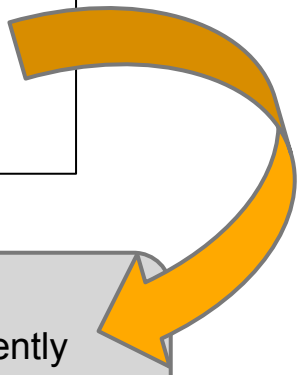
```
for (t = 1 to T) {
  doThings()
  parallelUpdate(x,θ)
  doOtherThings()
}
```

Why need new Big ML systems?

ML practitioner's view

- Want correctness, fewer iters to converge
- … but assume an ideal system

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



- Oversimplify systems issues
  - e.g. machines perform consistently
  - e.g. can sync parameters any time

Why need new Big ML systems?

Systems view

- Want more iters executed per second

Why need new Big ML systems?

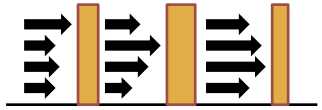
Systems view

- Want more iters executed per second
- … but assume ML algo is a black box

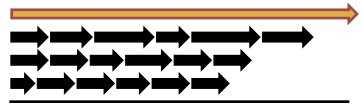
Why need new Big ML systems?

Systems view

- Want more iters executed per second
- … but assume ML algo is a black box
- … or assume ML algo "still works" under different execution models



Slow-but-correct
Bulk Sync. Parallel

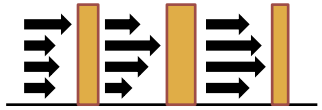


Fast-but-unstable
Asynchronous Parallel

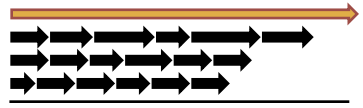
Why need new Big ML systems?

Systems view

- Want more iters executed per second
- … but assume ML algo is a black box
- … or assume ML algo "still works" under different execution models



Slow-but-correct
Bulk Sync. Parallel



Fast-but-unstable
Asynchronous Parallel

- Oversimplify ML issues
  - e.g. assume ML algo "works" without proof
  - e.g. ML algo "easy to rewrite" in chosen abstraction: MapR, vertex program, etc.

Why need new Big ML systems?

ML practitioner's view

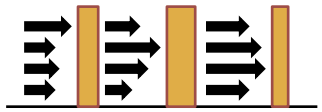
- Want correctness, fewer iters to converge
- … but assume an ideal system

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

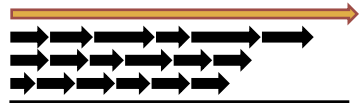
- Oversimplify systems issues
  - e.g. machines perform consistently
  - e.g. can sync parameters any time

Systems view

- Want more iters executed per second
- … but assume ML algo is a black box
- … or assume ML algo "still works" under different execution models



Slow-but-correct
Bulk Sync. Parallel



Fast-but-unstable
Asynchronous Parallel

- Oversimplify ML issues
  - e.g. assume ML algo "works" without proof
  - e.g. ML algo "easy to rewrite" in chosen abstraction: MapR, vertex program, etc.

Why need new Big ML systems?

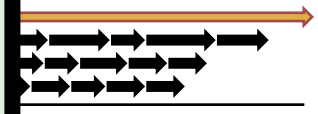
ML practitioner's view

- Want correctness, fewer iters to converge
- … but assume an ideal system

```
for (t = 1 to
  doThings()
  paralle
  doOtherThir
}
```

Systems view

- Want more iters executed per second
- … but assume ML algo is a black box
- "still works" under
  dels



ast-but-unstable
nchronous Parallel

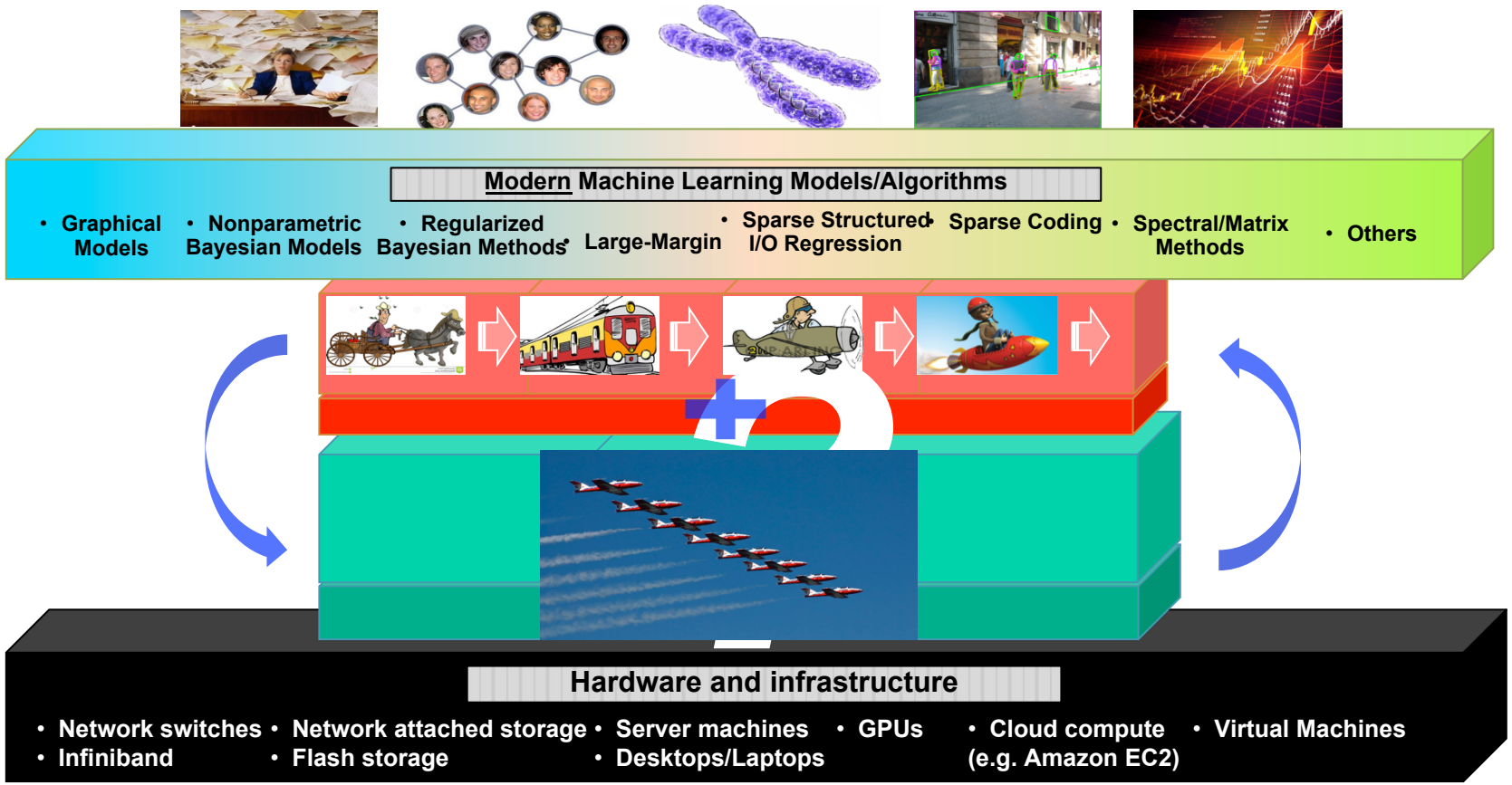
Alone, neither side has full picture ...

New opportunities exist in the middle!

- Oversimplify systems issues
  - e.g. machines perform consistently
  - e.g. can sync parameters any time

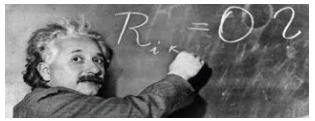
- Oversimplify ML issues
  - e.g. assume ML algo "works" without proof
  - e.g. ML algo "easy to rewrite" in chosen abstraction: MapR, vertex program, etc.

Solution: An Alg/Sys **INTERFACE** for Big ML



The Big ML "Stack" - More than just software

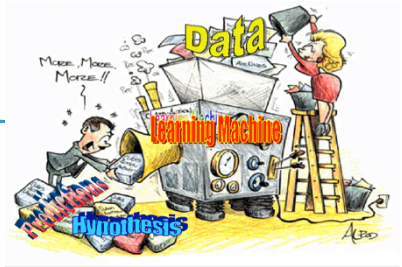
Theory: Degree of parallelism, convergence analysis, sub-sample complexity …



Representation: Compact, informative features

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

Algorithm: Parallelizable and stochastic MCMC, VI, Opt, Spectral learning …

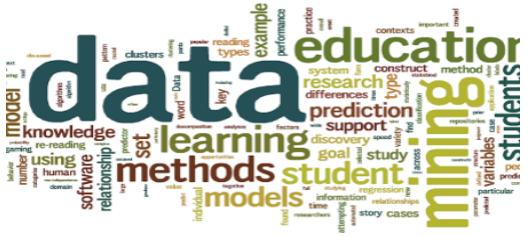


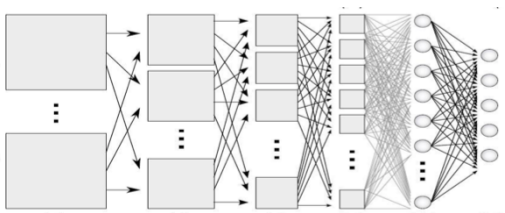
Programming model & Interface:
High: Toolkits
Med: Matlab/R/Python
Low: C/Java

System: Distributed architecture: DFS, parameter server, task scheduler…

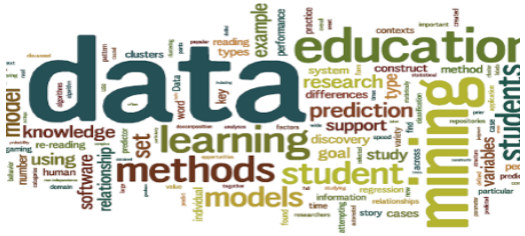
Hardware: GPU, flash storage, cloud …

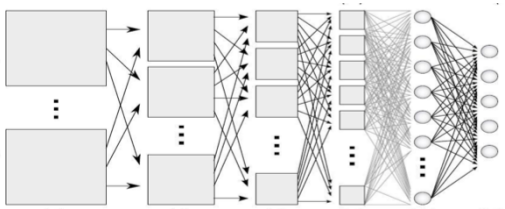
2 parallelization strategies





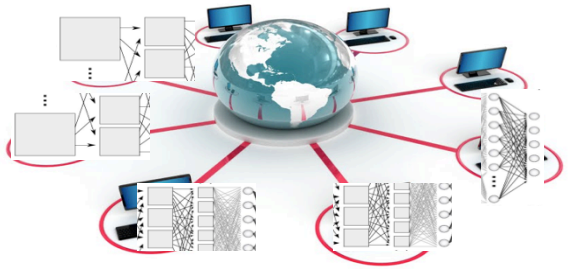
2 parallelization strategies





Data Parallel

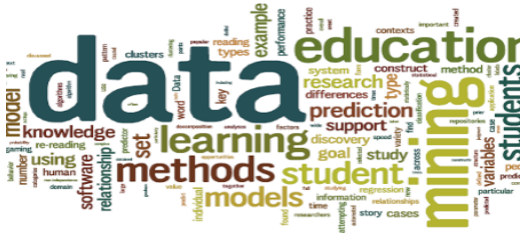
Model Parallel

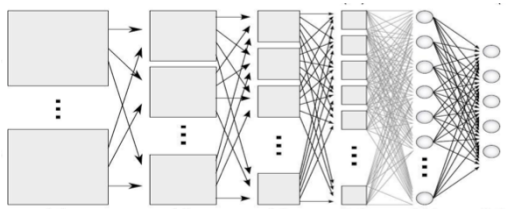


2 parallelization strategies

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

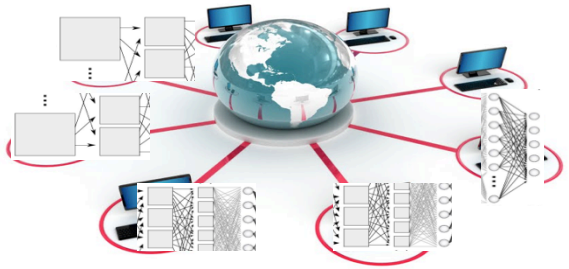
New Model = Old Model + Update(Data)





Data Parallel

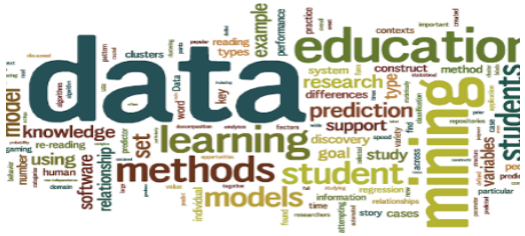
Model Parallel

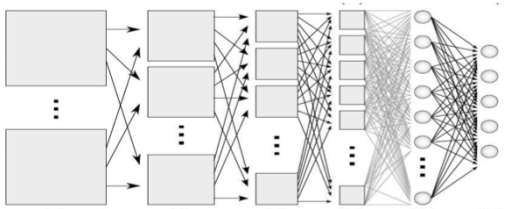


2 parallelization strategies

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

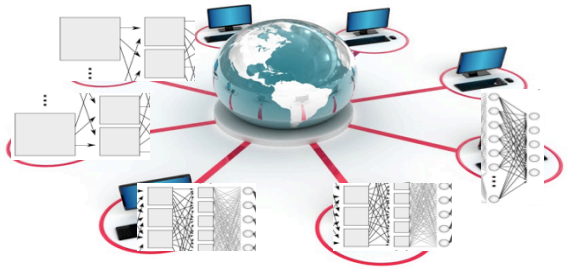
New Model = Old Model + Update(Data)





Data Parallel

Model Parallel

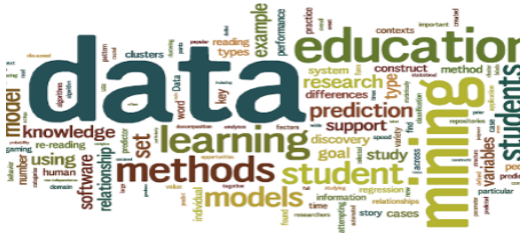


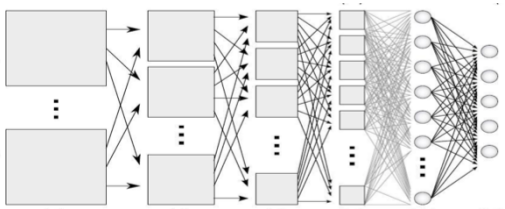
$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$

2 parallelization strategies

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

New Model = Old Model + Update(Data)

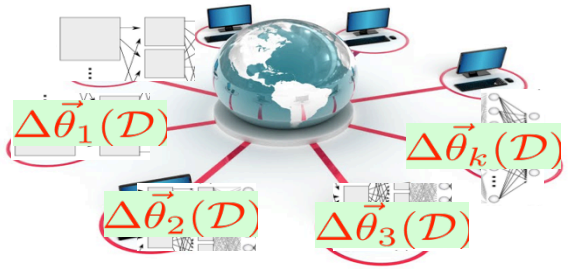




Data Parallel

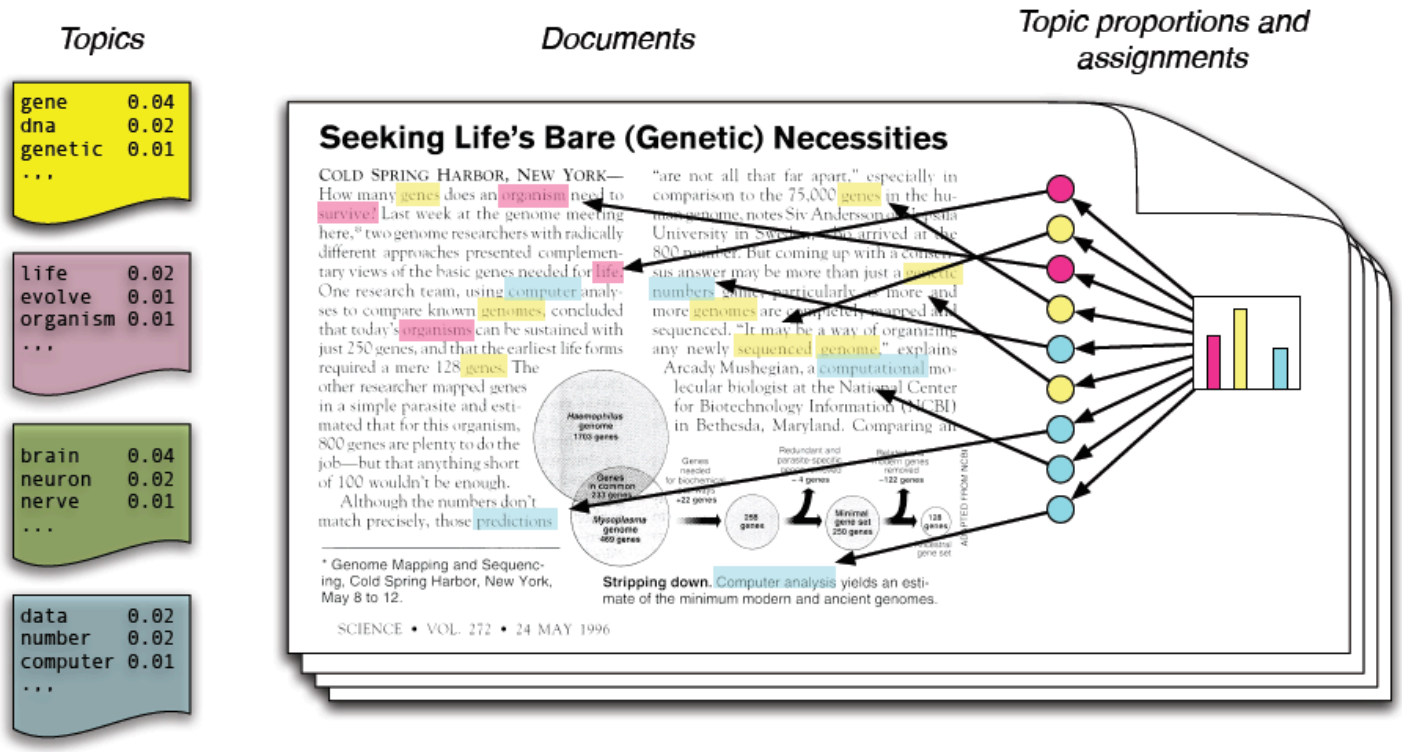
$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$

Model Parallel



$$\vec{\theta} \equiv [\vec{\theta_1}^{\mathrm{T}}, \vec{\theta_2}^{\mathrm{T}}, \ldots, \vec{\theta_k}^{\mathrm{T}}\}^{\mathrm{T}}$$

Modern ML Parallelism: Topic Models

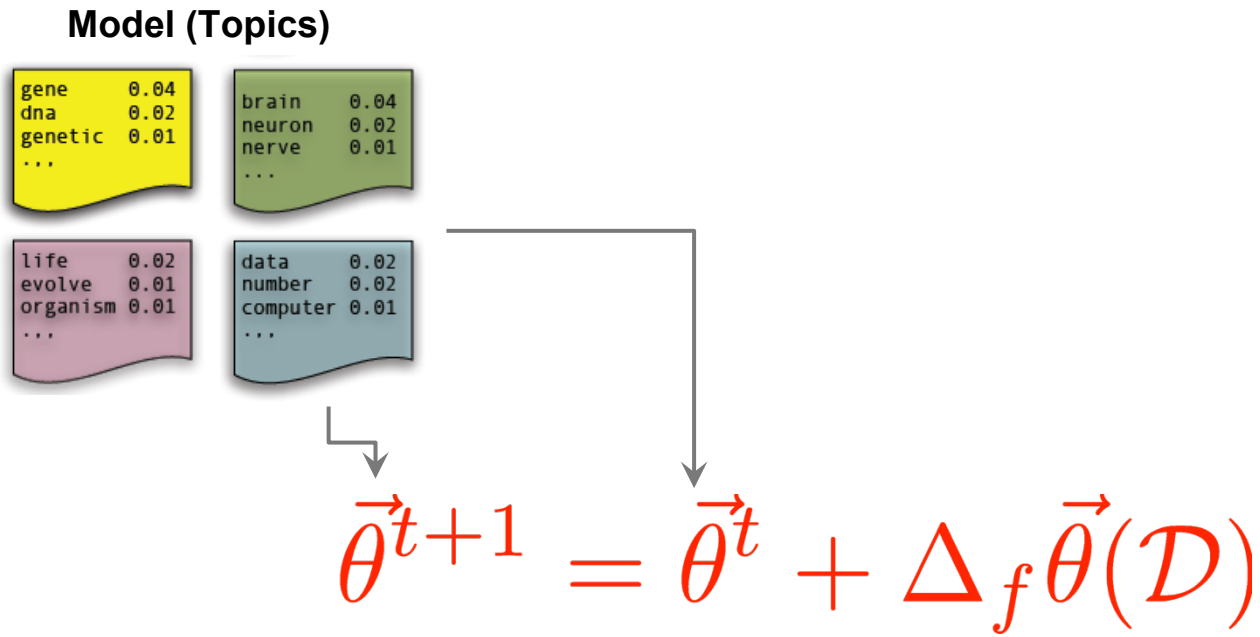


Source: D. Blei (2012)

Modern ML Parallelism: Topic Models

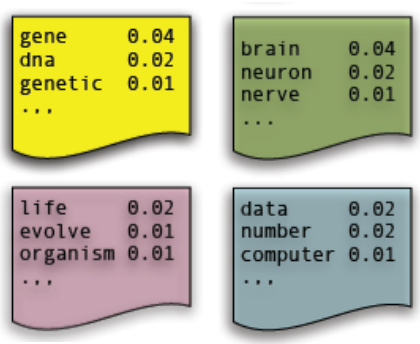
$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

Modern ML Parallelism: Topic Models

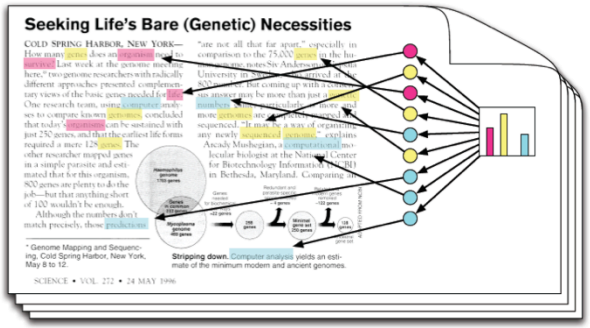


Modern ML Parallelism: Topic Models

Model (Topics)

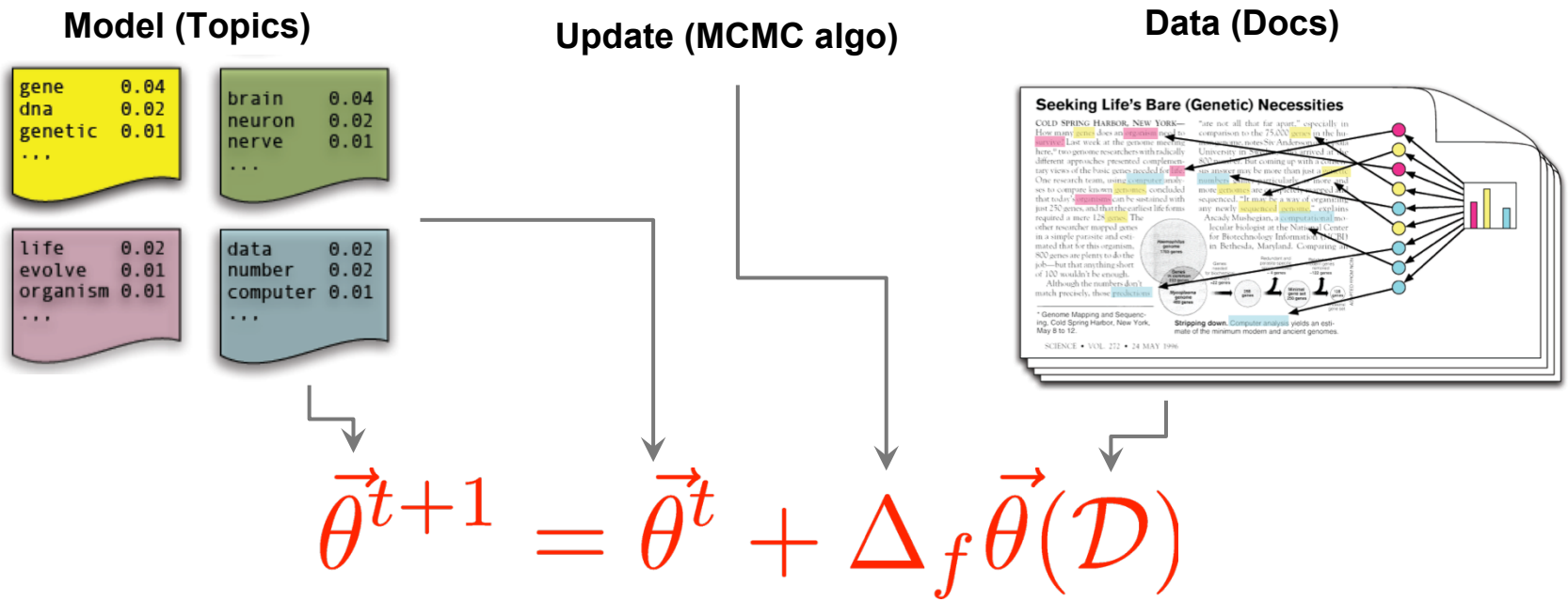


Data (Docs)



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

Modern ML Parallelism: Topic Models



Modern ML Parallelism: Topic Models

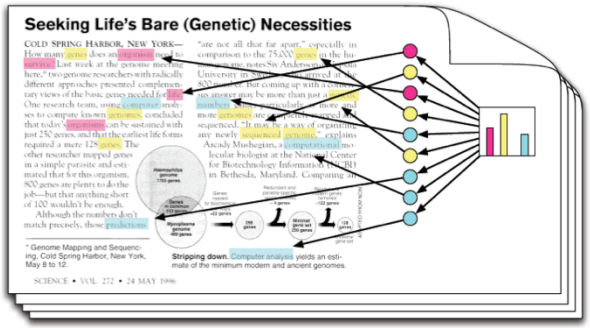
BIG DATA (billions of docs)

Data (Docs)

Model (Topics)

Update (MCMC algo)

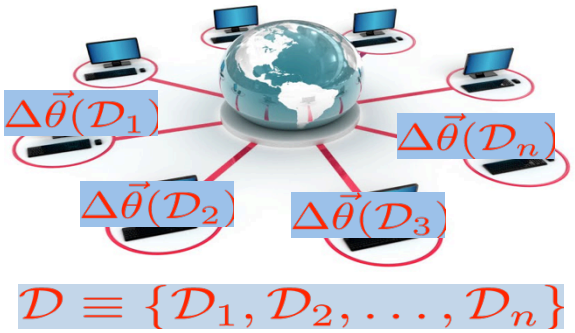
gene	0.04	brain	0.04
dna	0.02	neuron	0.02
genetic	0.01	nerve	0.01
...		...	
life	0.02	data	0.02
evolve	0.01	number	0.02
organism	0.01	computer	0.01
...		...	



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

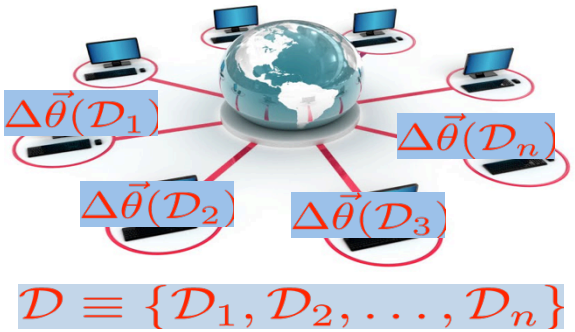
Modern ML Parallelism: Topic Models

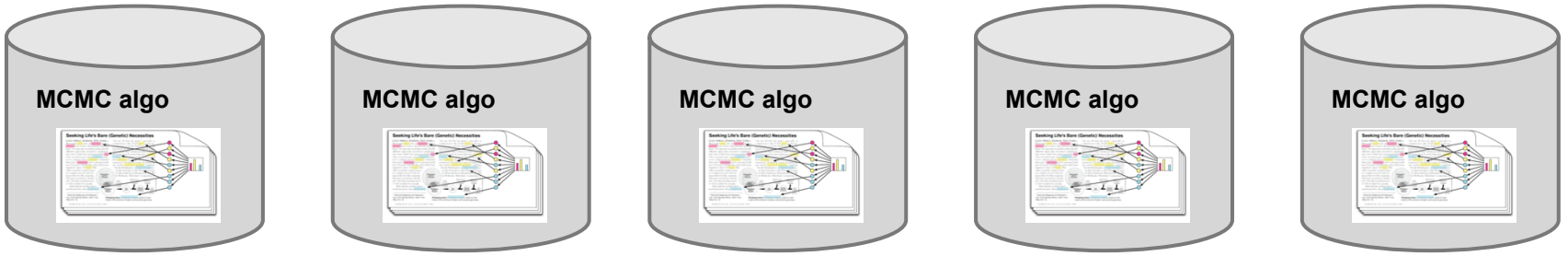
Data-parallel strategy for topic models



Modern ML Parallelism: Topic Models

Data-parallel strategy for topic models

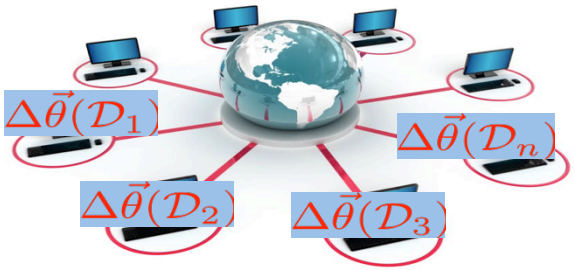




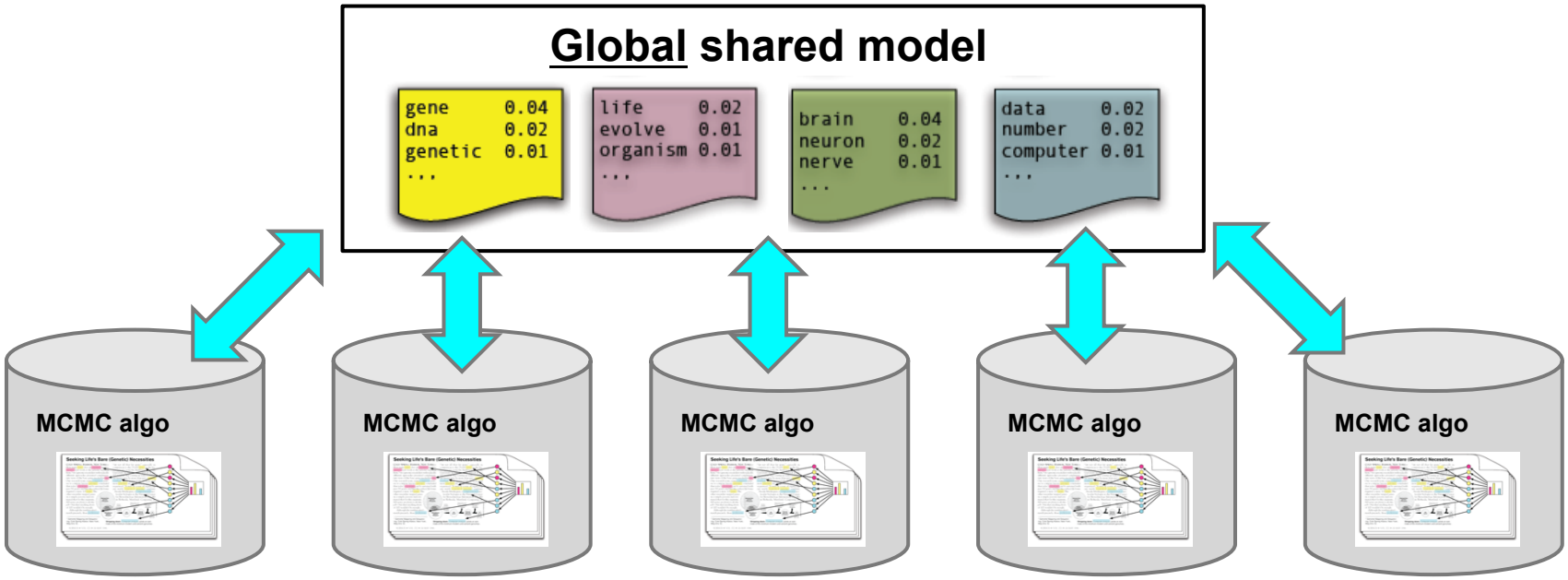
Worker machines with <u>local</u> data

Modern ML Parallelism: Topic Models

Data-parallel strategy for topic models



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

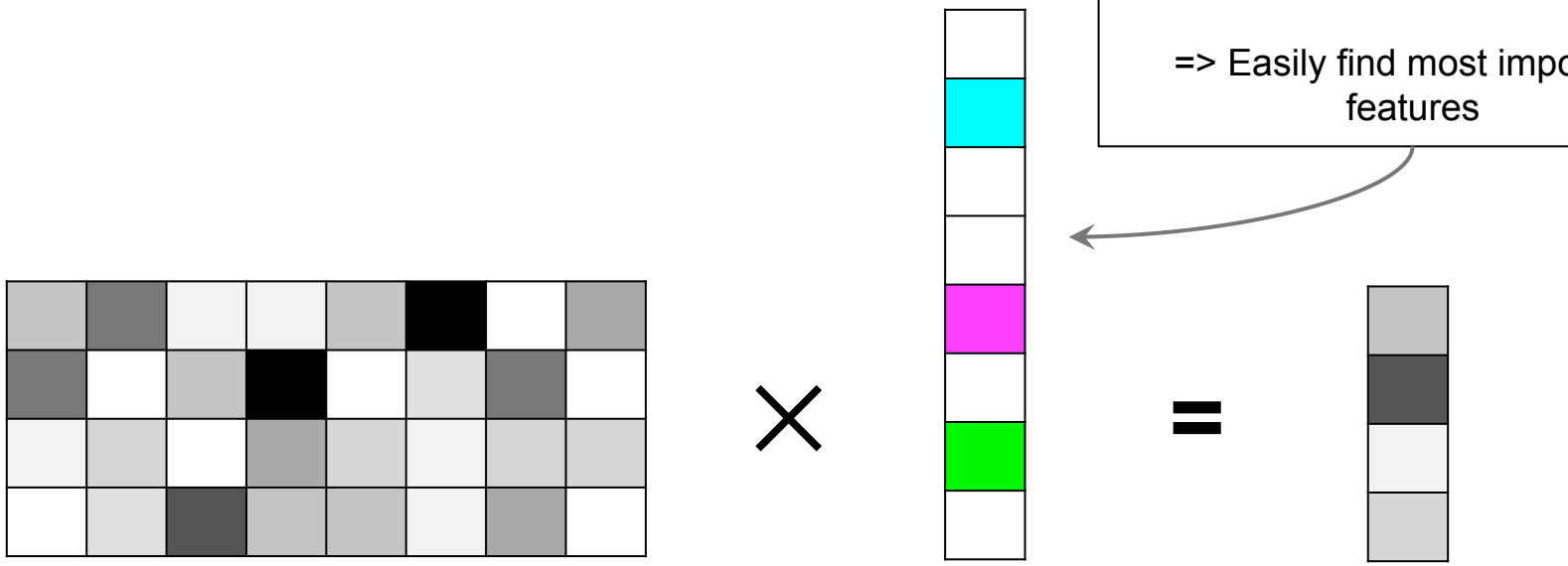


Worker machines with local data

Modern ML Parallelism: Lasso Regression

Lasso outputs sparse parameter vectors (few non-zeros)

=> Easily find most important features



Feature Matrix
(N samples by M features)
Input

Parameter Vector
(M features)
Output

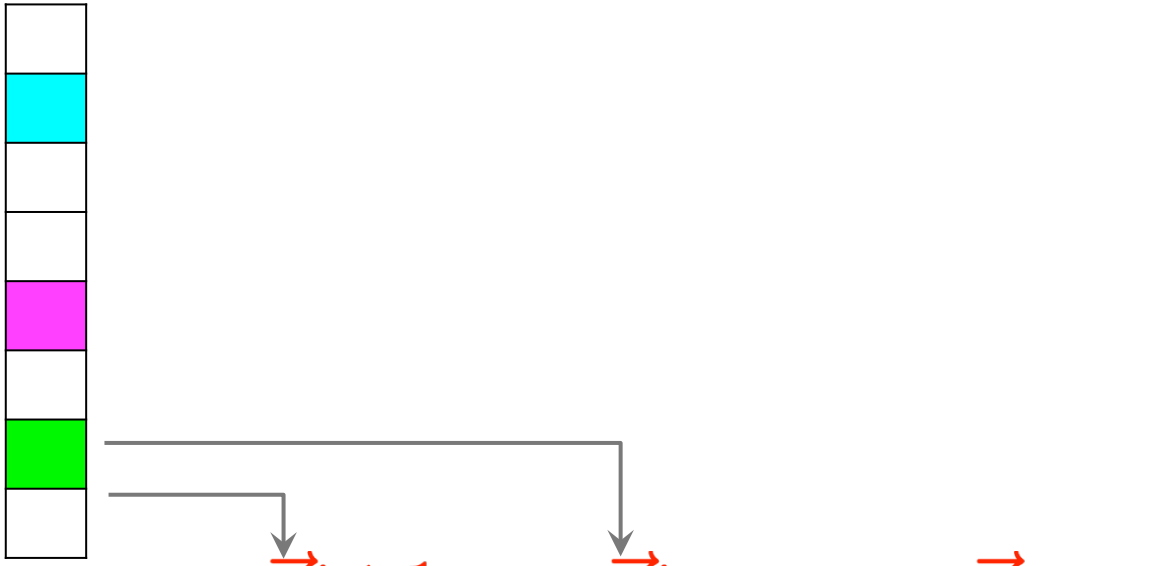
Response Matrix
(N samples)
Input

Modern ML Parallelism:
Lasso Regression

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

Modern ML Parallelism: Lasso Regression

Model (Parameter Vector)

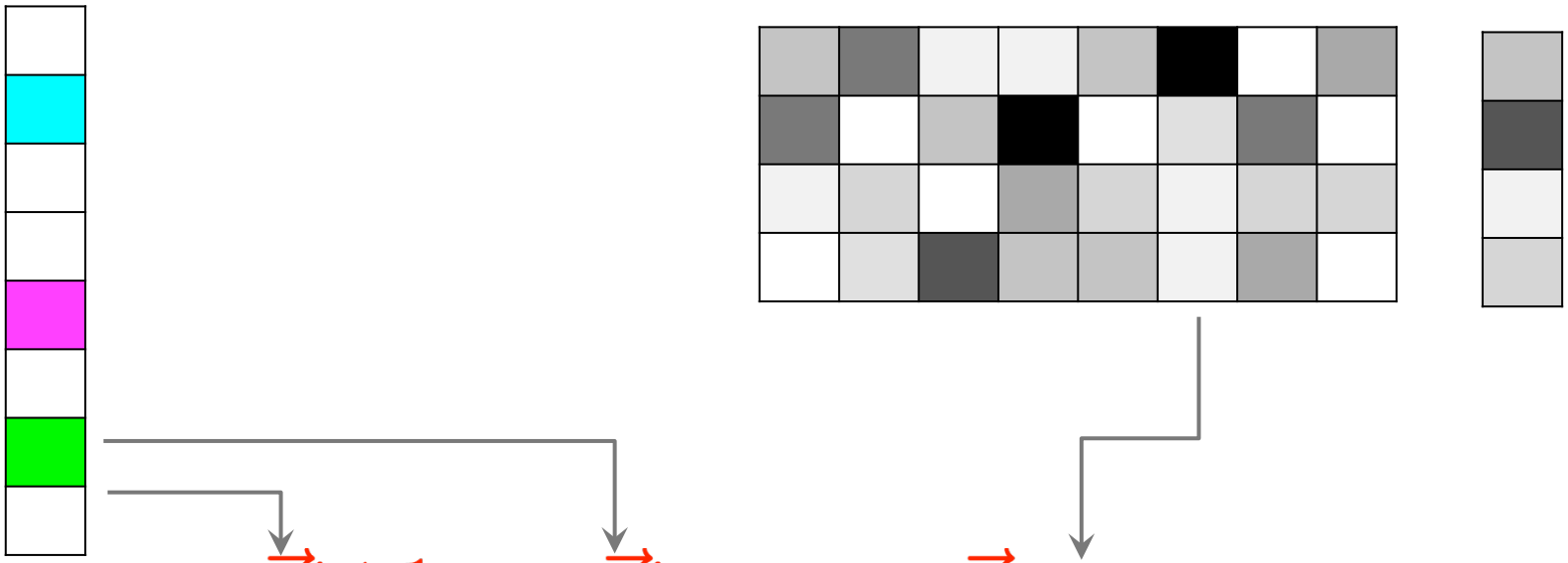


$$\vec{\theta}^{t+1} = \vec{\theta}^{t} + \Delta_f \vec{\theta}(\mathcal{D})$$

Modern ML Parallelism: Lasso Regression

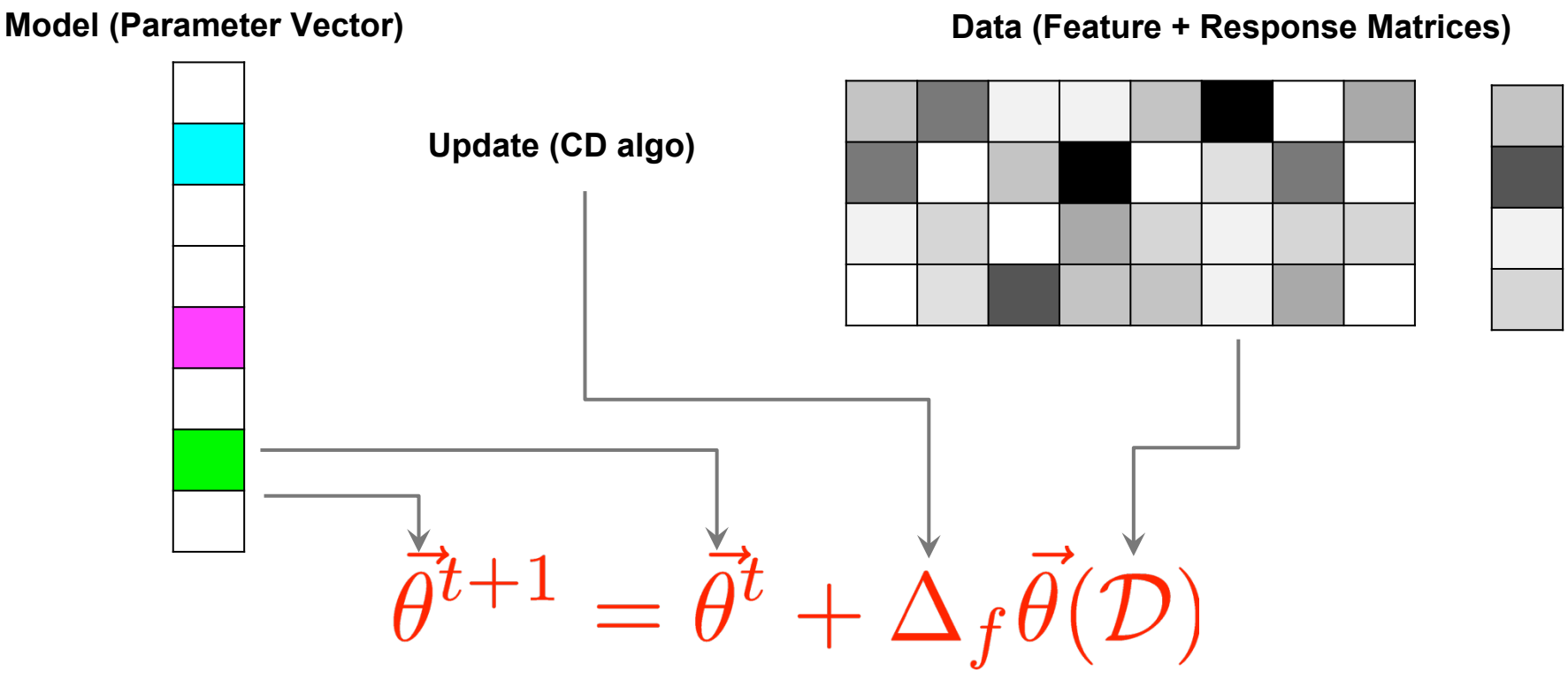
Model (Parameter Vector)

Data (Feature + Response Matrices)



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

Modern ML Parallelism: Lasso Regression

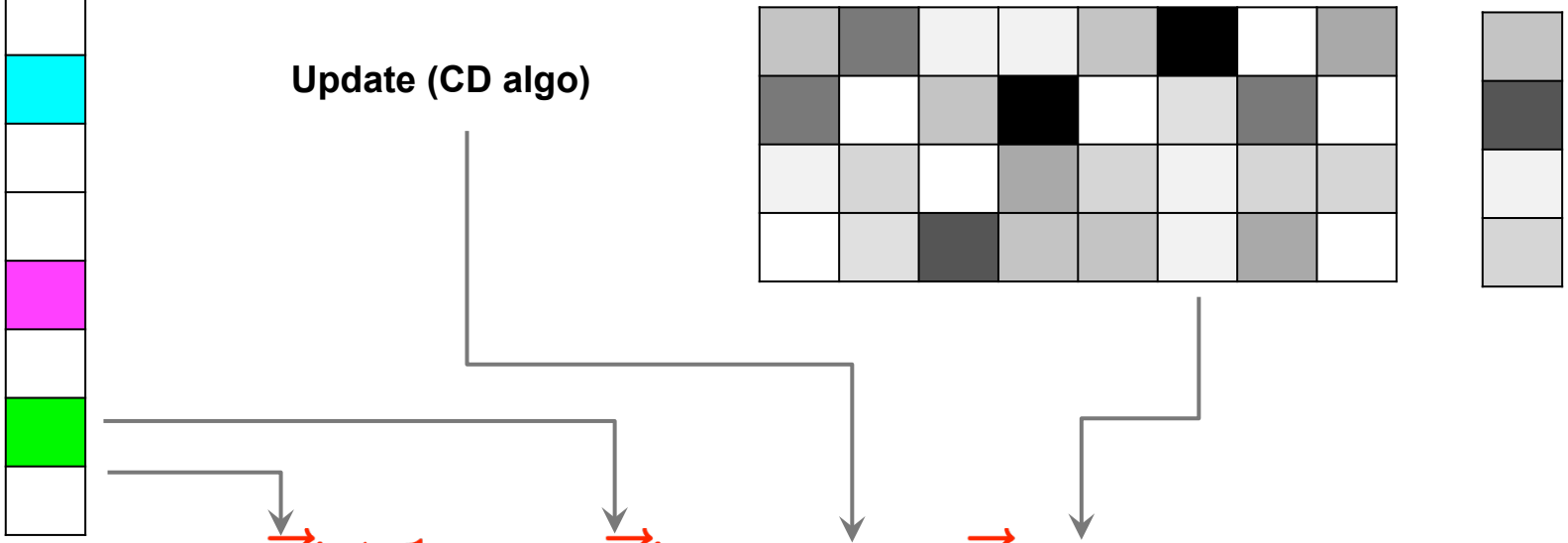


Modern ML Parallelism: Lasso Regression

BIG MODEL (100 billions of params)

Model (Parameter Vector)

Data (Feature + Response Matrices)

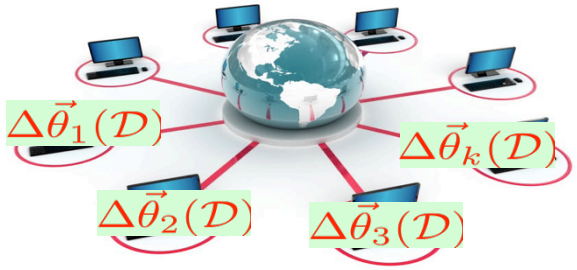


Update (CD algo)

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

Modern ML Parallelism: Lasso Regression

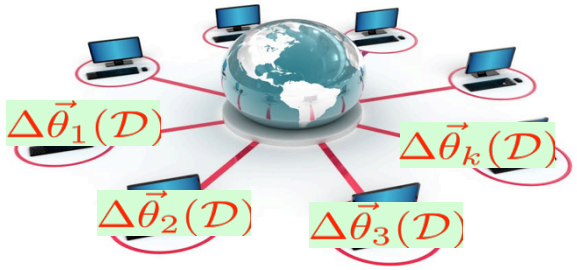
Model-parallel strategy for Lasso



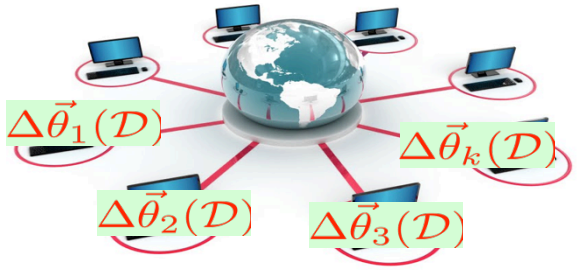
$$\vec{\theta} \equiv [\vec{\theta_1}^{\mathrm{T}}, \vec{\theta_2}^{\mathrm{T}}, \ldots, \vec{\theta_k}^{\mathrm{T}}\}^{\mathrm{T}}$$

Modern ML Parallelism: Lasso Regression

Model-parallel strategy for Lasso



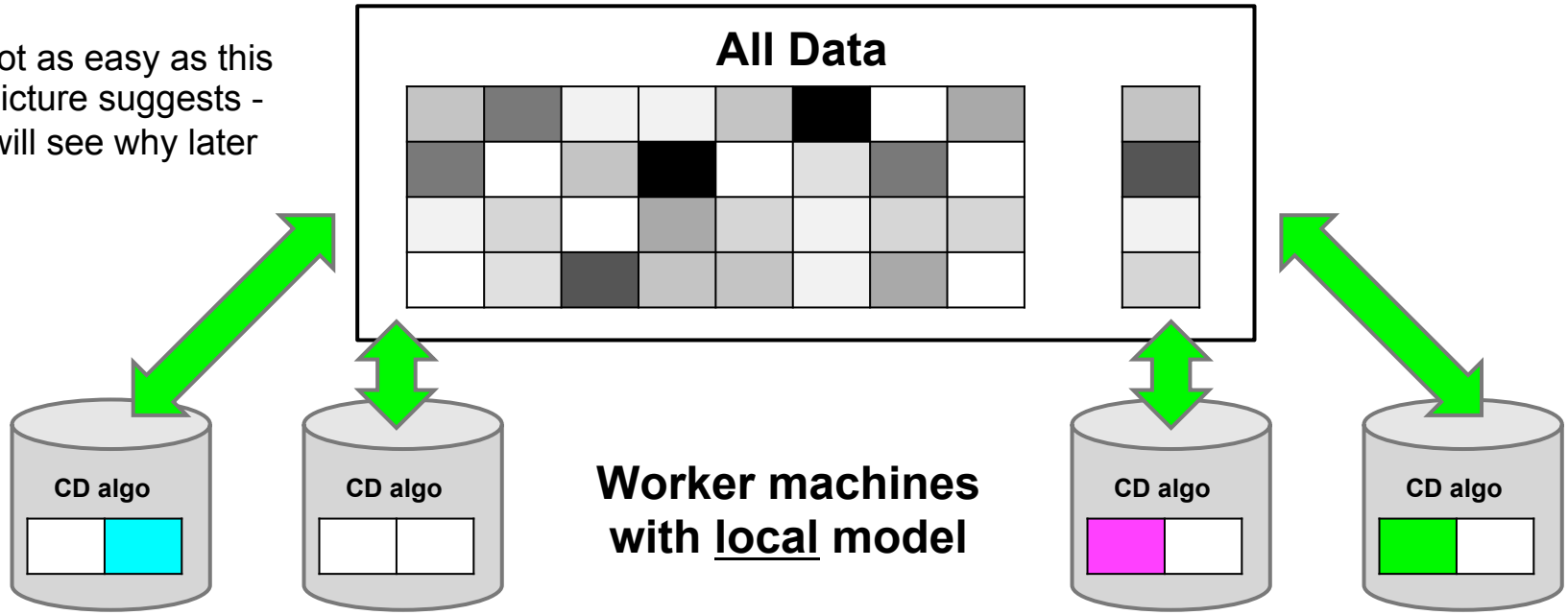
Modern ML Parallelism: Lasso Regression



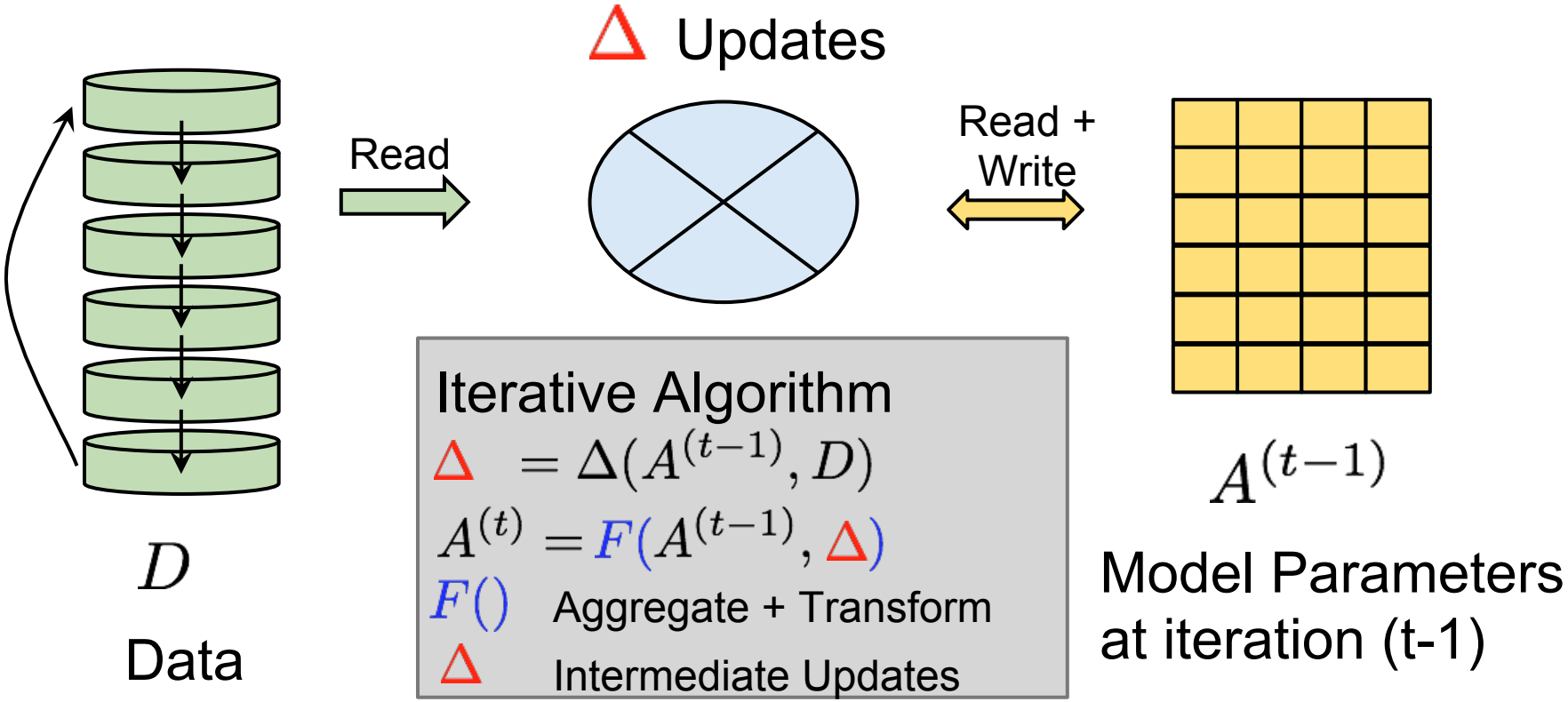
$$\vec{\theta} \equiv [\vec{\theta}_1{}^{\mathrm{T}}, \vec{\theta}_2{}^{\mathrm{T}}, \ldots, \vec{\theta}_k{}^{\mathrm{T}}\}^{\mathrm{T}}$$

Model-parallel strategy for Lasso

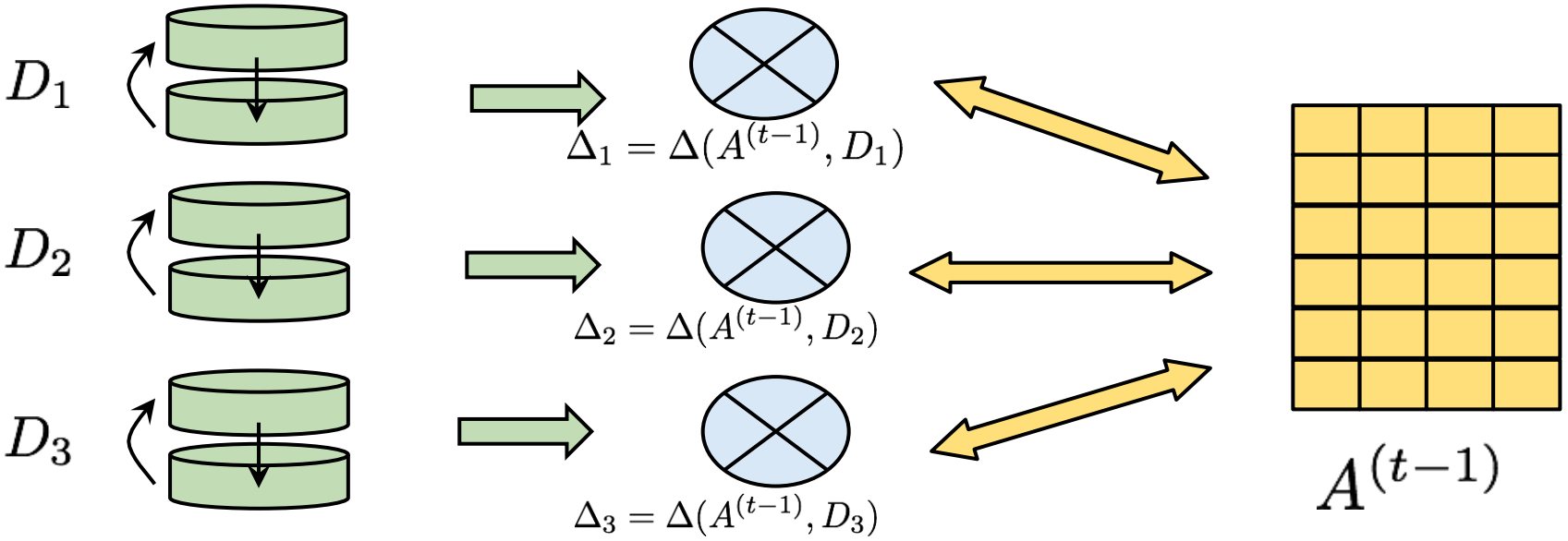
Not as easy as this picture suggests - will see why later



A General Picture of ML Iterative Algos



Data Parallelism

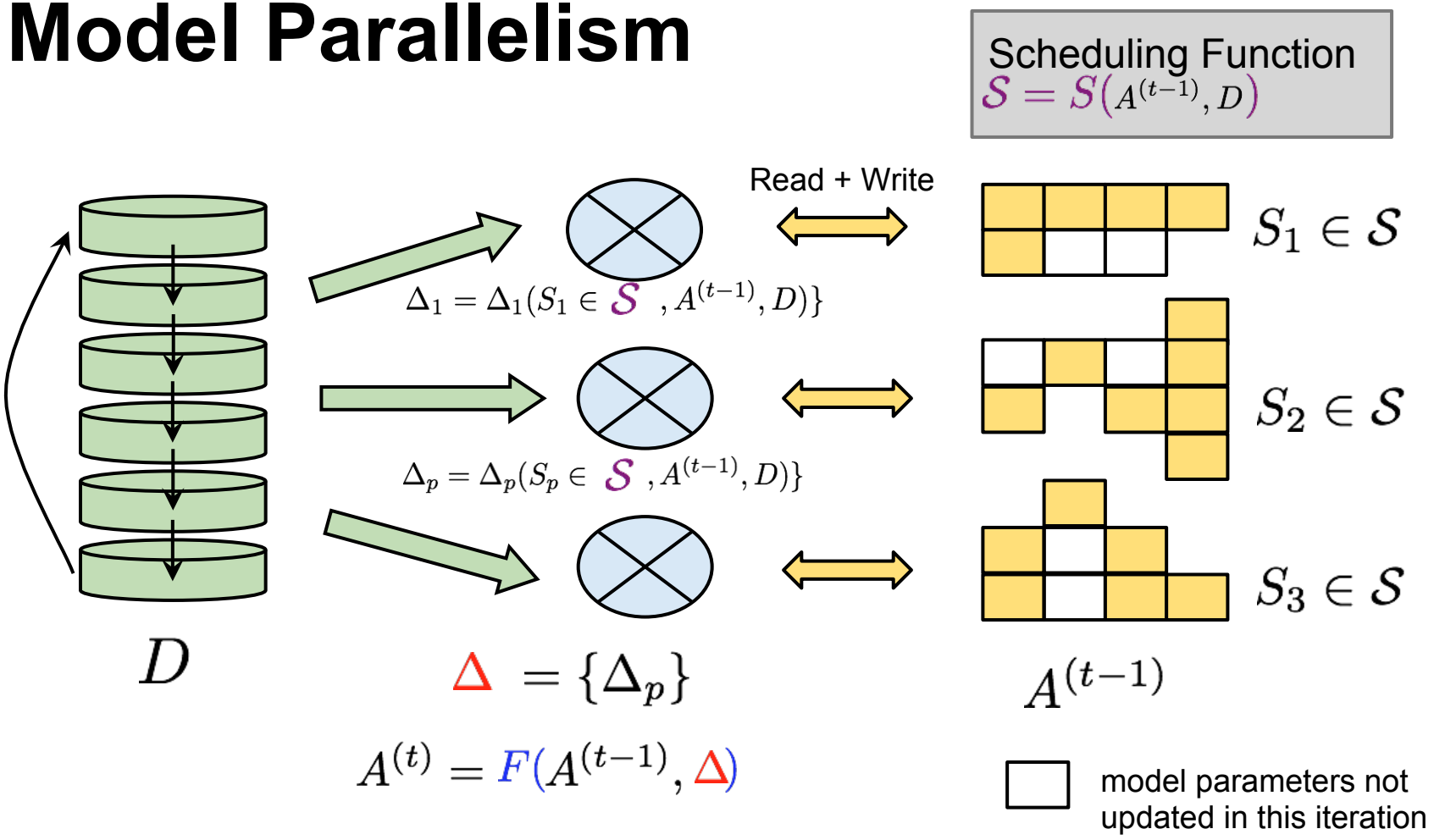


Additive Updates

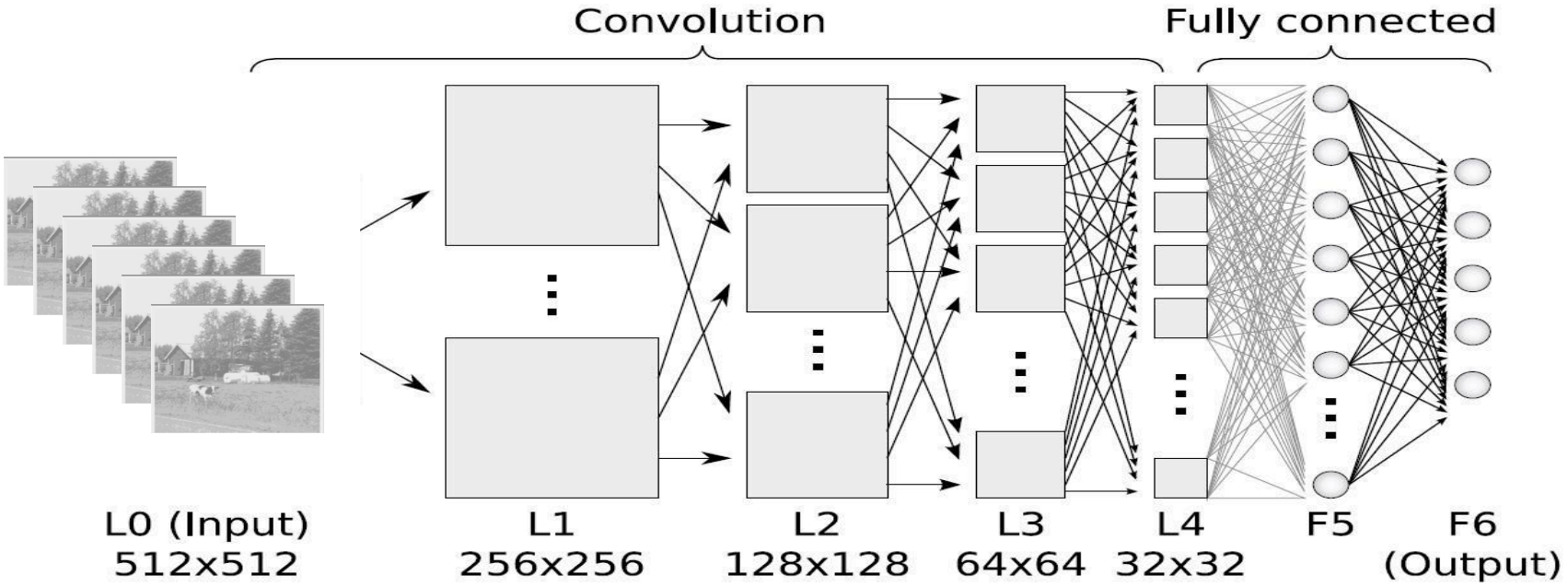
$$\Delta = \sum_{p=1}^{3} \Delta_p$$

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

Model Parallelism



Modern ML Parallelism: Deep Neural Networks

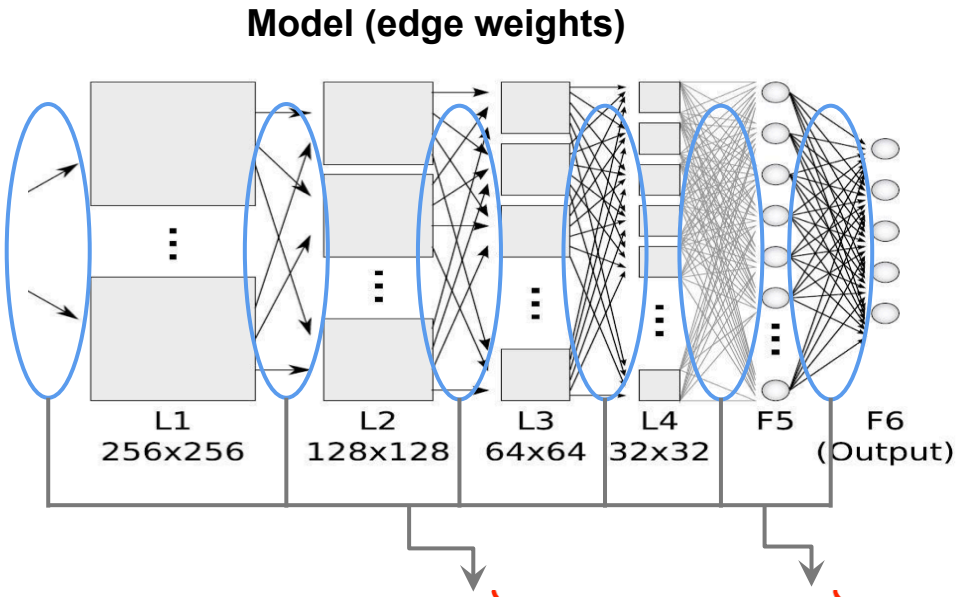


Source: University of Bonn

Modern ML Parallelism: Deep Neural Networks

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

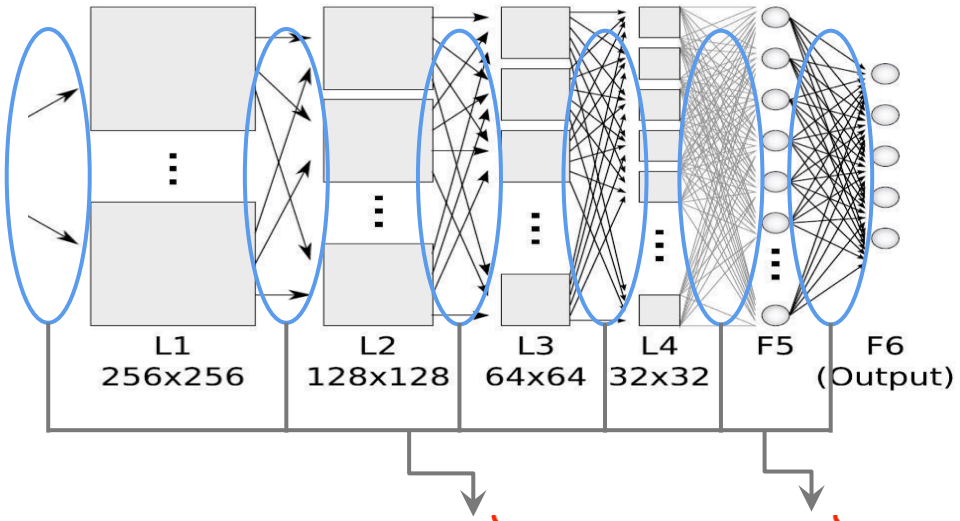
Modern ML Parallelism: Deep Neural Networks



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

Modern ML Parallelism: Deep Neural Networks

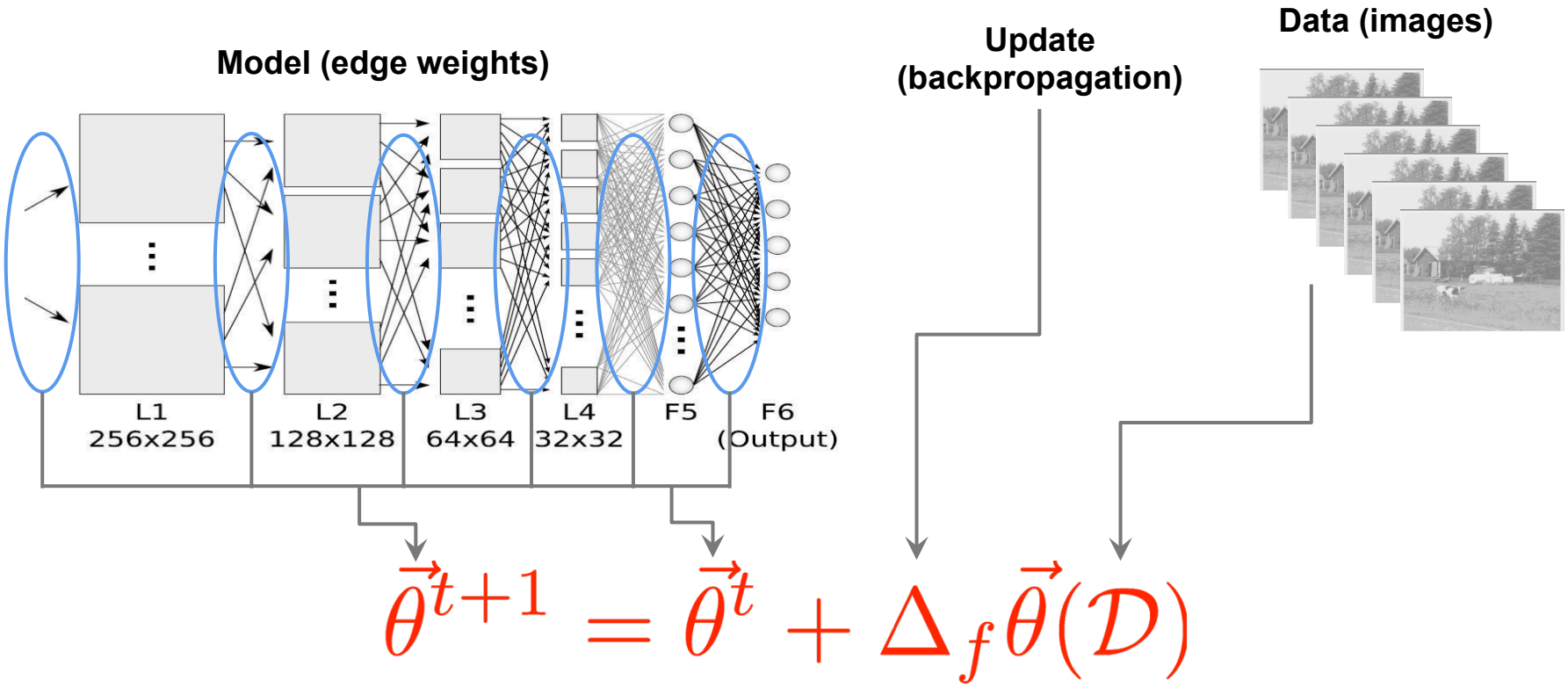
Model (edge weights)



Data (images)

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

Modern ML Parallelism: Deep Neural Networks



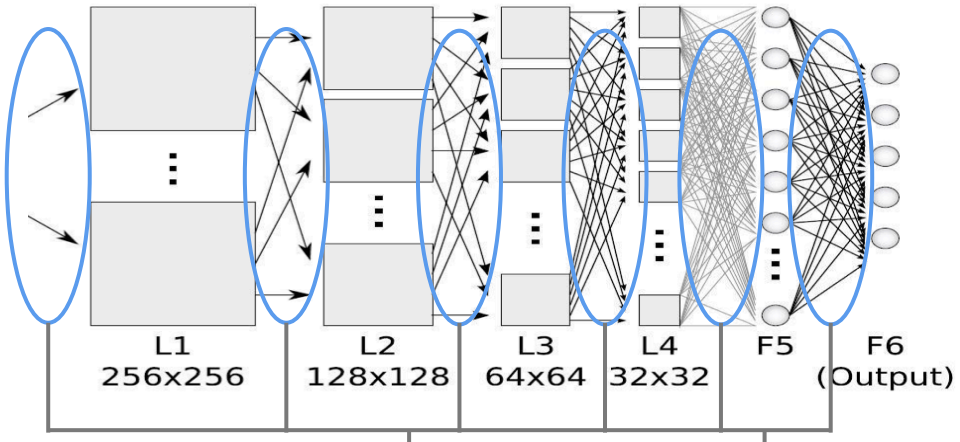
Modern ML Parallelism: Deep Neural Networks

Data and Model can both be big!
Millions of images, Billions of weights
What to do?

Model (edge weights)

Update (backpropagation)

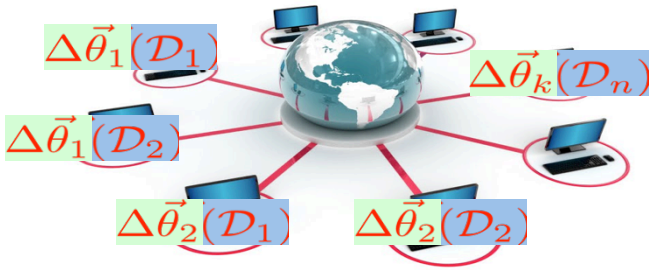
Data (images)



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

Modern ML Parallelism: Deep Neural Networks

Data-and-Model-parallel strategy for DNN

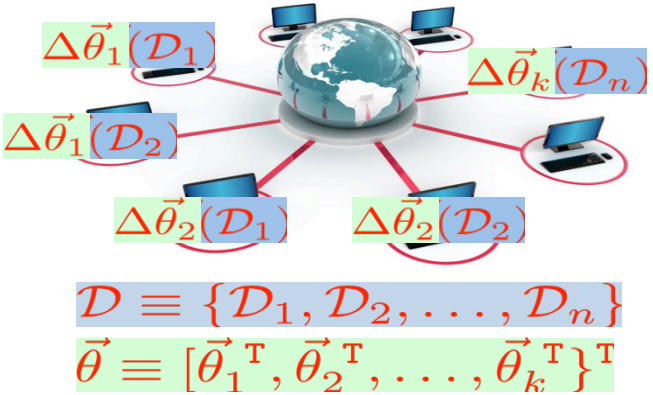


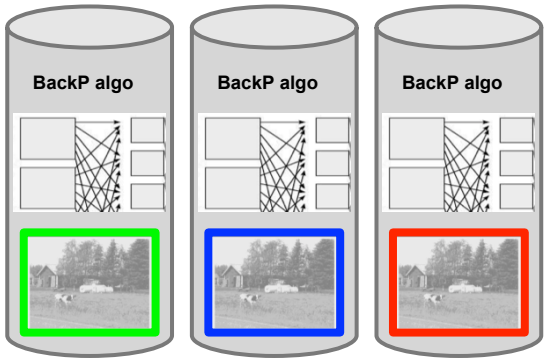
$$\mathcal{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$$

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

Modern ML Parallelism: Deep Neural Networks

Data-and-Model-parallel strategy for DNN

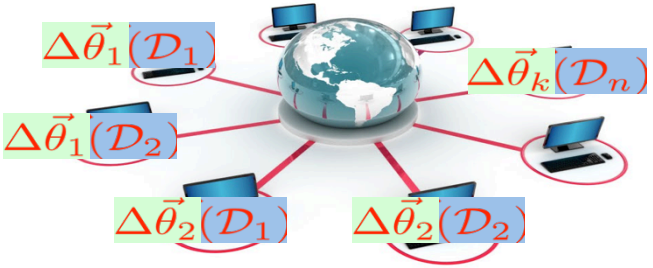




Modern ML Parallelism: Deep Neural Networks

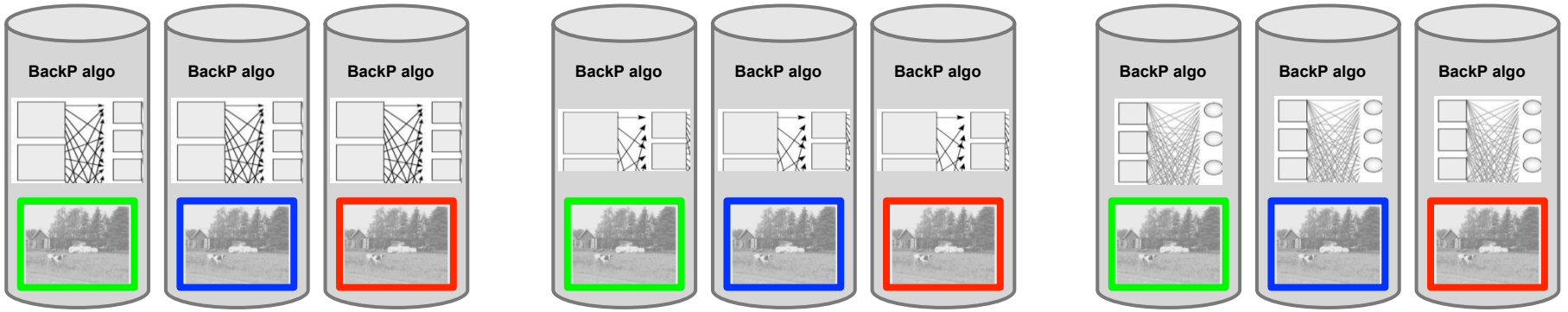
Data-and-Model-parallel strategy for DNN

"All-pairs" of data and model chunks



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

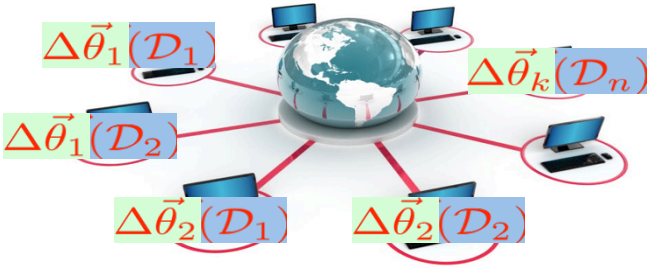
$$\vec{\theta} \equiv [\vec{\theta}_1{}^{\mathrm{T}}, \vec{\theta}_2{}^{\mathrm{T}}, \ldots, \vec{\theta}_k{}^{\mathrm{T}}\}^{\mathrm{T}}$$



Modern ML Parallelism: Deep Neural Networks

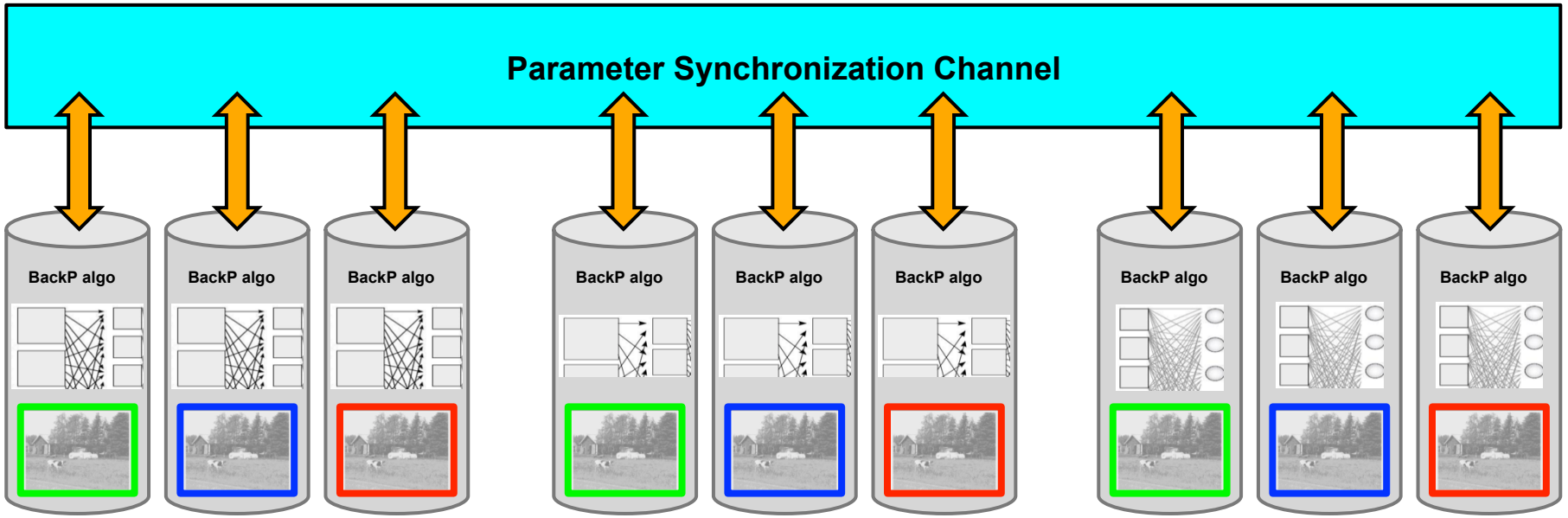
Data-and-Model-parallel strategy for DNN

"All-pairs" of data and model chunks



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

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



Is data/model-parallelism that easy?

- Not always - certain 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 - will see why later!
  - NOT equivalent to serial execution of ML algo
- What about software to write data/model-parallel ML easily, quickly?

Modern Systems for Big ML

- Just now: basic ideas of data-, model-parallelism in ML
- What systems allow ML programs to be written, executed this way?

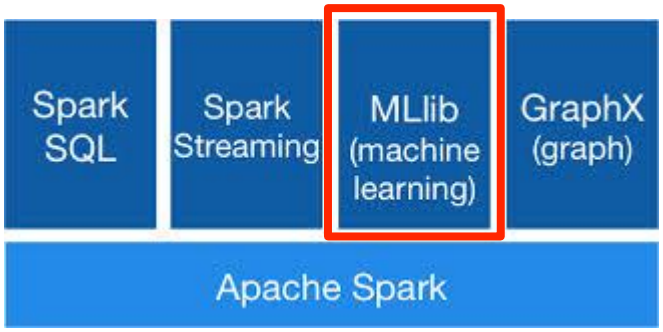
Modern Systems for Big ML

- Just now: basic ideas of data-, model-parallelism in ML
- What systems allow ML programs to be written, executed this way?

Spark Overview

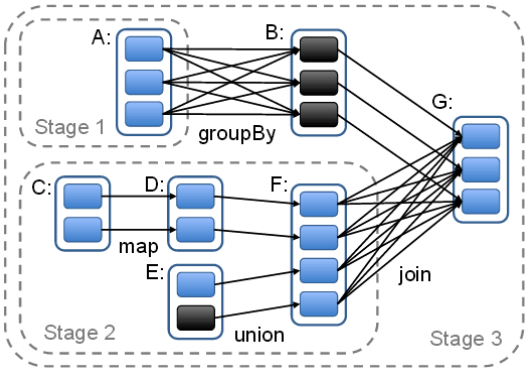
- General-purpose system for Big Data processing
  - Shell/interpreter for Matlab/R-like analytics

- MLlib = Spark's ready-to-run ML library
  - Implemented on Spark's API



Spark Overview

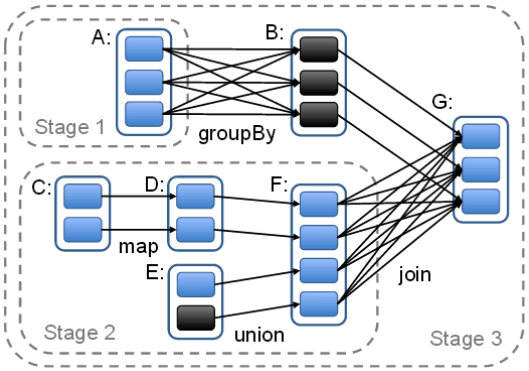
- Key feature: Resilient Distributed Datasets (RDDs)
  - Data processing = lineage graph of transforms
  - RDDs = nodes
  - Transforms = edges



Source: Zaharia et al. (2012)

Spark Overview

- 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 Demo in Linux VM

Logistic Regression (Spark Shell)

```
# Start Spark Shell
cd ~/spark-1.0.2/
bin/spark-shell

// Scala code starts here
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils

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

Spark Demo in Linux VM

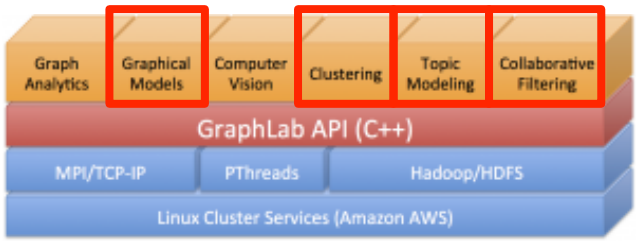
Logistic Regression (Spark Shell)

```
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// Clear the default threshold.
model.clearThreshold()

// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}
// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()

println("Area under ROC = " + auROC)

// More info @ https://spark.apache.org/docs/latest/mllib-linear-methods.html
```

GraphLab Overview

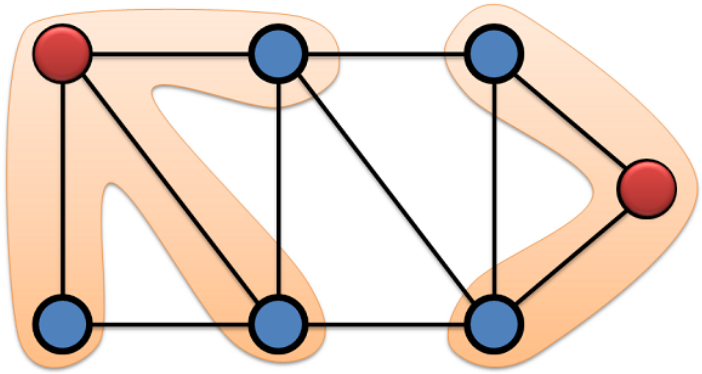
- System for Graph Programming
  - Think of ML algos as graph algos

- Comes with ready-to-run “toolkits”
  - ML-centric toolkits: clustering, collaborative filtering, topic modeling, graphical models



GraphLab Overview

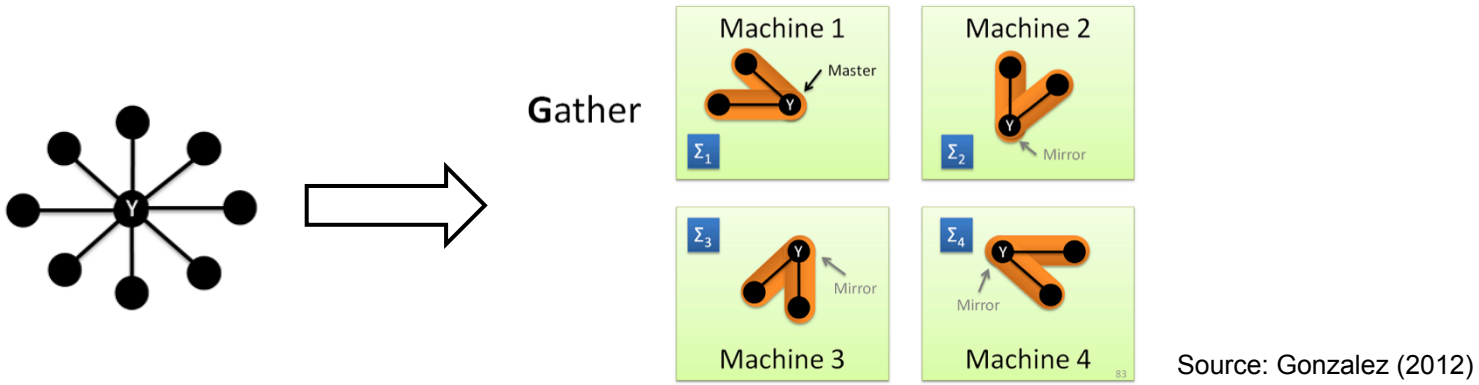
- Key feature: Gather-Apply-Scatter API
  - Write ML algos as vertex programs
  - Run vertex programs in parallel on each graph node
  - Graph nodes, edges can have data, parameters



Source: Gonzalez (2012)

GraphLab Overview

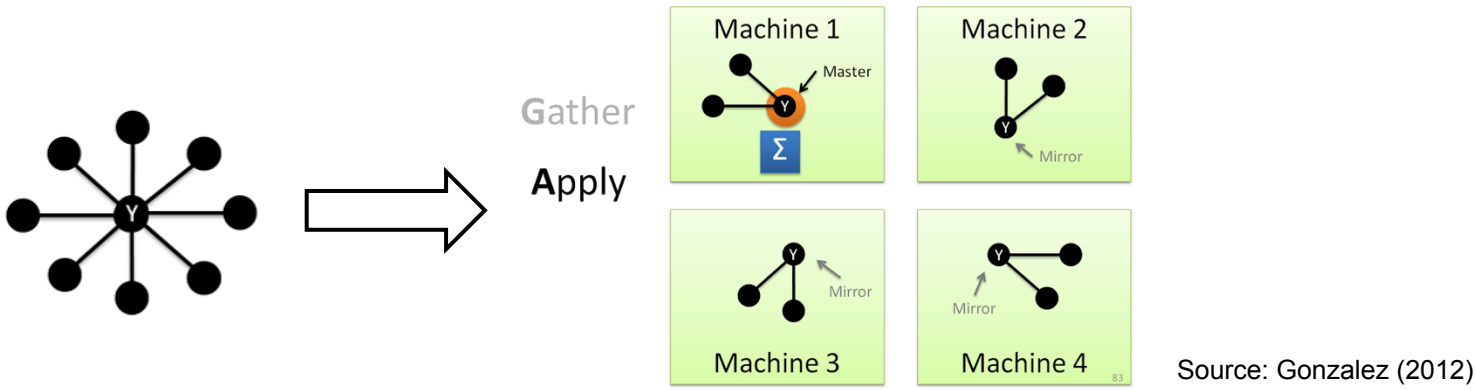
- GAS Vertex Programs:
  - **1) Gather():** Accumulate data, params from my neighbors + edges
  - 2) Apply(): Transform output of Gather(), write to myself
  - 3) Scatter(): Transform output of Gather(), Apply(), write to my edges



Source: Gonzalez (2012)

GraphLab Overview

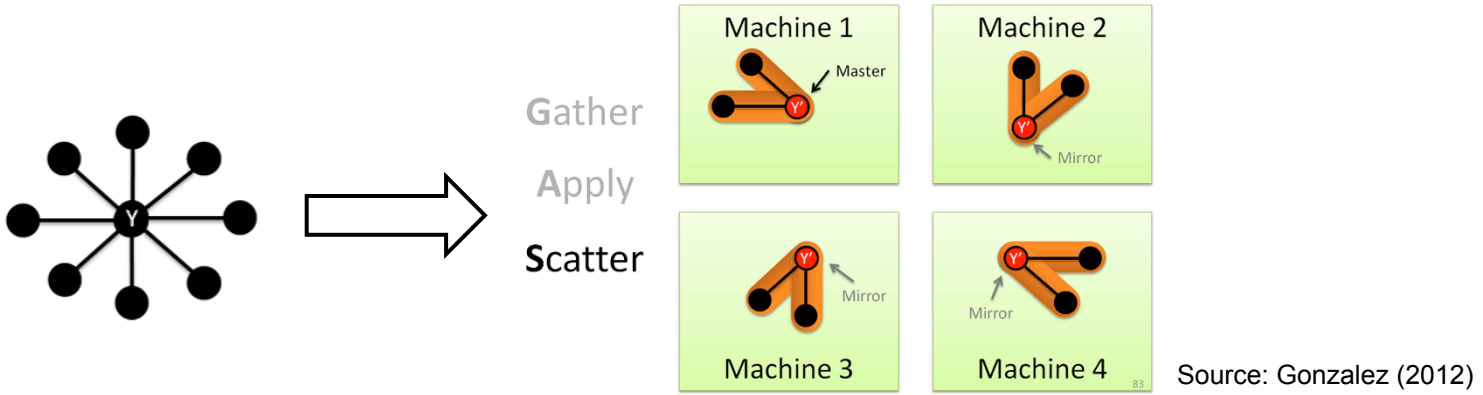
- GAS Vertex Programs:
  - 1) Gather(): Accumulate data, params from my neighbors + edges
  - **2) Apply():** Transform output of Gather(), write to myself
  - 3) Scatter(): Transform output of Gather(), Apply(), write to my edges



Source: Gonzalez (2012)

GraphLab Overview

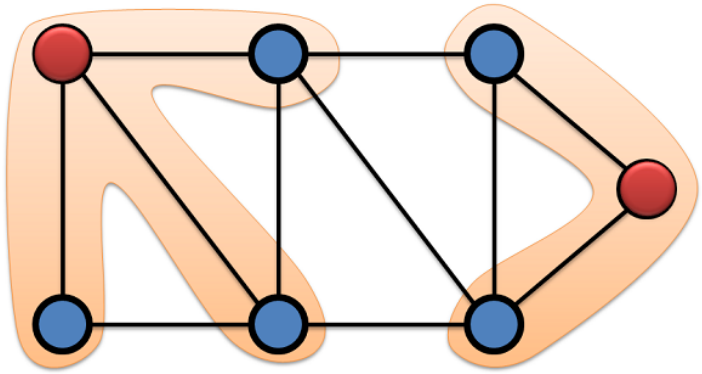
- GAS Vertex Programs:
  - 1) Gather(): Accumulate data, params from my neighbors + edges
  - 2) Apply(): Transform output of Gather(), write to myself
  - **3) Scatter():** Transform output of Gather(), Apply(), write to my edges



Source: Gonzalez (2012)

GraphLab Overview

- 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 Demo in Linux VM

Topic Modeling (Linux shell)

```
cd ~/graphlab-master/release/toolkits/topic_modeling

# Run Topic Model on sample dataset, continuous output to screen
./cgs_lda --corpus ./daily_kos/tokens --dictionary ./daily_kos/dictionary.txt --ncpus=4

# Run Topic Model on sample dataset, save output to disk
./cgs_lda --corpus ./daily_kos/tokens --dictionary ./daily_kos/dictionary.txt --ncpus=4 \
   --word_dir word_counts --doc_dir doc_counts --burnin=60

# More info @ http://docs.graphlab.org/topic_modeling.html
```

Petuum Overview

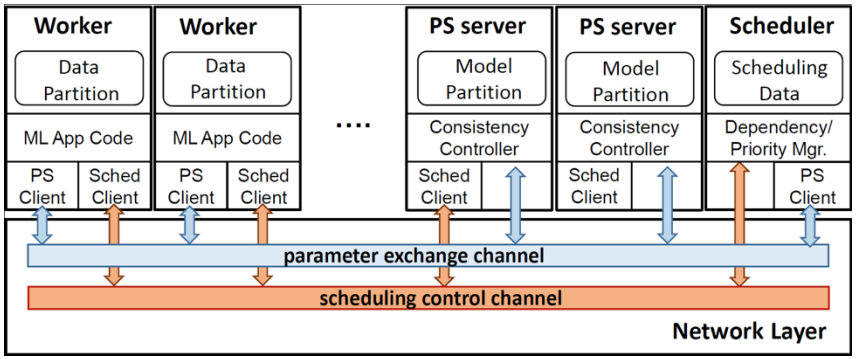
- System for iterative-convergent ML algos
  - Speeds up ML via data-, model-parallel insights

- Ready-to-run ML programs
  - Now: Topic Model, DNN, Lasso & Logistic Regression, MF
  - Soon: Tree ensembles, Metric Learning, Network Models, CNN, more

Petuum Overview

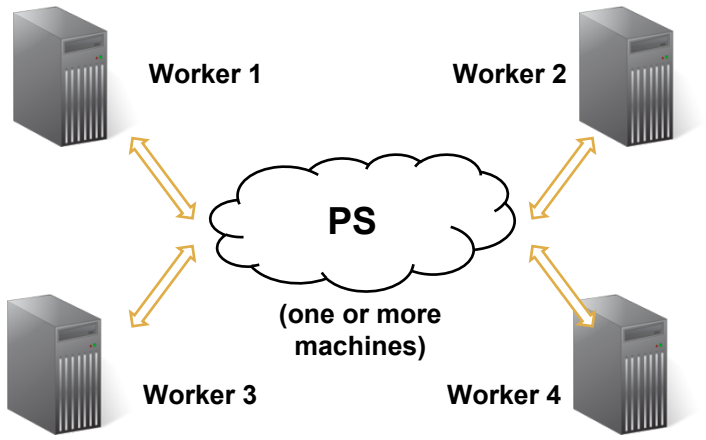
- Key modules
  - Parameter Server for data-parallel ML algos
  - Scheduler for model-parallel ML algos

- “Think like an ML algo”
  - ML algo = (1) update equations + (2) run those eqns in some order



Petuum Overview

- Parameter Server
  - Enables data-parallelism: model parameters become global
  - Special type of Distributed Shared Memory (DSM)



Single Machine

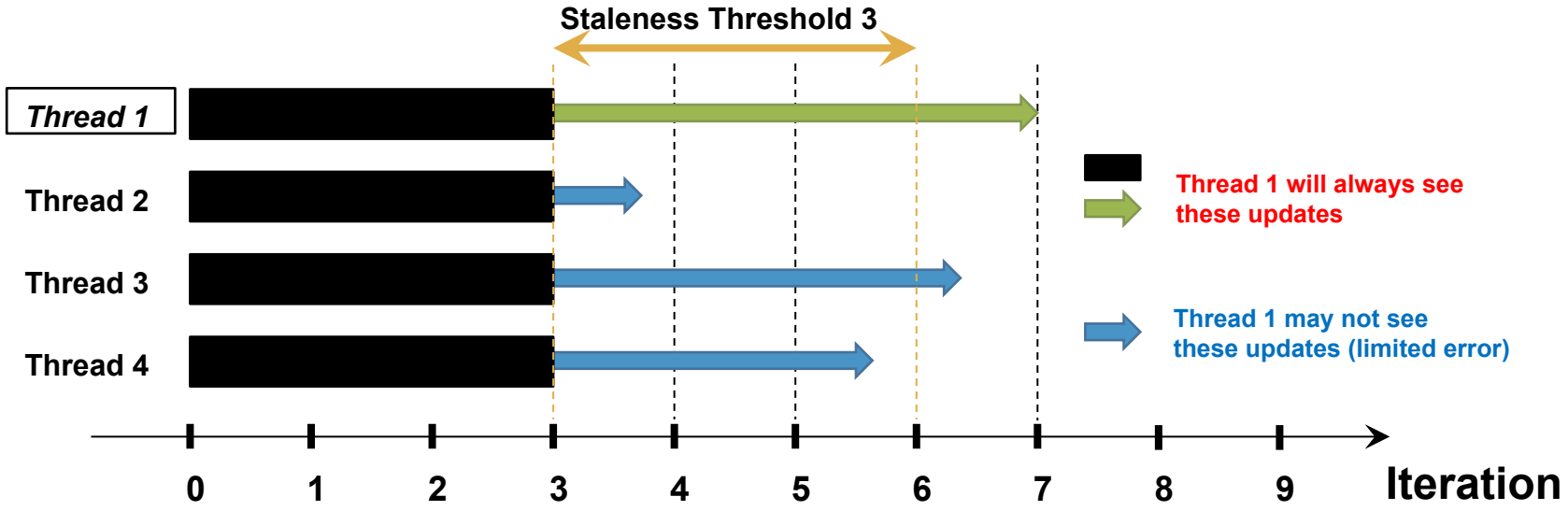
```
ProcessDataPoint(i) {
  for j = 1 to M {
    old = model[j]
    delta = f(model,data(i))
    model[j] += delta
  }
}
```

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)
  }
}
```

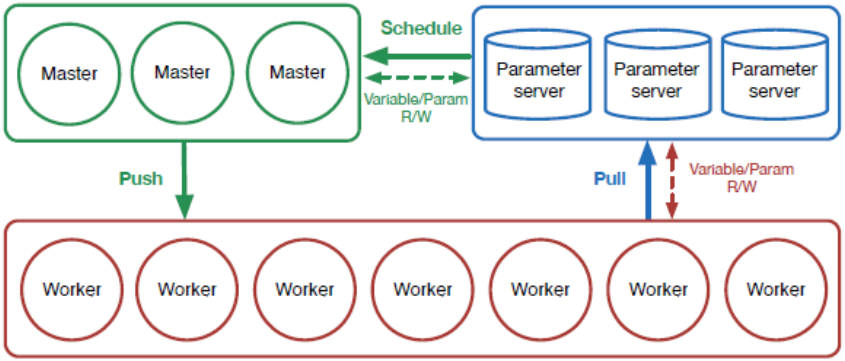
Petuum Overview

- Parameter Server benefits:
  - ML-tailored consistency model: Stale Synchronous Parallel (SSP)
  - Asynchronous-like speed, BSP-like ML correctness guarantees



Petuum Overview

- Scheduler
  - Enables correct model-parallelism
  - Can analyze ML model structure for best execution order



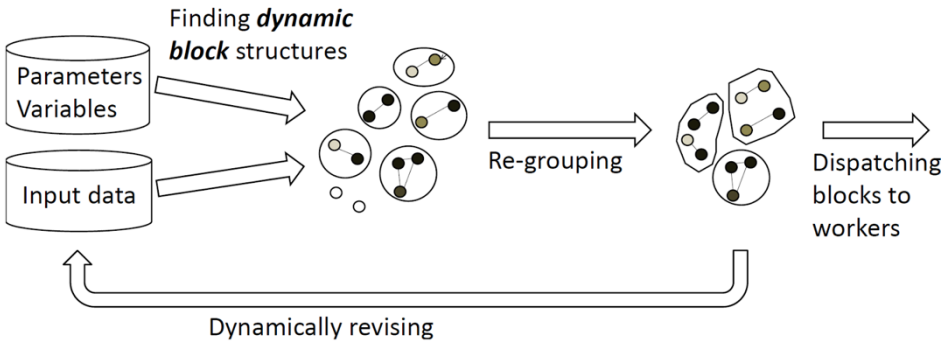

```
schedule() {
  // Select U vars x[j] to be sent
  // to the workers for updating
  ...
  return (x[j_1], ..., x[j_U])
}
```

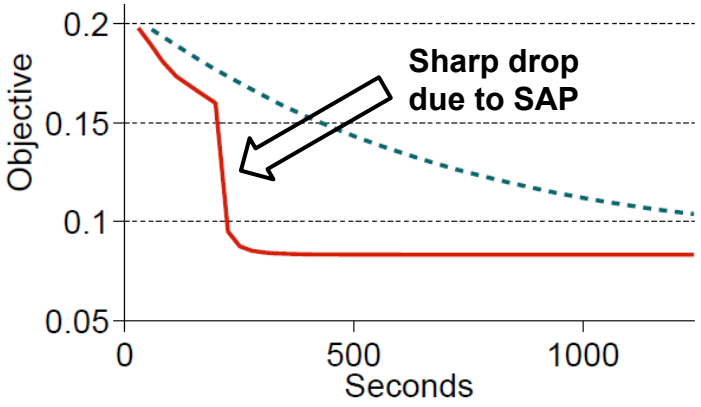
```
push(worker = p, vars = (x[j_1],...,x[j_U])) {
  // Compute partial update z for U vars x[j]
  // at worker p
  ...
  return z
}
```

```
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.
  ...
}
```

Petuum Overview

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





Petuum Demo in Linux VM

Deep Neural Network (Linux shell)

```
cd ~/petuum-release_0.93/apps/dnn

# Generate N=10,000 simulated dataset (args: #samples, #features, #classes, #partitions)
scripts/gen_data.sh 10000 440 1993 3 datasets

# Run 6-layer, 2M-param DNN on simulated dataset (args: #threads_per_machine, staleness)
scripts/run_dnn.sh 4 5 machinefiles/localserver datasets/para_imnet.txt \
  datasets/data_ptt_file.txt weights.txt biases.txt

# More info @ https://github.com/petuum/public/wiki/ML-App:-Deep-Neural-Network
```

Science of BigML: Principles, design, theory

- Just saw Spark, GraphLab, Petuum in action

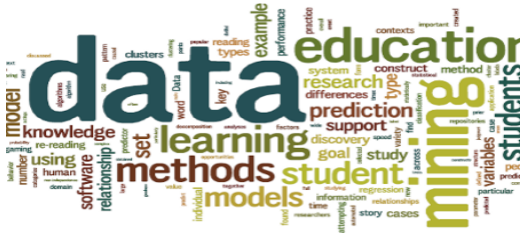
- Each has distinct technical innovations
  - How suited are they to Big ML problems?
  - How do they enable Data, Model-Parallel execution?

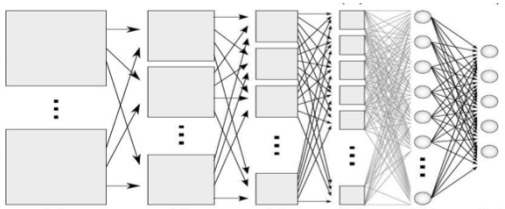
- Key insight: ML algos have special properties
  - Error-tolerance, dependency structures, uneven convergence
  - How to harness for faster data/model-parallelism?

Refresher: data, model-parallel

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

New Model = Old Model + Update(Data)

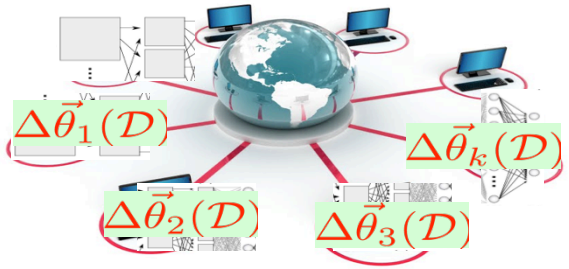




Data Parallel

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

Model Parallel

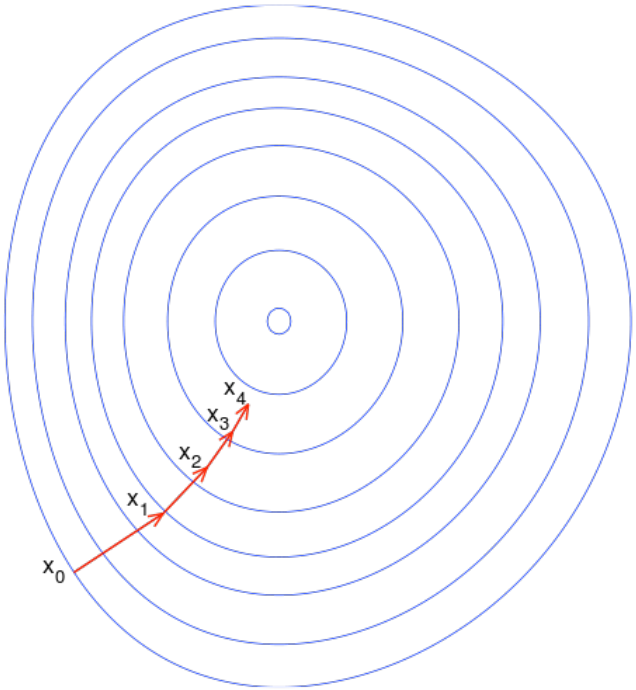


$$\vec{\theta} \equiv [\vec{\theta_1}^{\mathrm{T}}, \vec{\theta_2}^{\mathrm{T}}, \ldots, \vec{\theta_k}^{\mathrm{T}}\}^{\mathrm{T}}$$

ML algos are iterative-convergent

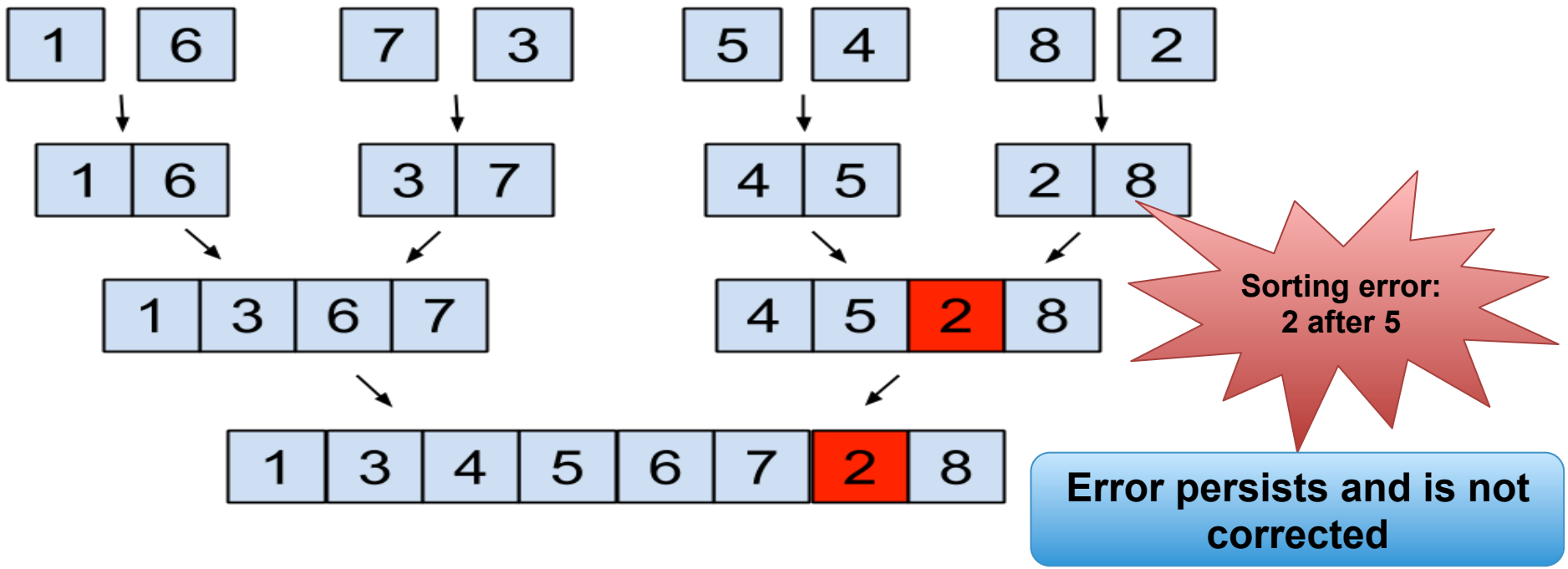
- **“Hill-climbing”**
  - Repeat update function until no change
  - True for sequential, as well as data/model-parallel ML algos

- **Why are ML algos I-C?**
  - Vast majority of ML algos are optimization or MCMC-based (and both are I-C procedures)

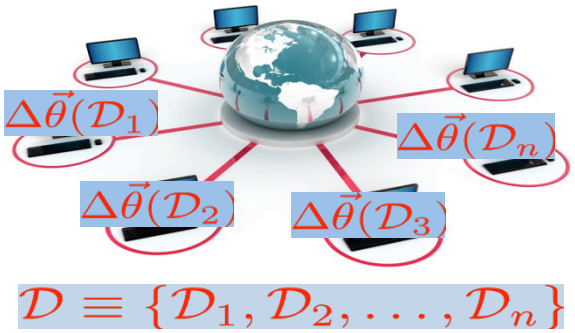


Contrast: Non-iterative-convergent

Example: Merge sort



Why not Hadoop?



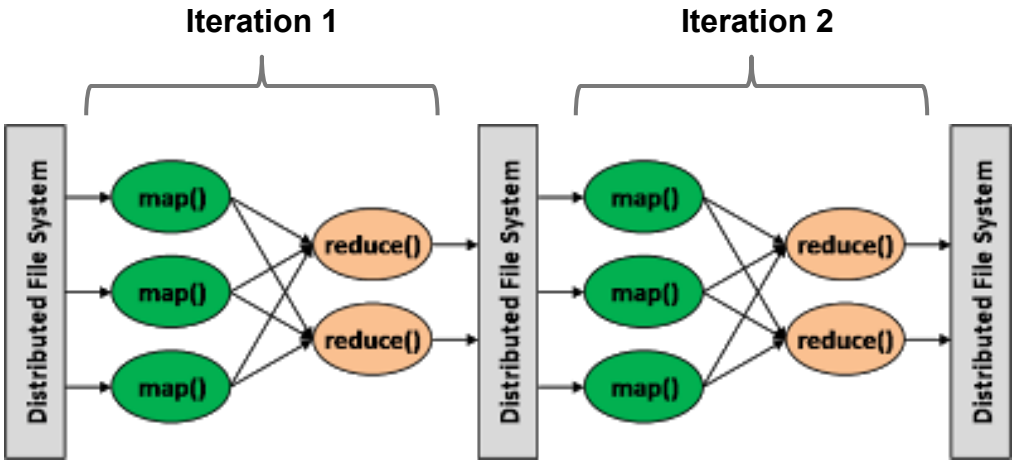


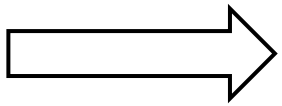
Image source: dzone.com

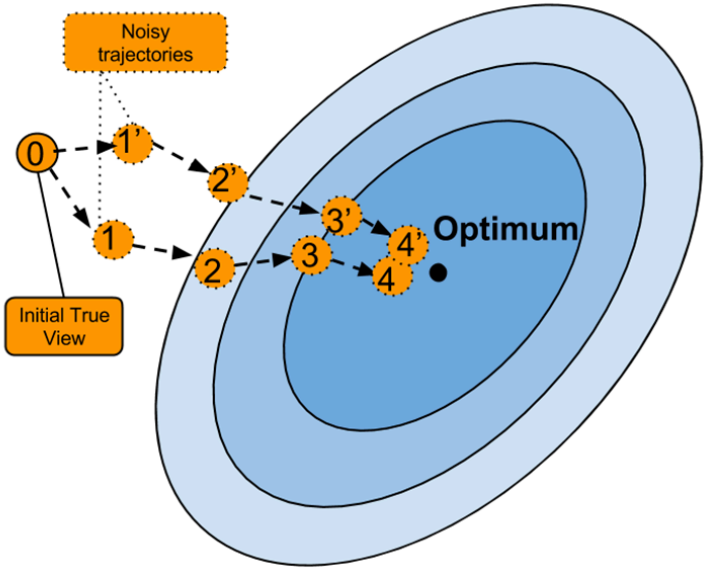
- Hadoop can execute iterative-convergent, data-parallel ML...
  - map() to distribute data samples i, compute update $\Delta(D_i)$
  - reduce() to combine updates $\Delta(D_i)$
  - Iterative ML algo = repeat map()+reduce() again and again
- But reduce() writes to HDFS before starting next iteration's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

Properties of I-C ML algos

- (1) “Self-healing” and error-tolerant
  - Model parameters a bit wrong => won’t affect final outcome

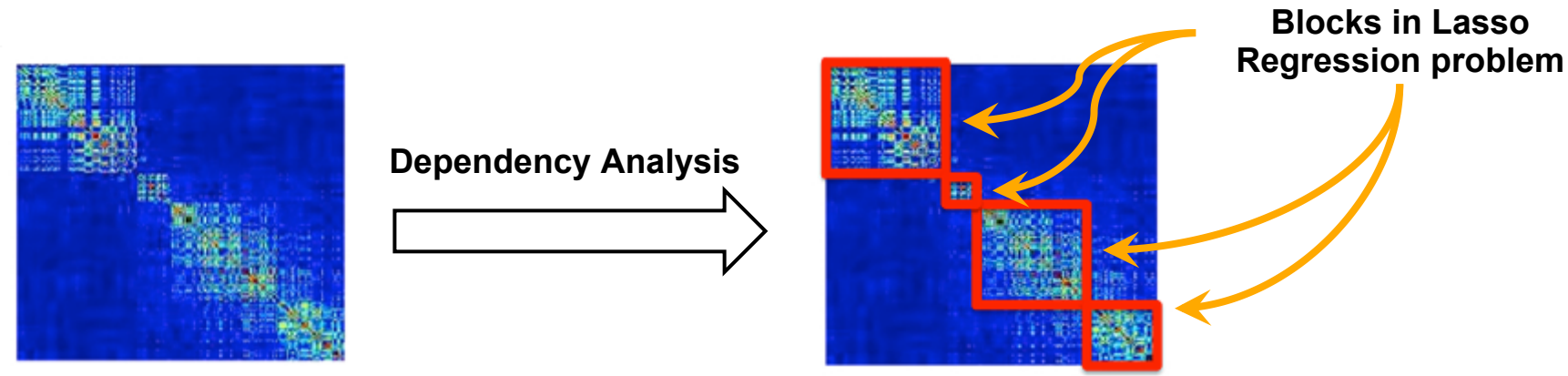
Topic Models, Lasso Regression, DNNs, all essentially do this:





Properties of I-C ML algos

- (2) Block-structured dependencies
  - Model parameters NOT independent, form blocks
  - Inside a block: should update sequentially
  - Between blocks: safe enough to model-parallelize

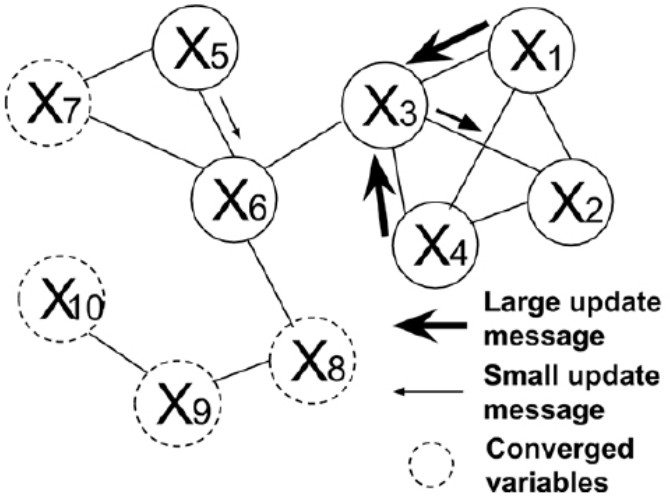


Properties of I-C ML algos

- (3) Non-uniform convergence
  - Some model parameters converge much faster!

Pagerank is a famous example.

Also applies to many ML algos, especially Lasso Regression and DNN



ML Properties vs BigML Platforms

- Data/Model-parallel ML algos are:
  - Iterative-convergent
    - (1) Self-healing/error-tolerant
    - (2) Have block-structured dependencies
    - (3) Exhibit non-uniform convergence

- How do Spark, GraphLab, Petuum fit the above?

Spark: I-C done faster than Hadoop

- Hadoop's problem: can't execute many MapR iterations quickly
  - Must write to HDFS after every reduce()

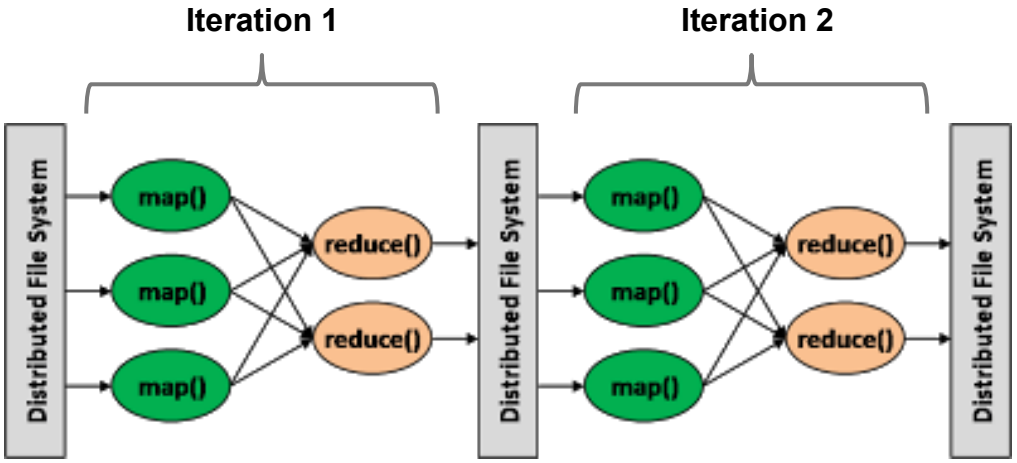
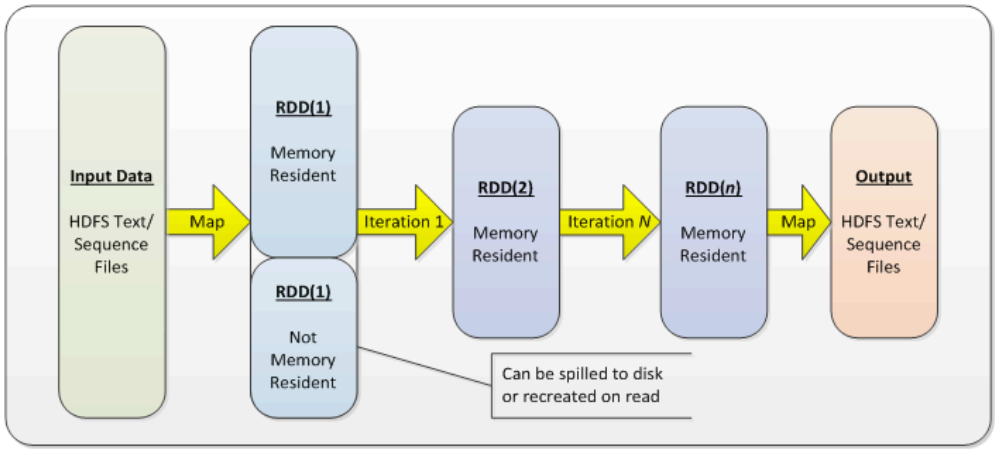


Image source: dzone.com

Spark: I-C done faster than Hadoop

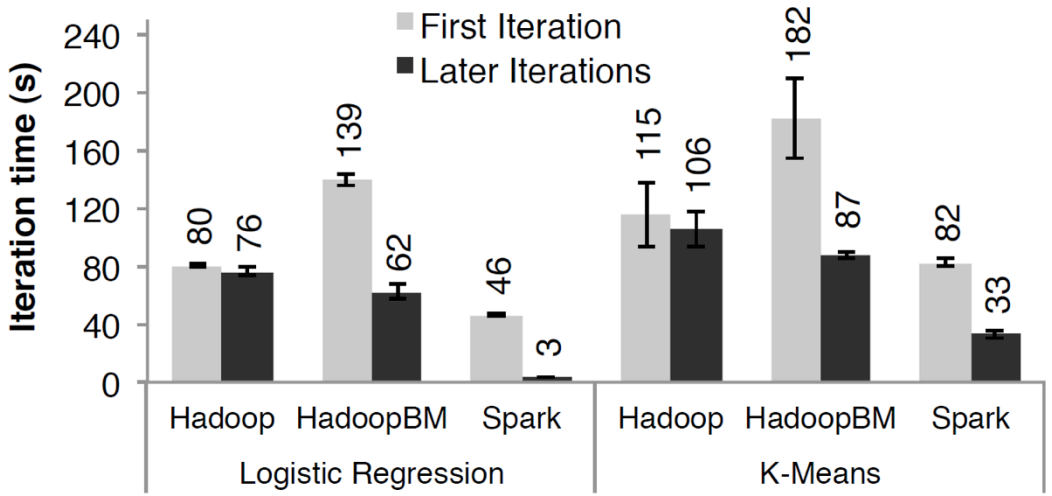
- Spark's solution: **Resilient Distributed Datasets (RDDs)**
  - Input data → load as RDD → apply transforms → output result
  - RDD transforms strict superset of MapR
  - RDDs cached in memory, avoid disk I/O



Source: ebaytechblog.com

Spark: I-C done faster than Hadoop

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

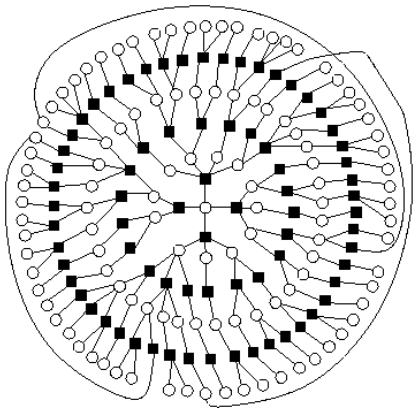


Zaharia et al. (2012)

GraphLab: Model-parallel graphs

- Graph-structured problems really model-parallel
  - Common reason: graph data not IID; data-parallel style poor fit
  - In ML: sparse MatrixFact, some graphical models, pagerank

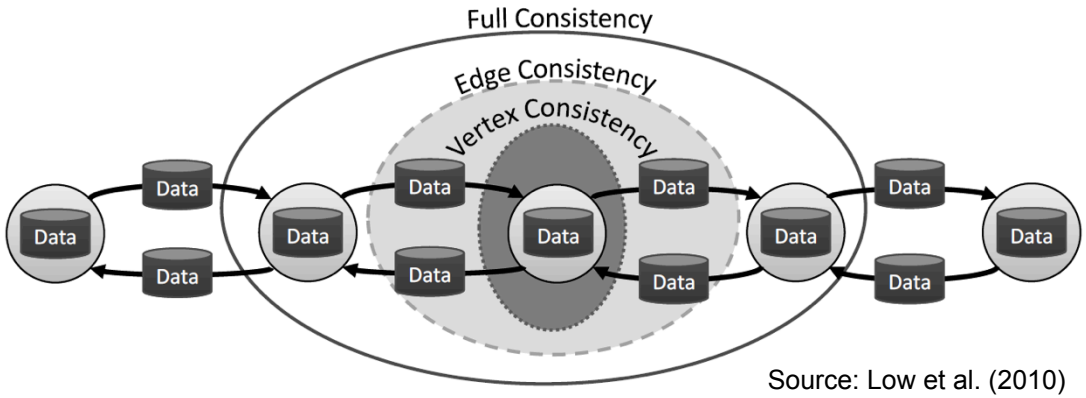
- How to correctly parallelize graph ML algos?



Source: M. Wainwright

GraphLab: Model-parallel graphs

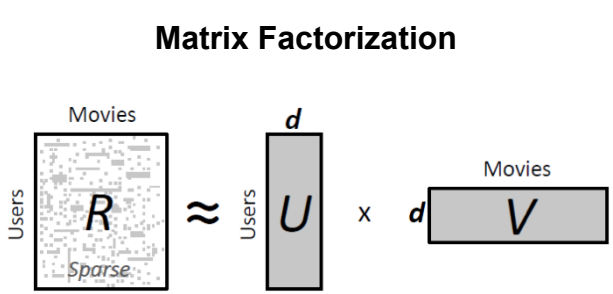
- GraphLab **Graph consistency models**
  - Guide search for “ideal” model-parallel execution order
  - ML algo correct if input graph has all dependencies
  - Similar to “block structures”: Graph = “hard” deps, Blocks = “soft”

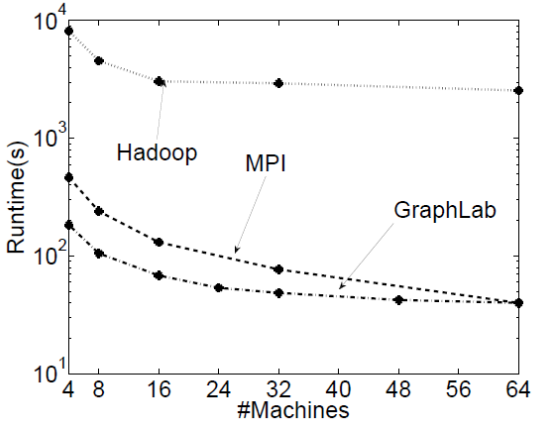


Source: Low et al. (2010)

GraphLab: Model-parallel graphs

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



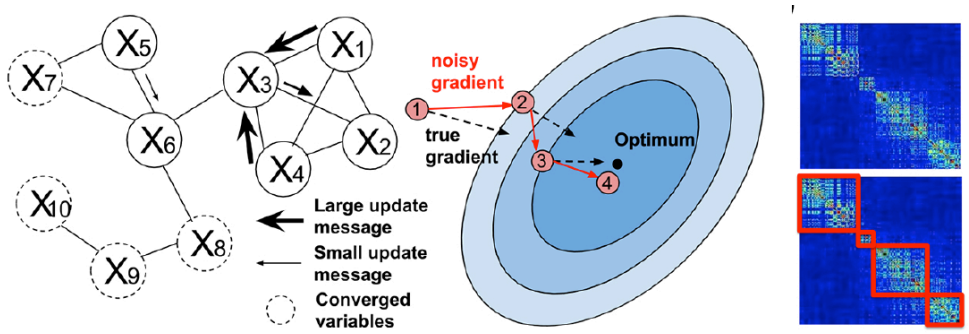


Source: Low et al. (2012)

Petuum: ML props = 1st-class citizen

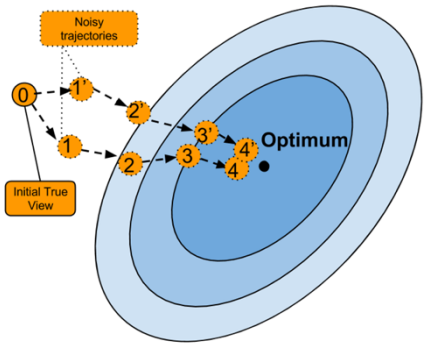
- Idea: ML properties expose new speedup opportunities

- Can data/model-parallel ML run faster if we
  - Allow error in model state? (error-tolerance)
  - Dynamically compute model dependencies? (block structure)
  - Prioritize parts of the model? (non-uniform convergence)



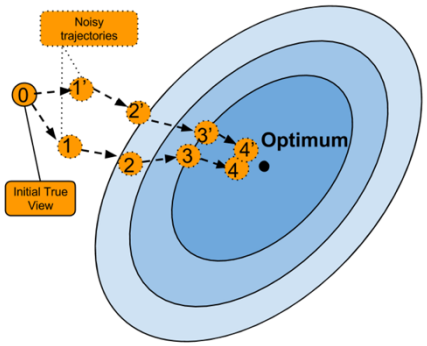
Petuum: ML props = 1st-class citizen

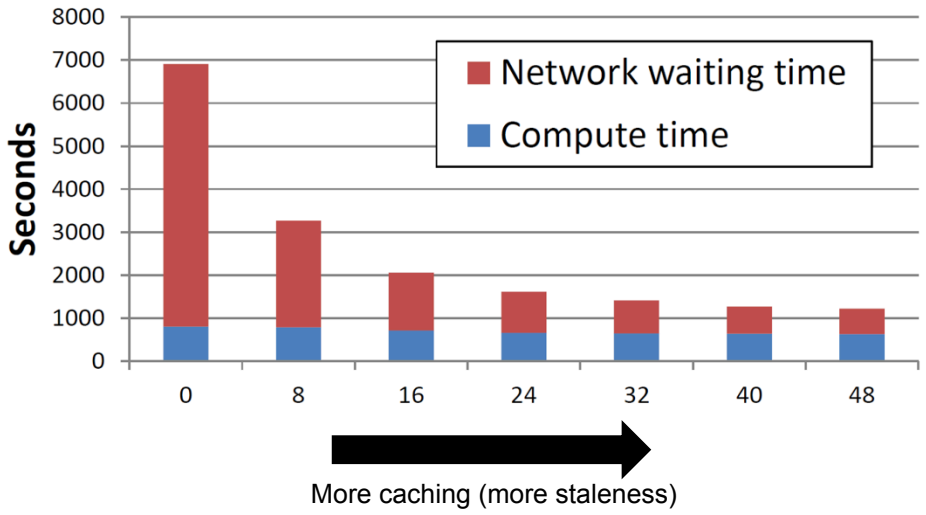
- Error tolerance via Stale Sync Parallel (SSP) Parameter Server (PS)
  - ML Insight 1: old, cached params = small deviation to current params
  - ML Insight 2: deviation strictly limited => ML algo still correct



Petuum: ML props = 1st-class citizen

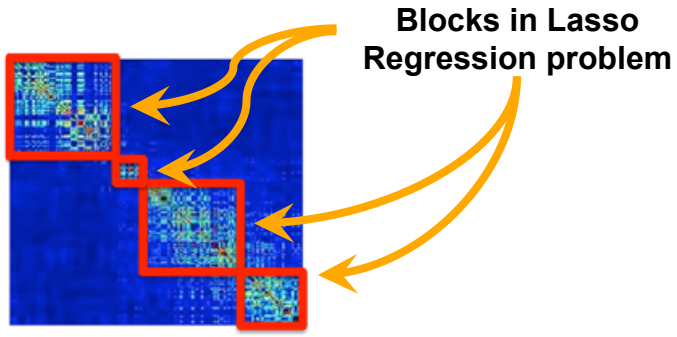
- Error tolerance via Stale Sync Parallel **Parameter Server (PS)**
  - System Insight 1: ML algos bottleneck on network comms
  - System Insight 2: More caching => less comms => faster execution





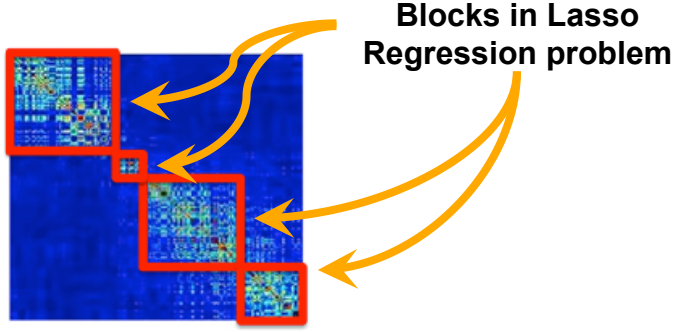
Petuum: ML props = 1st-class citizen

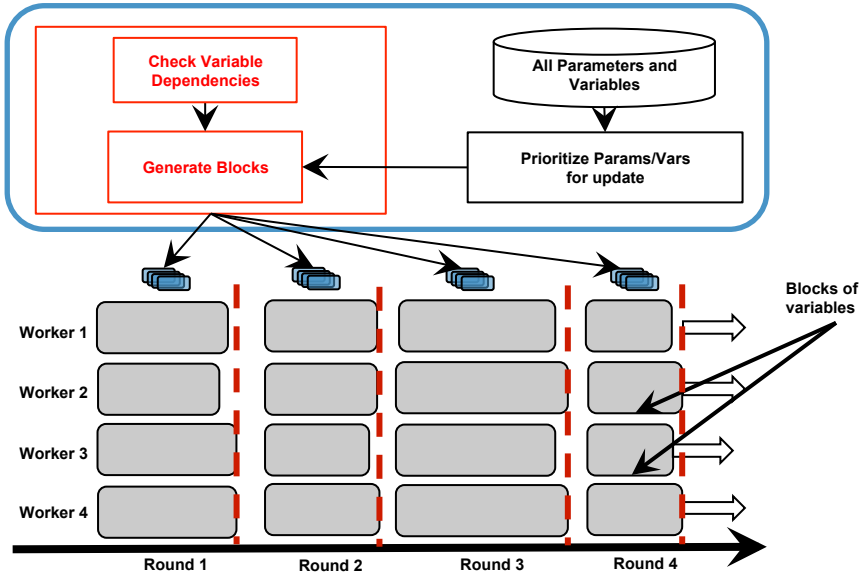
- Harness Block dependency structure via **Scheduler**
  - Model-parallel execution = update many params at same time
  - ML Insight 1: Correctness affected by inter-param dependency
  - ML Insight 2: Even w/o explicit graph, can still compute deps



Petuum: ML props = 1st-class citizen

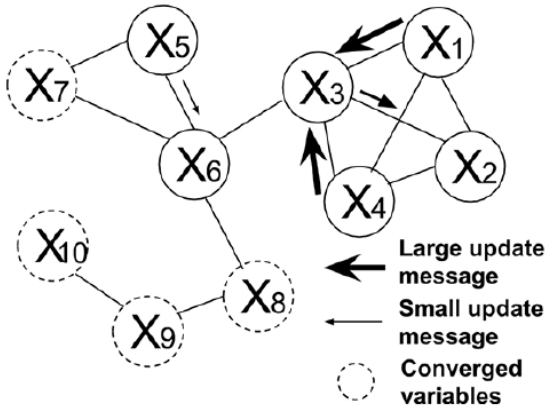
- Harness Block dependency structure via **Scheduler**
  - System Insight 1: Pipeline scheduler to hide latency
  - System Insight 2: Load-balance blocks to prevent stragglers





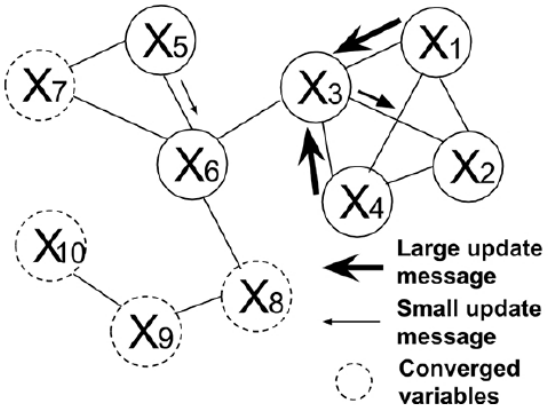
Petuum: ML props = 1st-class citizen

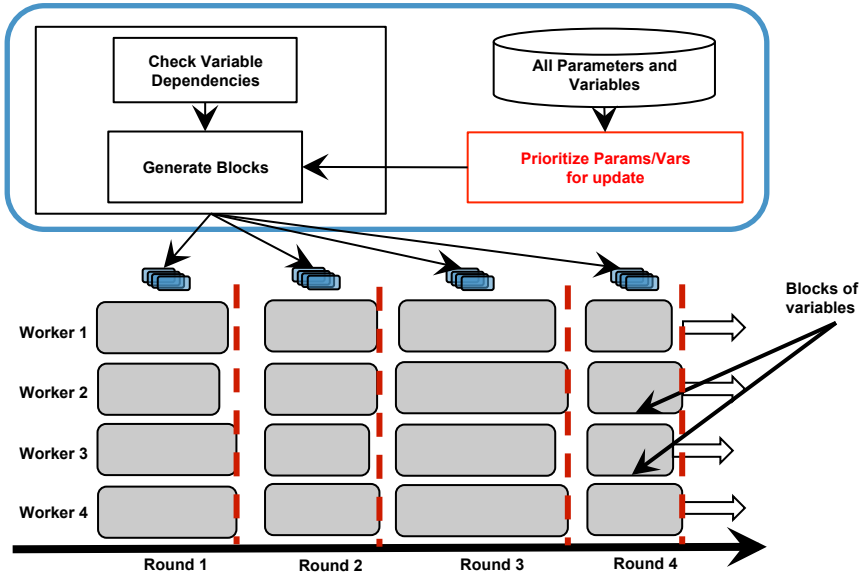
- Exploit Uneven Convergence via **Prioritizer**
  - ML Insight 1: “Steepest descent” - progress correlated with last iter
  - ML Insight 2: Complex model deps => params converge at diff rates



Petuum: ML props = 1st-class citizen

- Exploit Uneven Convergence via **Prioritizer**
  - System Insight 1: Prioritize small # of vars => fewer deps to check
  - System Insight 2: Great synergy with **Scheduler**





Open research topics

- Early days for data-, model-parallelism
  - New properties, principles still undiscovered
  - Potential to accelerate ML beyond naive strategies

- Deep analysis of BigML systems limited to few ML algos
  - Need efforts at deeper, foundational level

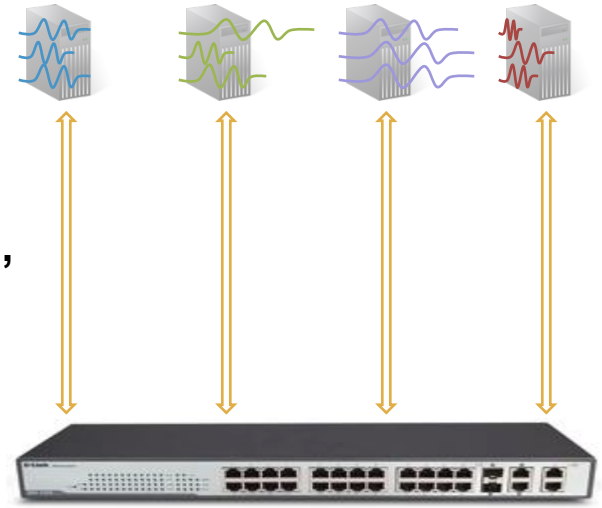
- Major obstacle: lack common formalism for data/model parallelism
  - Model of ML execution under error due to imperfect system?
  - Model not just “theoretical” ML costs, but also system costs?

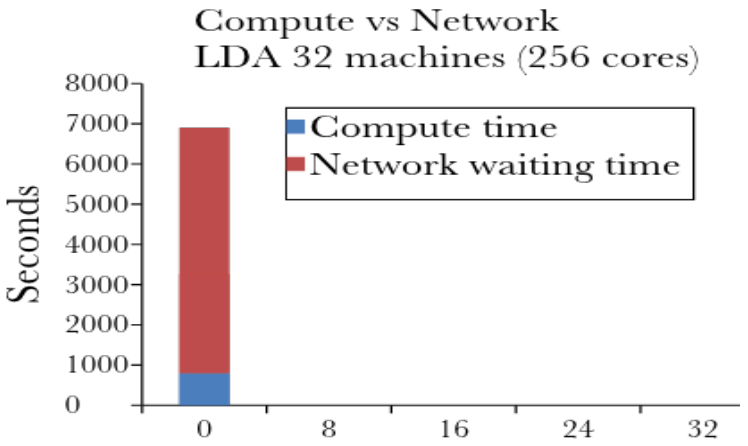
No Ideal Distributed System!

- **Two distributed challenges for ML:**
  - Networks are slow
  - “Identical” machines rarely perform equally

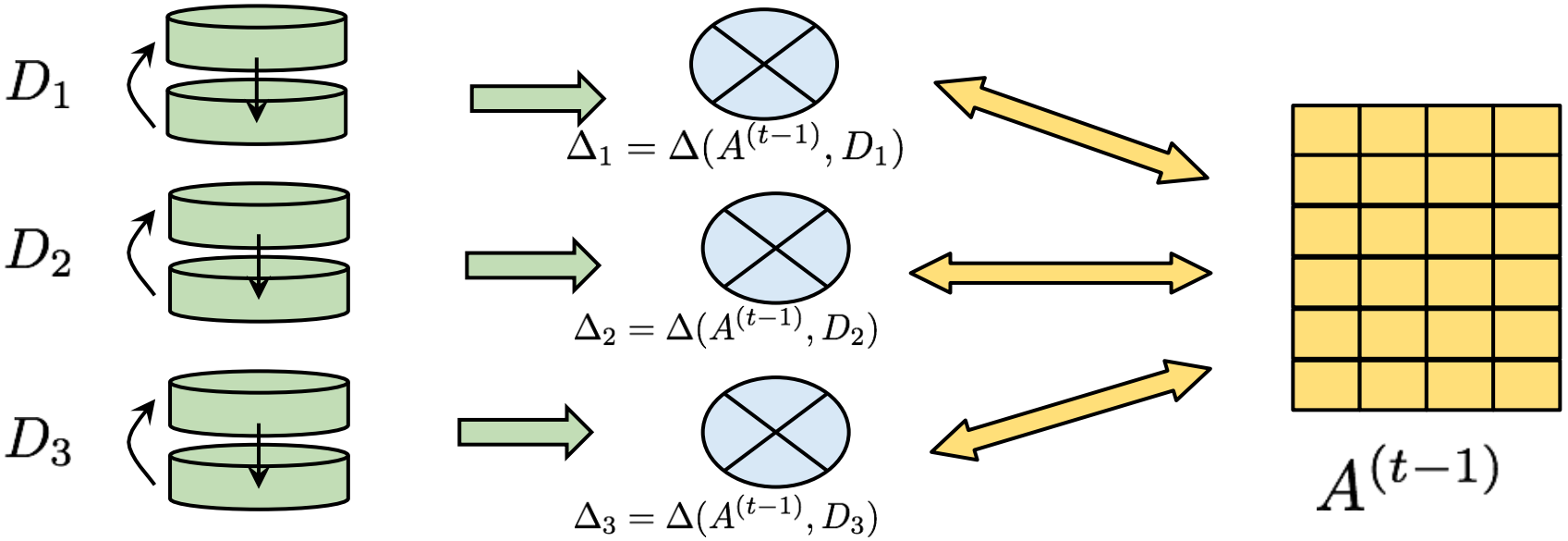
Unequal performance

Low bandwidth, High delay





Recall Data Parallelism



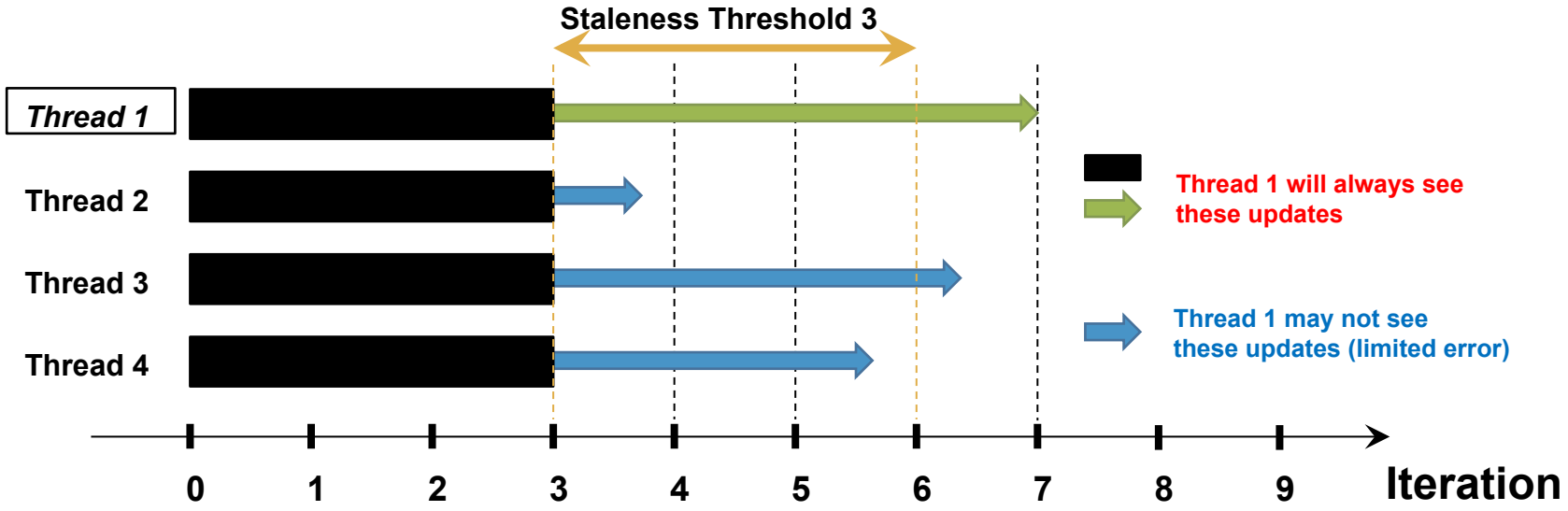
Additive Updates

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

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

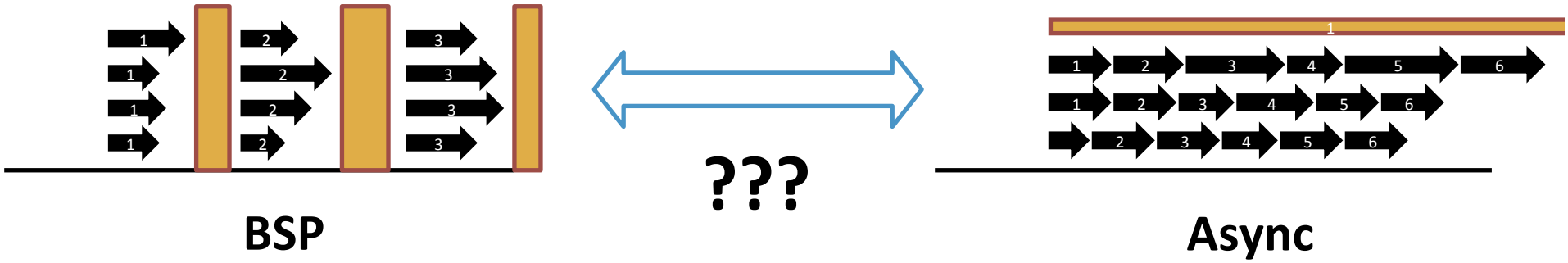
High-Performance Consistency Models for Fast Data-Parallelism

- Recall **Stale Synchronous Parallel (SSP)**
  - Asynchronous-like speed, BSP-like ML correctness guarantees
  - Guaranteed age bound (staleness) on reads
  - C.f.: no-age-guarantee **Eventual Consistency** seen in Cassandra, Memcached



The BSP-Async dichotomy

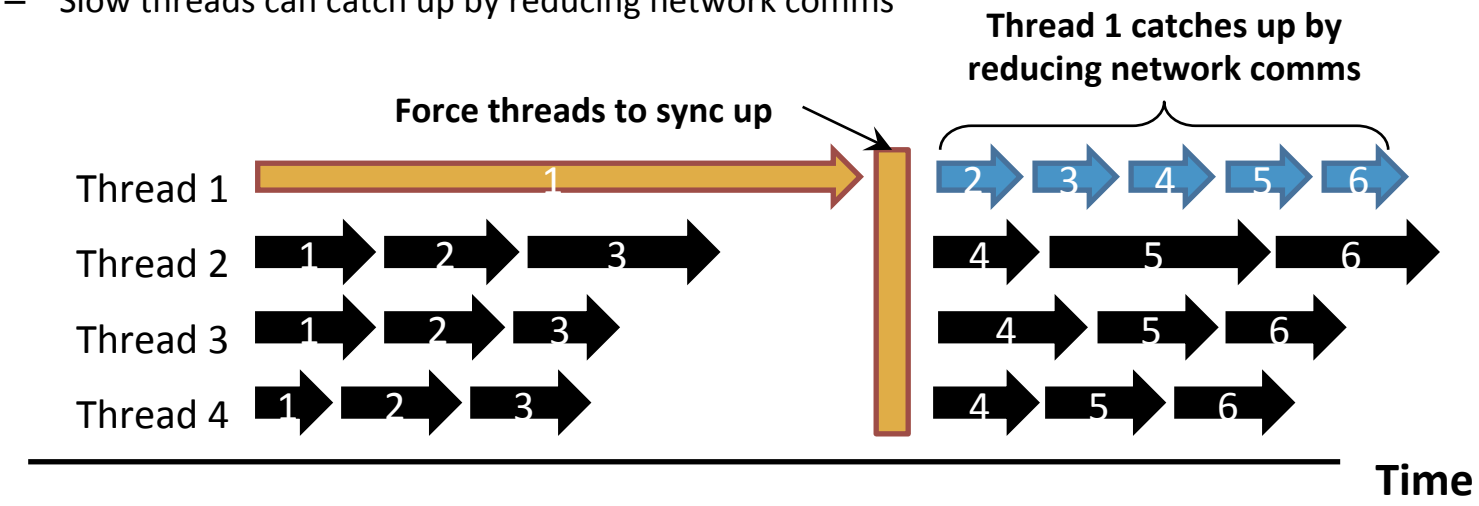
- **BSP**
  - Barrier after every iteration; wait for stragglers
  - Since all comms occur at barrier, the barrier itself can be slow
- **Async**
  - No barriers, but risk unlimited straggler duration
- **Want best of BSP (slow, correct) and Async (fast, no guarantees)**



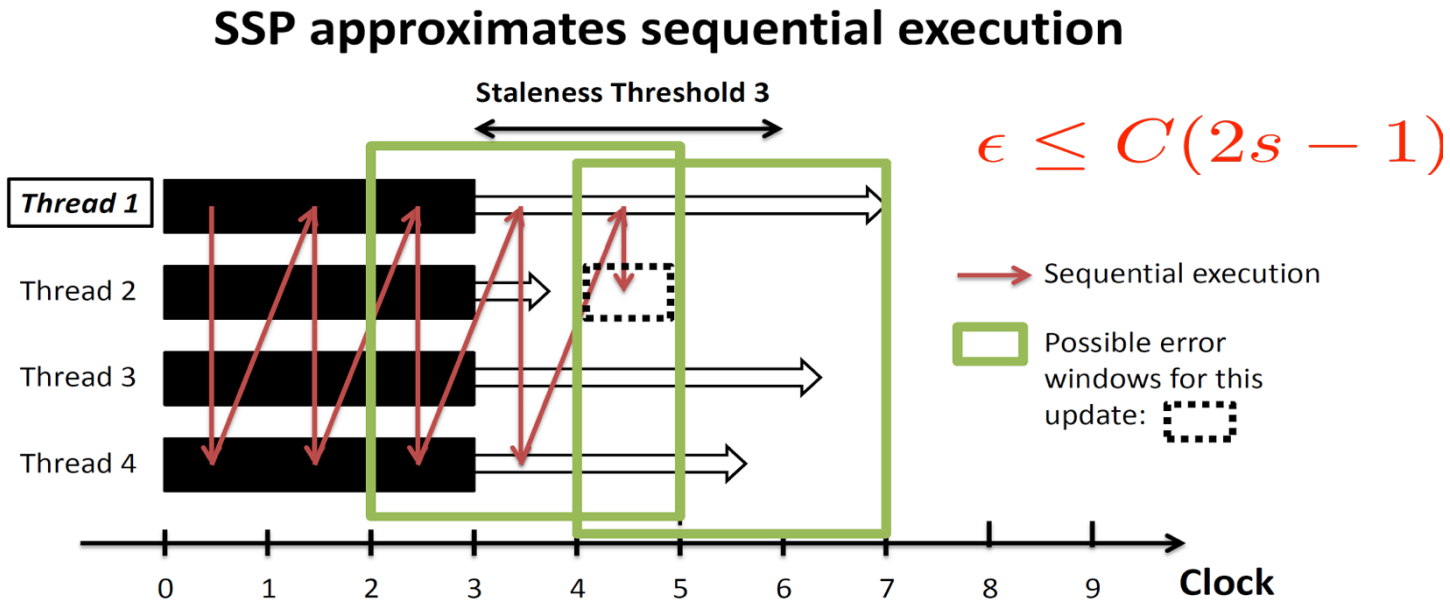
SSP is best-of-both-worlds

- **"Partial" synchronicity**
  - Spread network comms evenly (don't sync unless needed)
  - Threads usually shouldn't wait – but mustn't drift too far apart!

- **Straggler tolerance**
  - Slow threads can catch up by reducing network comms



Why Does SSP Converge?

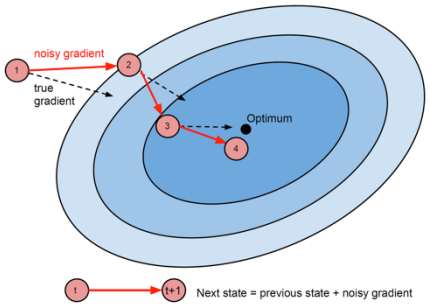


- When a thread reads a parameter, # of "missing updates" is bounded
- Partial, but bounded, loss of serializability
- Hence numeric error in parameter also bounded

SSP Convergence Theorem

- **Goal:** minimize convex $f(\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} f_t(\mathbf{x})$

  (Example: Stochastic Gradient)
  - L-Lipschitz, T is num iters, problem diameter F^2
  - Staleness s, using P threads across all machines
  - Use step size $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$



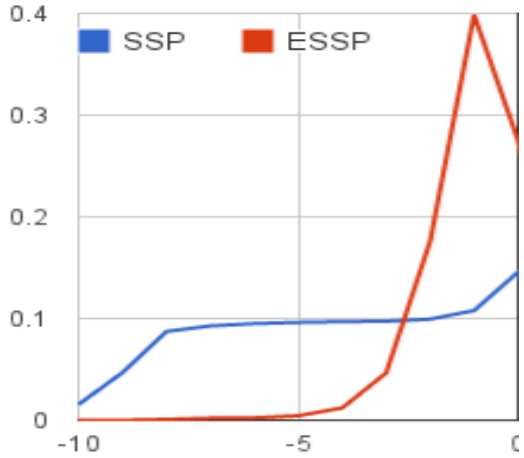
- **SSP converges according to**

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

- Note the RHS interrelation between (L, F) and (s, P)
  - An interaction between theory and systems parameters

Eager SSP (ESSP)

- **Better SSP protocol**
  - Use spare bandwidth to push fresh params sooner
- Figure shows difference in stale reads between SSP and ESSP
- ESSP has fewer stale reads; lower variance



ESSP has faster convergence

- Theorem: Given lipschitz objective f_t and step size η_t,

$$P\left[\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\sigma L^2 + \frac{F^2}{\sigma} + 2\sigma L^2 \epsilon_m\right) \geq \tau\right] \leq \exp\left\{\frac{-T\tau^2}{2\bar{\sigma}_T \epsilon_v + \frac{2}{3}\sigma L^2 (2s+1) P\tau}\right\}$$

where

$R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*)$

L is a lipschitz constant, and ε_m and ε_v are the mean and variance of the observed staleness.

- Intuition: Under ESSP, distance between current param and optimal value decreases exponentially with more iters => guarantees faster convergence than normal SSP.

ESSP has steadier convergence

- Theorem: the variance in the ESSP estimate is

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

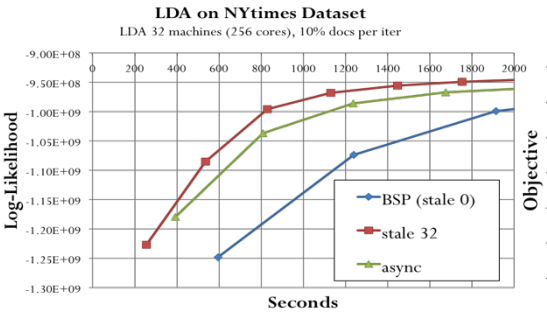
where
$cov(\boldsymbol{v}_1, \boldsymbol{v}_2) := \mathbb{E}[\boldsymbol{v}_1^T \boldsymbol{v}_2] - \mathbb{E}[\boldsymbol{v}_1^T]\mathbb{E}[\boldsymbol{v}_2]$

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

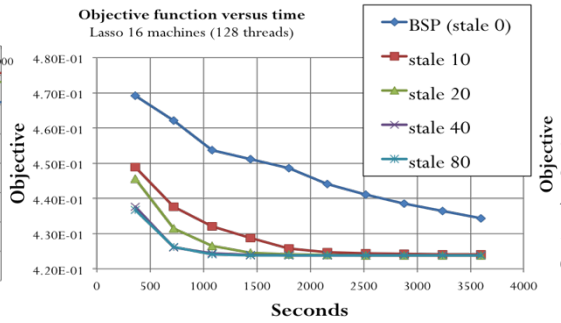
- Intuition: under ESSP, parameter variance decreases near the optimum
- Lower variance => less oscillation in estimate => more confidence in estimate quality and stopping criterion

(E)SSP: Async Speed + BSP Guarantees

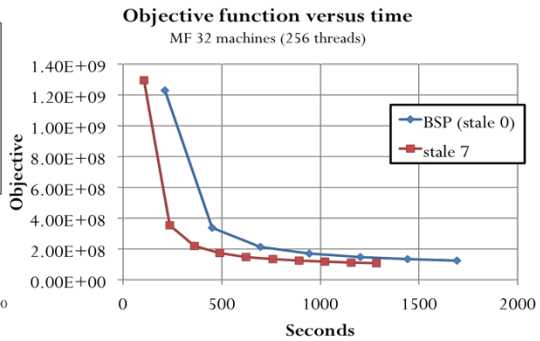
- Massive Data Parallelism
- Effective across different algorithms



LDA

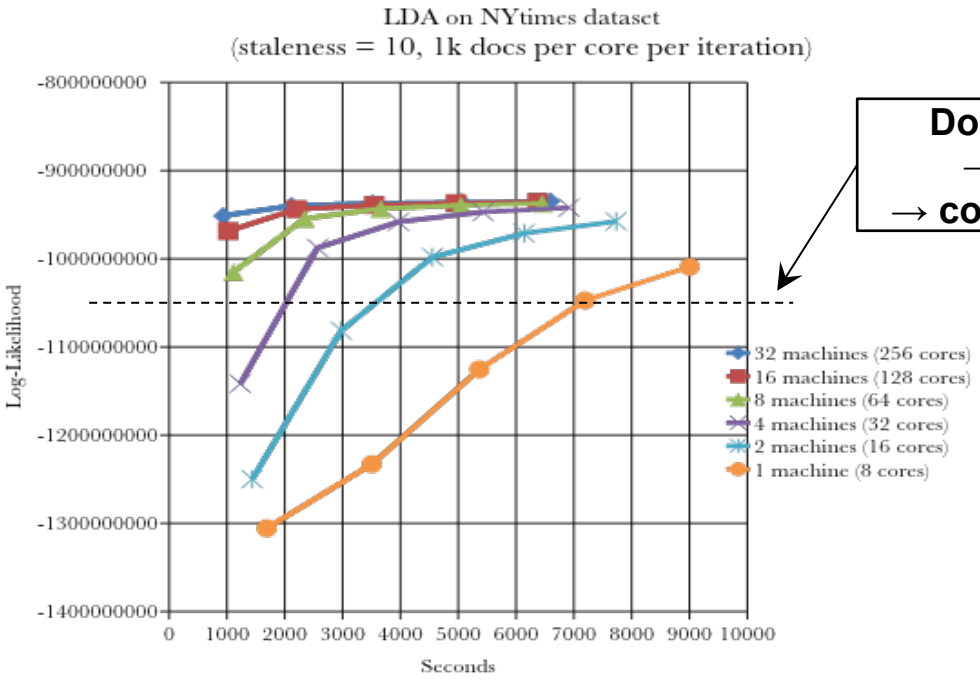


Lasso

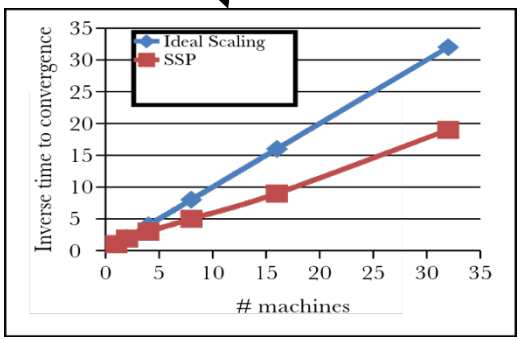


Matrix Fact

(E)SSP Scales With # Machines

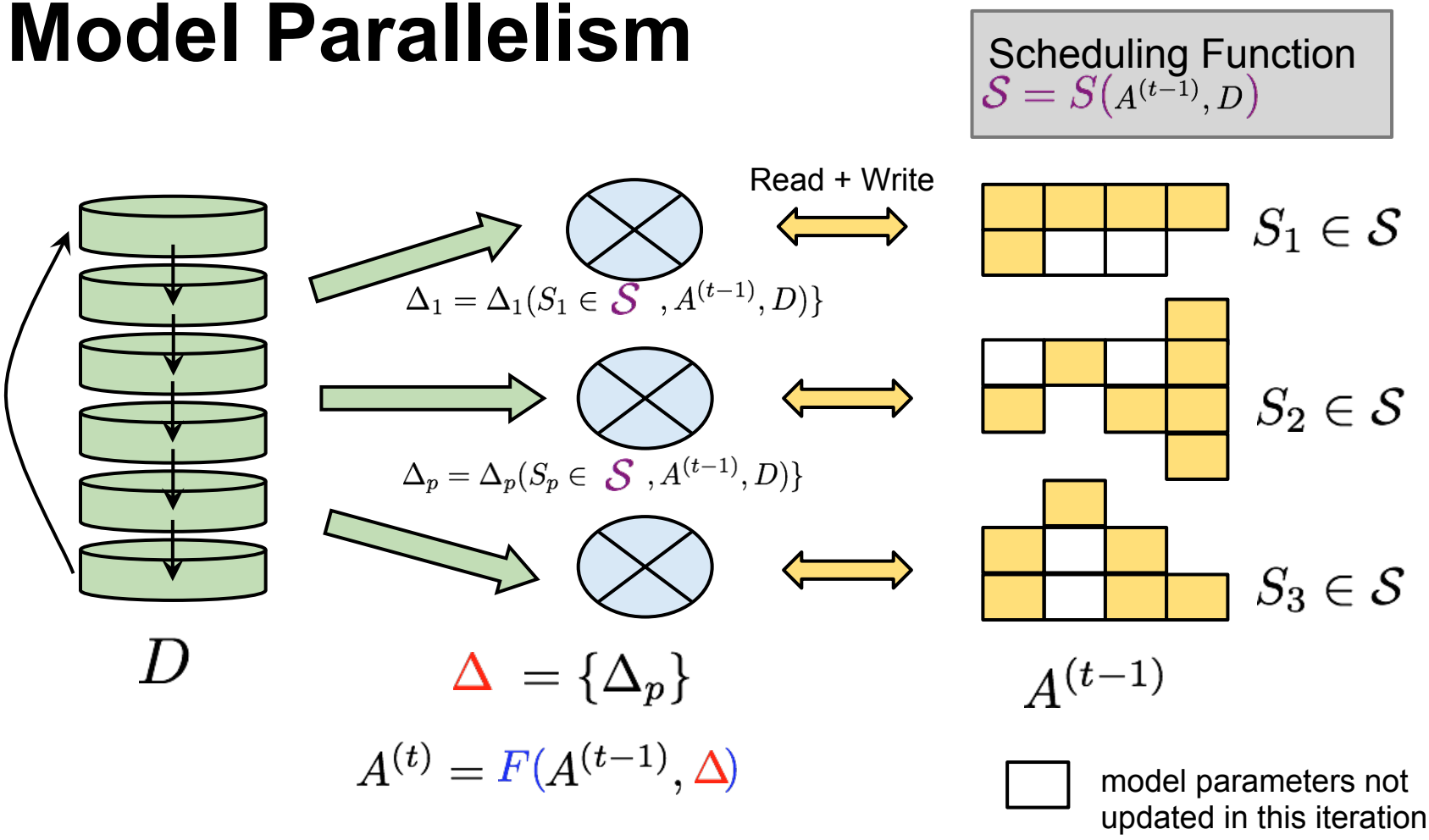


Double # machines:
→ 78% speedup
→ converge in 56% time



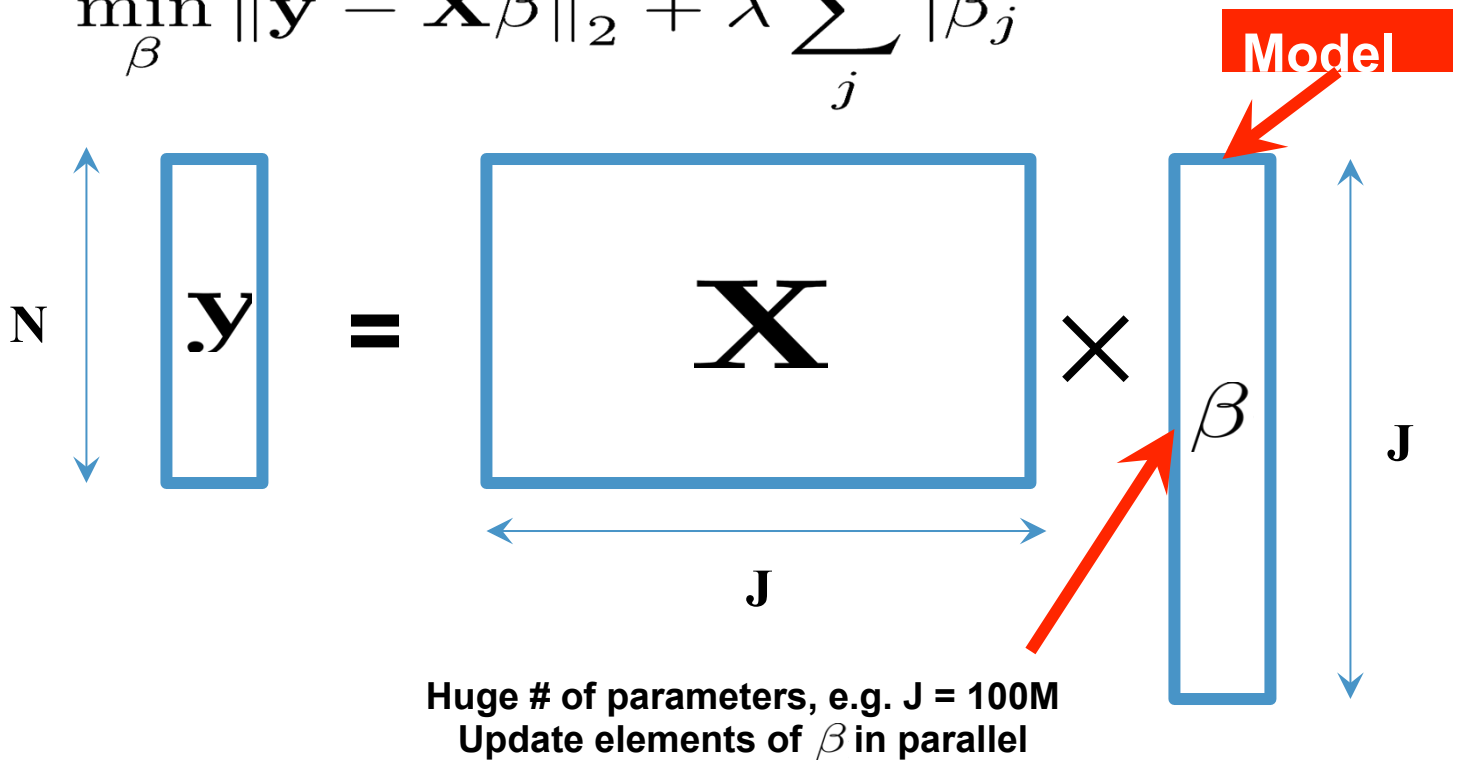
(E)SSP: linear scaling with # machines

Model Parallelism



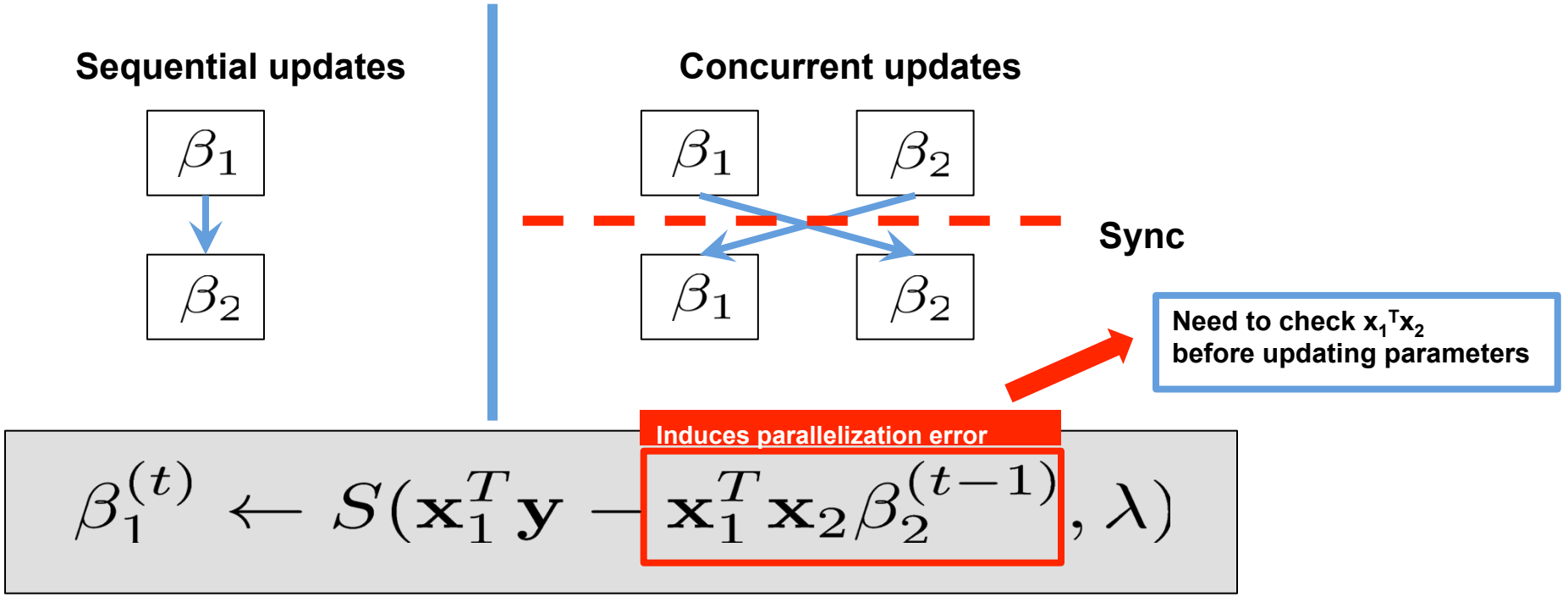
Challenges in Model Parallelism
Lasso as case study

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



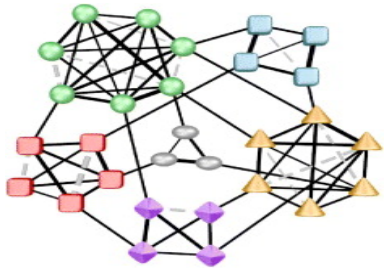
Model Dependencies in Lasso

- Concurrent updates of β may induce errors



The model-parallelism dichotomy

- **Ideal: compute dependency graph, then partition the graph**
  - Expensive and not practical; $O(J^2)$ computation just to find all dependencies
- **Naive: randomly partition**
  - Very fast, but risks parallelization error and algo instability/divergence!
- **Is there a balanced middle ground?**

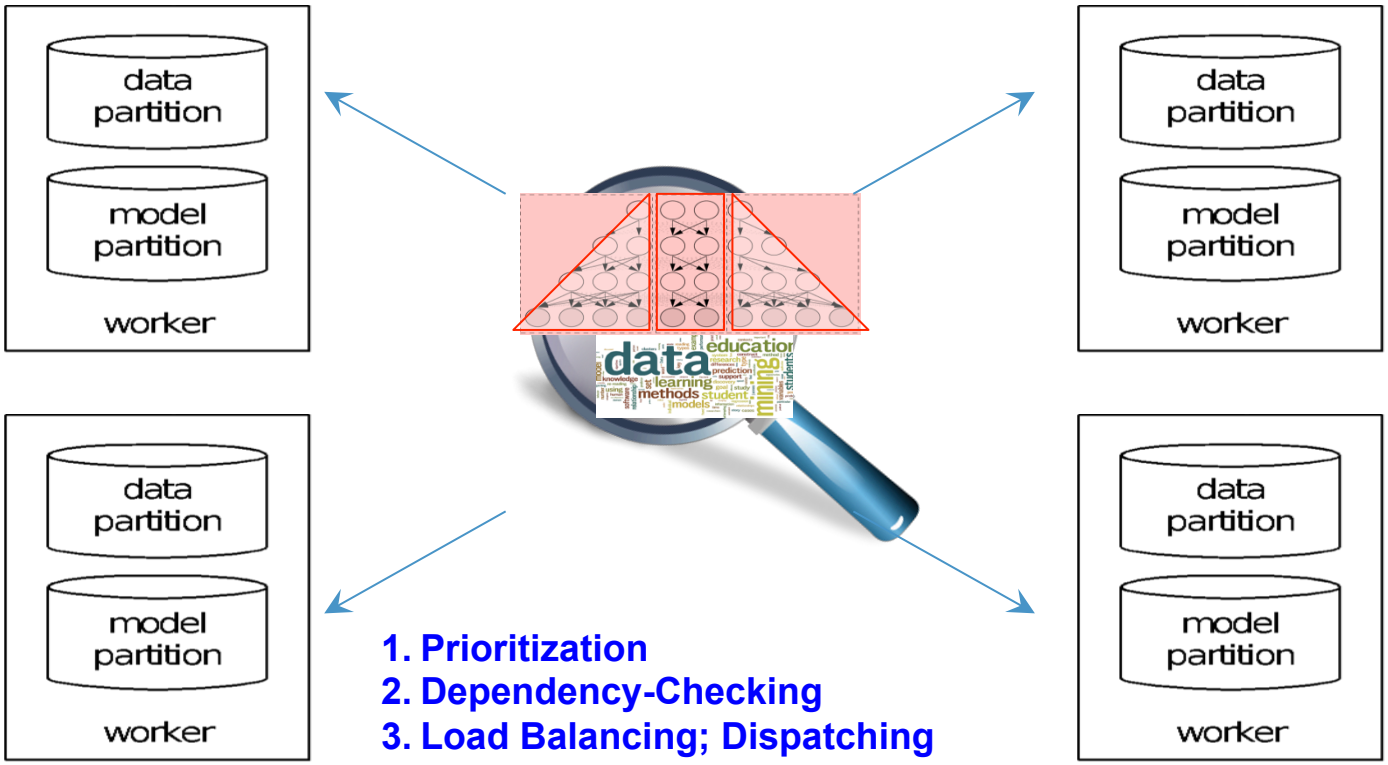


Graph Partition

???

Random Partition

Structure-Aware Parallelization (SAP) as a middle ground



SAP for Lasso

- Prioritizer:

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

Select a few vars which are changing quickly

- Dependency checker:

$$|\mathbf{x}_j \mathbf{x}_k| < \rho \text{ for all } j \neq k \in \mathcal{U}$$

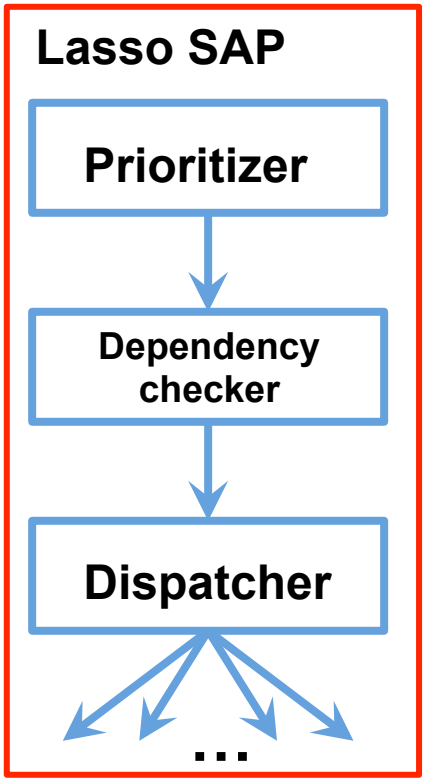
Very fast - only need to check (few) prioritized vars

Vars which violate above must be updated sequentially

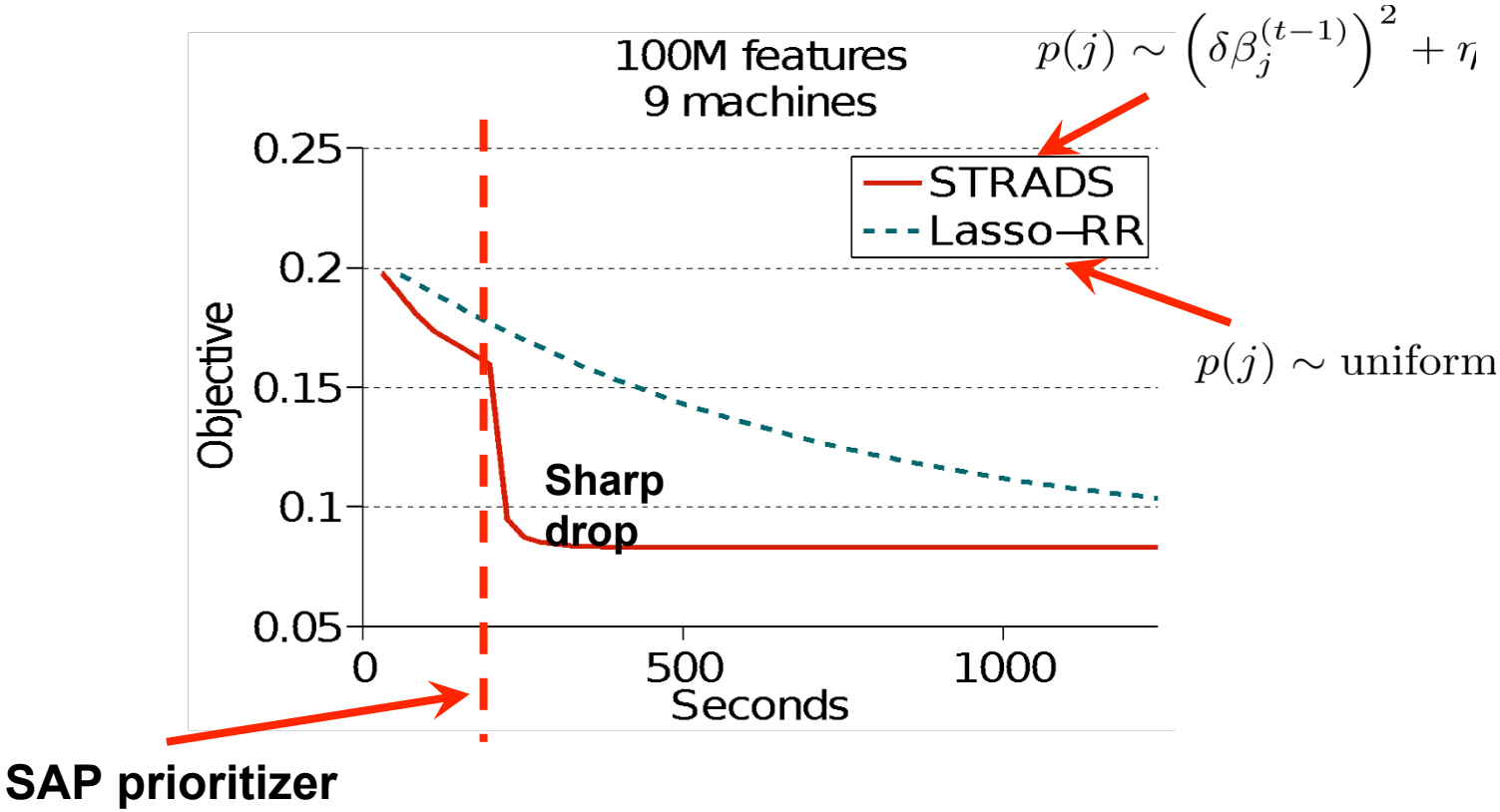
- Dispatcher:

Construct $\{\beta_j\}$ sets according to sequential constraints

Load-balance $\{\beta_j\}$ sets, and dispatch to workers

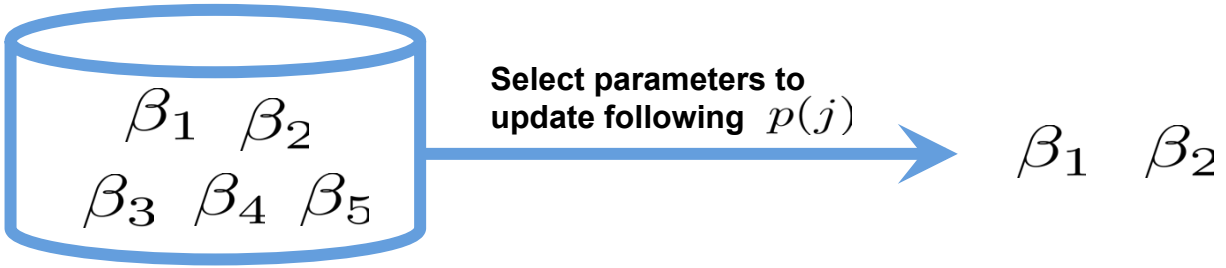


SAP versus Naive partitioning



Theoretical Guarantee on SAP

Guaranteed optimality on Lasso:

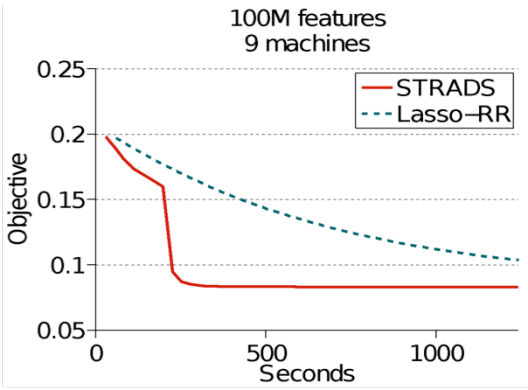


Theorem 1 *Suppose* $\mathcal{P} = \{v_l\}_{l=1}^{L}$ *is the set of indices of coefficients updated in parallel at the t-th iteration, and* ρ *is sufficiently small such that* $\rho\delta\beta_i^{(t)}\delta\beta_j^{(t)} < \epsilon$, *for all* $i \neq j \in \mathcal{P}$, *where* ϵ *is a small positive constant. Then, the distribution* $p(j) \propto \left(\delta\beta_j^{(t)}\right)^2$ *approximately maximizes a lower bound* $\mathcal{L}$ *to the expected decrease in the objective function* $F(\boldsymbol{\beta}^{(t)})$ *after updating coefficients indexed by* $\mathcal{P}$, *where* $\mathcal{L}$ *is defined as*

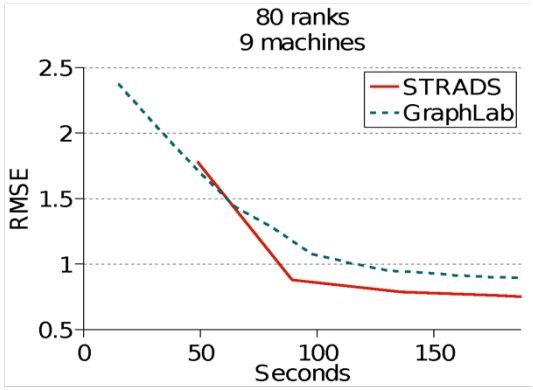
$$\mathcal{L} \leq E_{\mathcal{P}}\left[F(\boldsymbol{\beta}^{(t)}) - F(\boldsymbol{\beta}^{(t)} + \Delta\boldsymbol{\beta}^{(t)})\right]. \quad (1)$$

Convergence

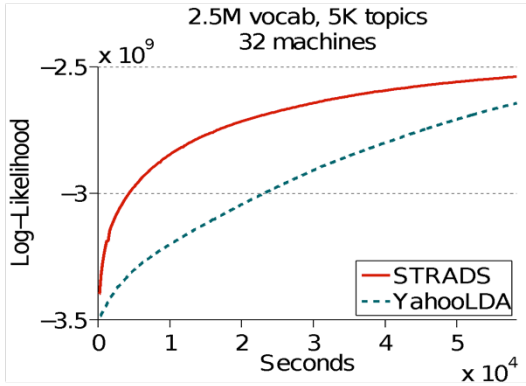
- SAP achieves better speed and objective



Lasso

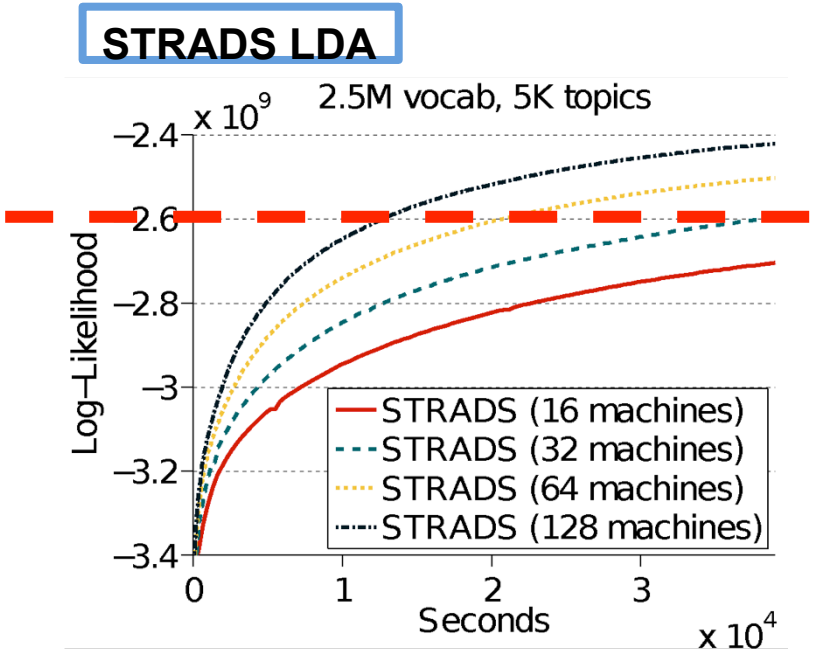


MF

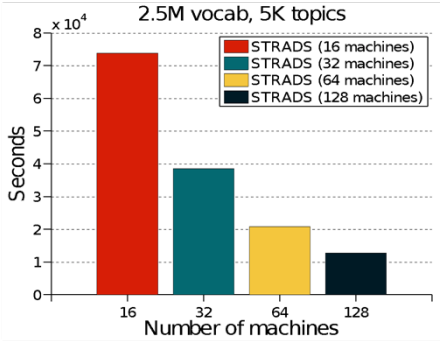


LDA

SAP: Scales With # Machines

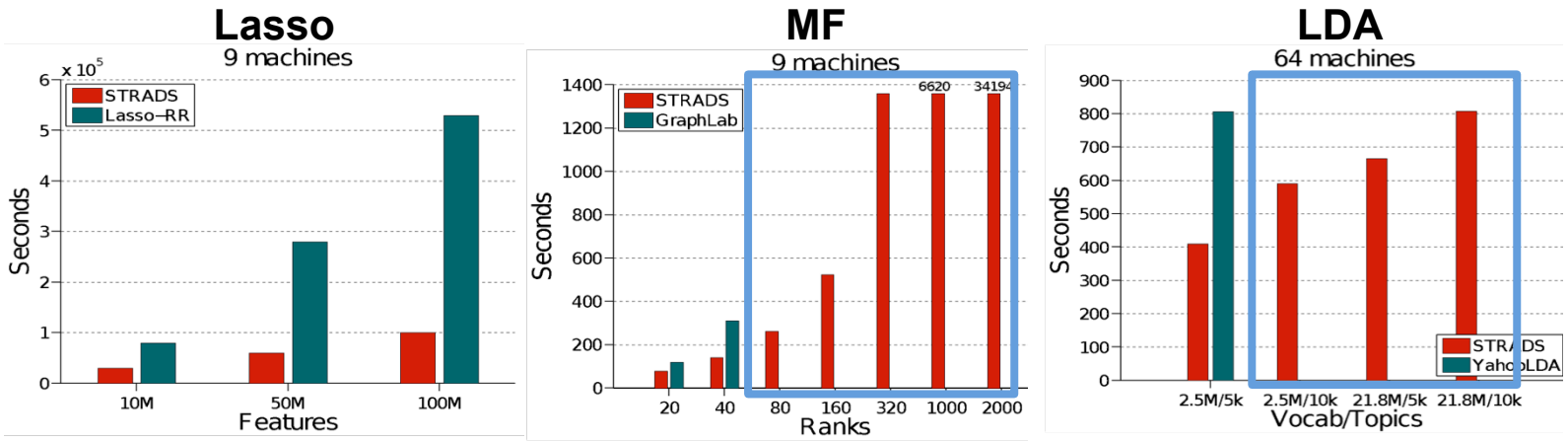


Increase # of machines; time to reach fixed objective decreases



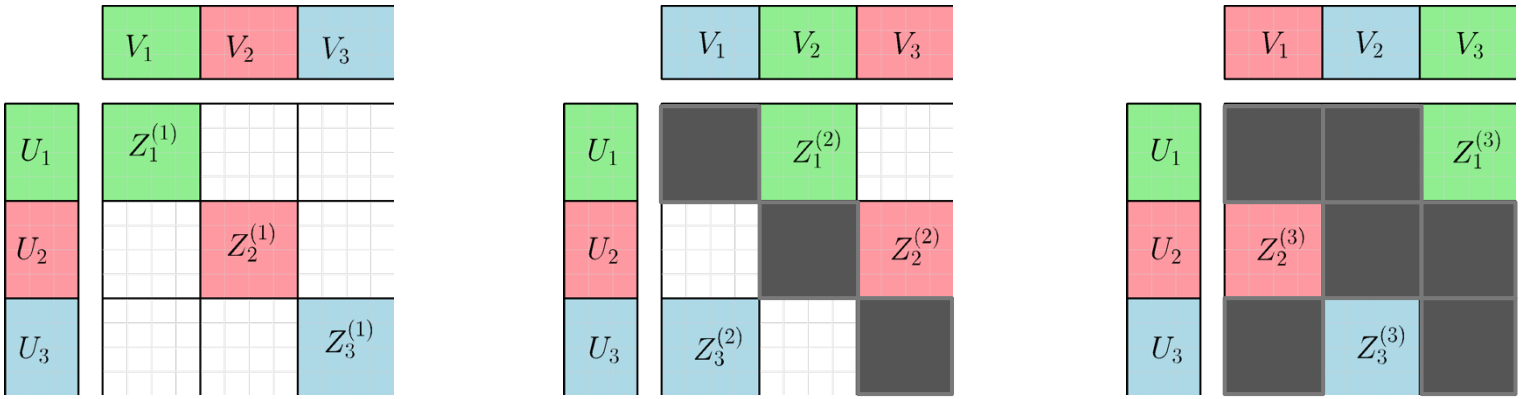
SAP: Bigger Models Now Manageable

- Massive Model Parallelism
- Effective across different models



Another model-parallel idea: Block Scheduling (Gemulla '11)

Partition data & model into *d* × *d* blocks



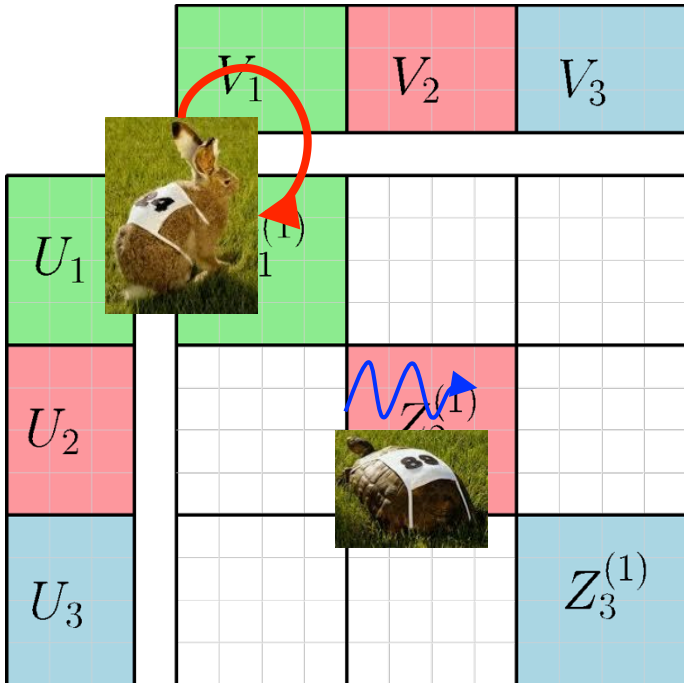
This example: *d=3* strata

In strata: Process blocks with different colors in parallel

Between strata: Process strata sequentially

Fugue: Slow-Worker Agnosticism to solve the straggler problem

- **In real distributed systems:**
  - Slow workers force fast workers to wait
- **Idea: fast workers keep working**
  - Keep updating same block
  - Until slowest worker completes block
- **Strong theoretical guarantees**
  - Faster convergence
  - Lower variance (stable convergence)



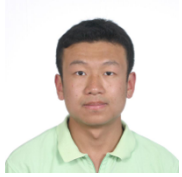
Acknowledgements

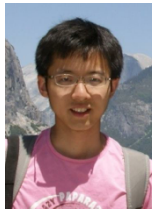
SAILING LAB

Laboratory for Statistical Artificial InteLligence & INtegrative Genomics


Jin Kyu Kim


Seunghak Lee


Jinliang Wei


Wei Dai


Pengtao Xie


Xun Zheng


Abhimanu Kumar

www.sailing.cs.cmu.edu

$$$:

Google IBM


Garth Gibson

Greg Ganger


Phillip Gibbons


James Cipar

Fugue Theorem (1):

Convergence: The Fugue updates and the exact gradient descent updates converge to the same set of limit points asymptotically given that the noise terms form a martingale difference sequence

Remark: a Martingale difference sequence is a weaker assumption than noise terms being independent and is easily satisfied

Fugue Theorem (2):

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

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

Remarks: How does it help?

- The higher order terms are negligible and thus variance decreases in every sub-epoch
- This leads to decrease in variance due to extra work done while waiting

Fugue Theorem (3):

Inter-subepoch variance: Variance between successive sub-epochs decreases making the solution trajectory stable. Specifically, the variance at the end of subepoch S_{n+1} and S_n is

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

Remarks: How does it help?

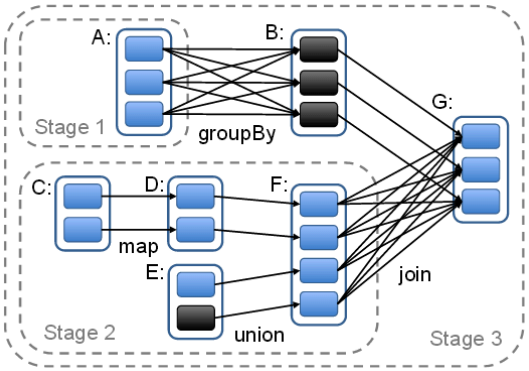
- The higher order terms are negligible
- This leads to decrease in variance every epoch

A New Framework for Large Scale Parallel Machine Learning

(Petuum.org)

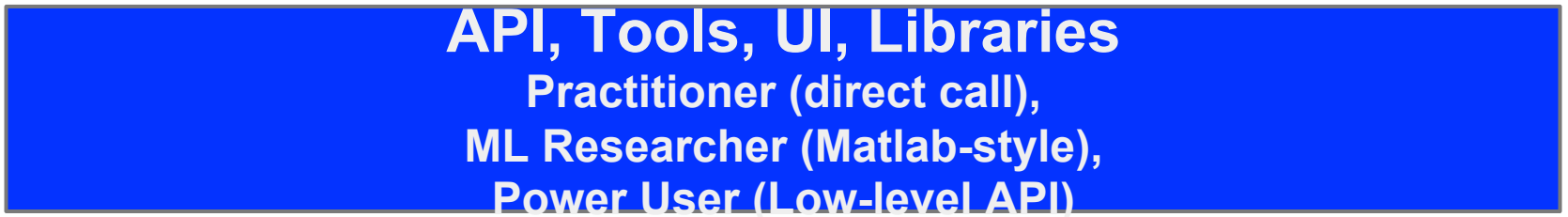
Spark Overview

- Limitations of RDDs:
  - No asynchronous operation
    - RDDs good for fault tolerance, but maybe straggler problem
  - No fine-grained writes
    - Substitute: Spark API has accumulators, broadcast variables



Source: Zaharia et al. (2012)

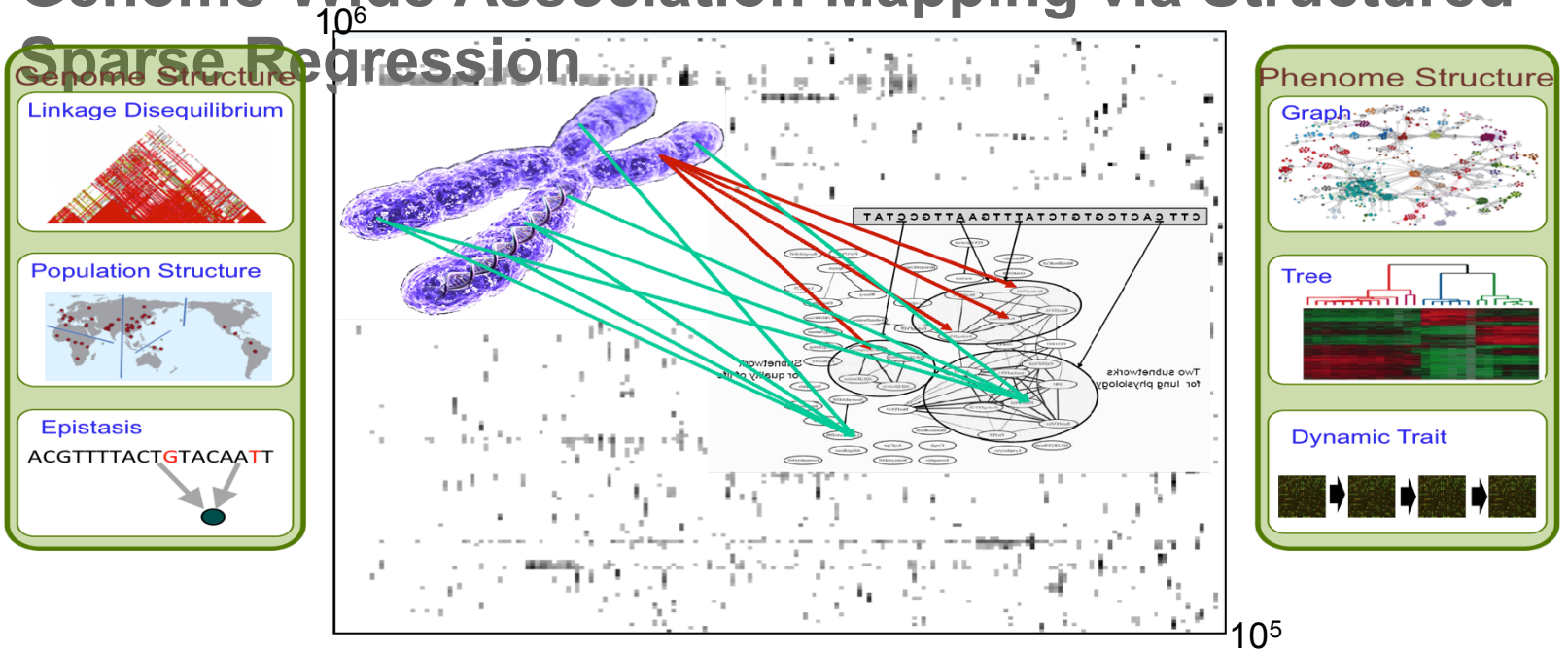
: A General-Purpose Big-ML Framework



$$\vec{\theta} \equiv [\vec{\theta_1}^{\mathrm{T}}, \vec{\theta_2}^{\mathrm{T}}, \ldots, \vec{\theta_k}^{\mathrm{T}}\}^{\mathrm{T}}$$

On Model Parallelism:

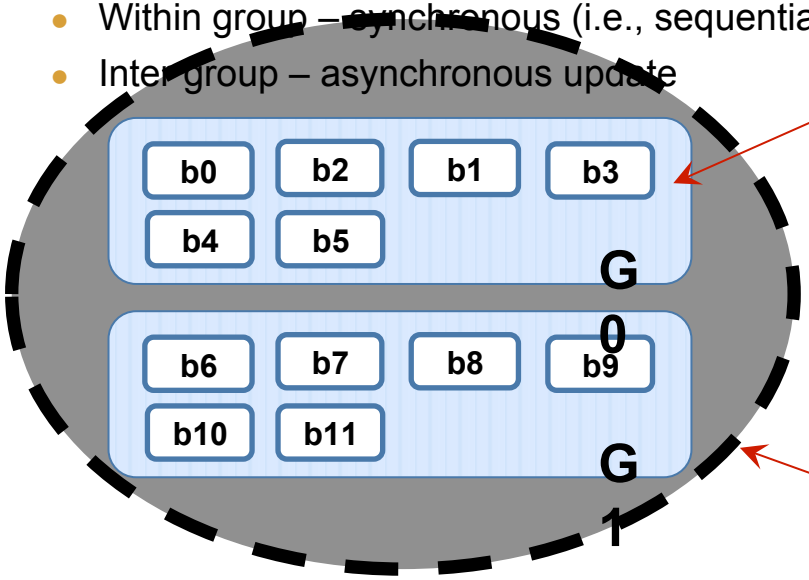
Genome-Wide Association Mapping via Structured Sparse Regression



$$\arg\max_{\beta} \equiv \mathcal{L}(\{\mathbf{x}_i, \mathbf{y}_i\}; \beta) + \Omega(\beta)$$

Model Parallelism

- Determine the degree of parallelization according to system resources
- Non-uniform execution/update policies
  - Within group – synchronous (i.e., sequential) update
  - Inter-group – asynchronous update



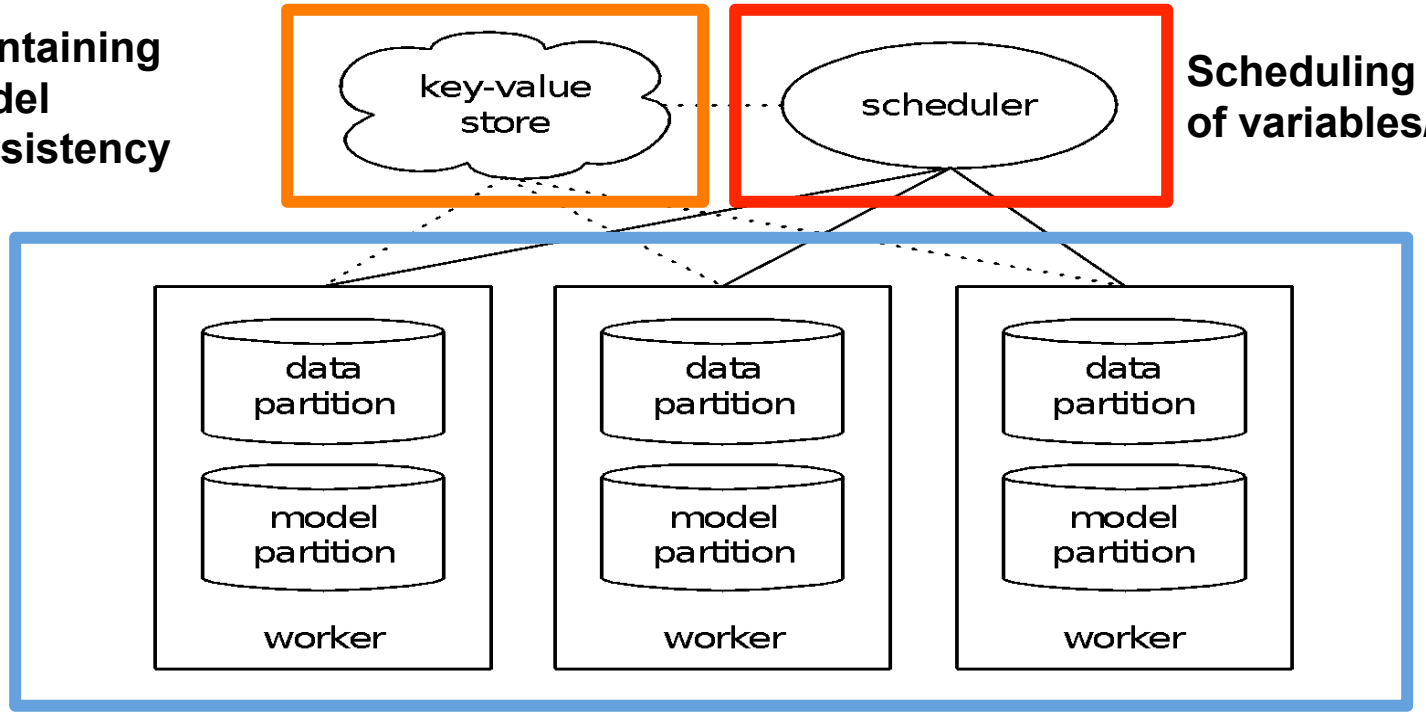
Intra-Group domain
Synchronous Execution/Update domain

G0 = {b0, b1, b2, b3, b4, b5}
G1 = {b6, b7, b8, b9, b10, b11}

Inter-Group domain
Asynchronous Execution/Update domain
Whole data = {G0, G1}

A General Framework

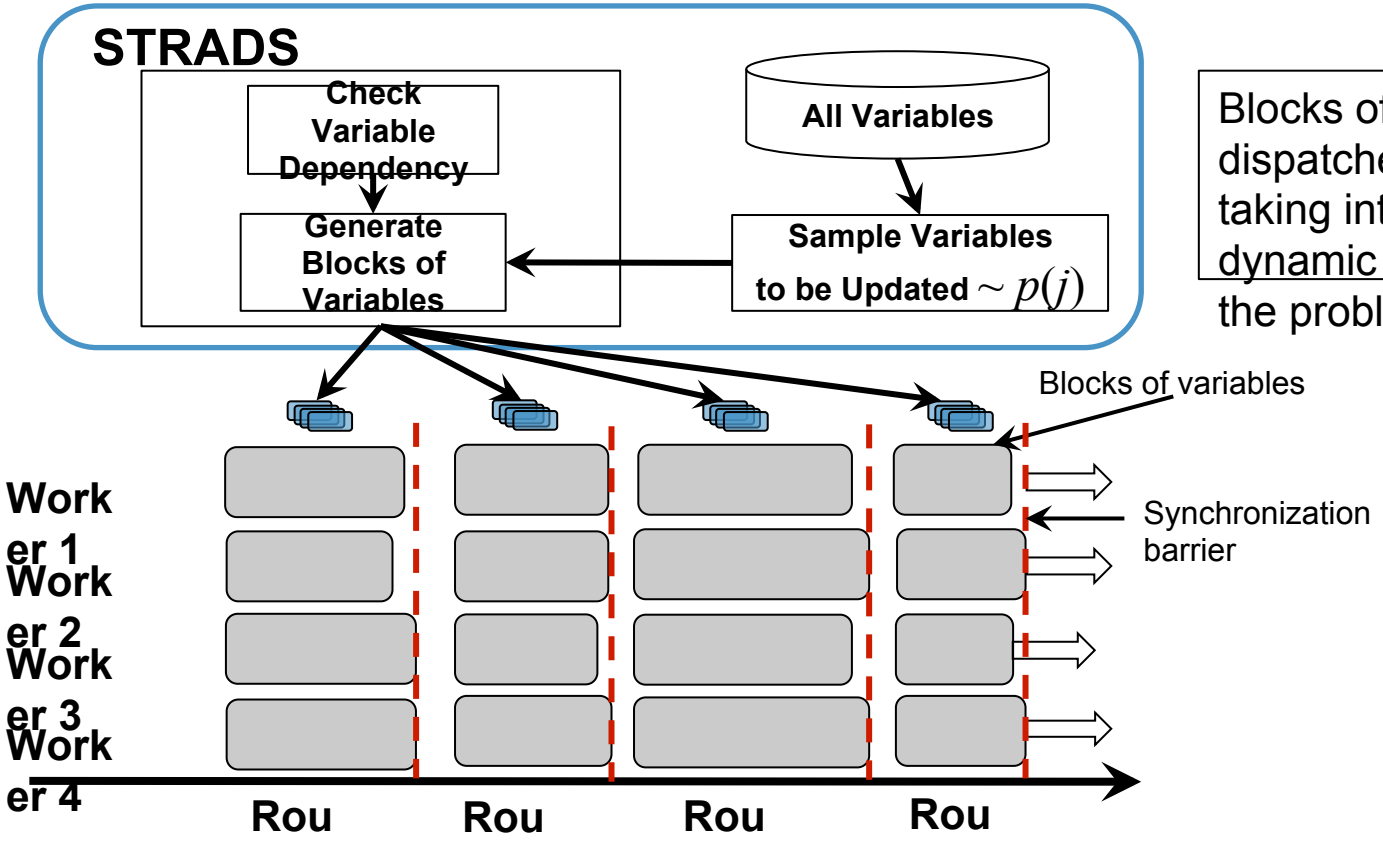
Maintaining model consistency



Scheduling updates of variables/params

Partitioning data and/or models

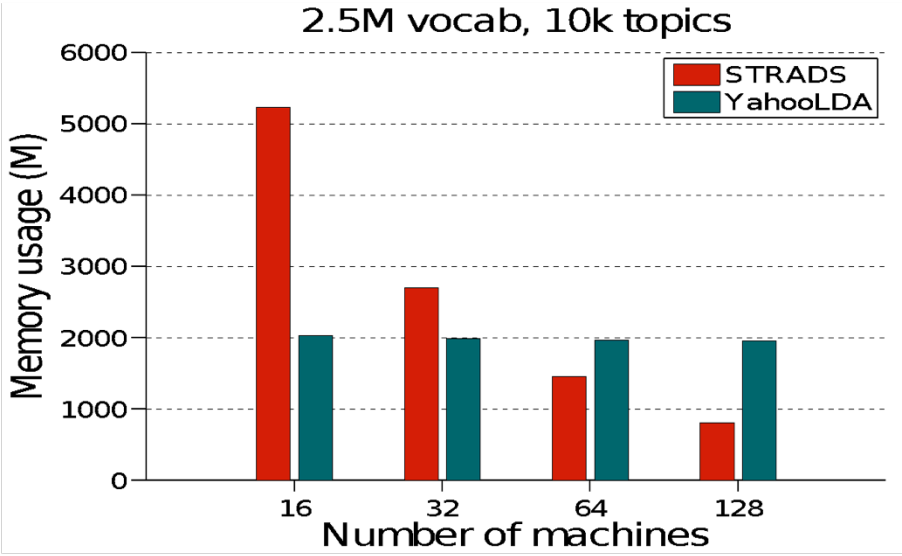
Structure-aware Dynamic Scheduler (STRADS)



Blocks of variables are dispatched to workers taking into account dynamic structures of the problems

Memory Bottleneck in ML Mitigated

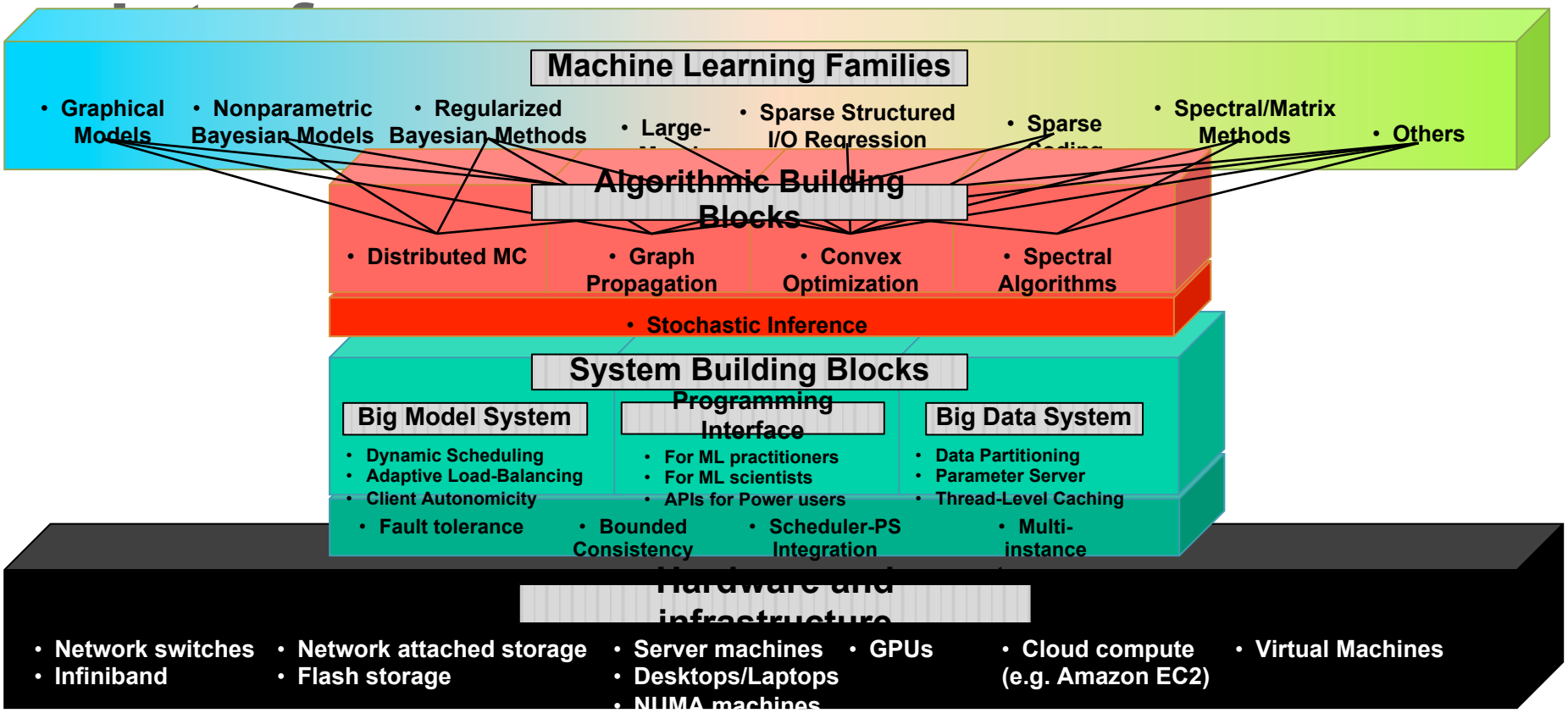
STRADS LDA



STRADS effectively partitions models and data

- **STRADS's memory usage per machine decreases as the number of machines increases**
- **YahooLDA uses the fixed amount of memory per machine, as each machine stores a duplicated copy of word-topic table**

An Algorithmic and System



Naïve Parallel vs Error-controlled Paralle