The Algorithm and System Interface of Distributed Machine Learning

Eric P. Xing

Carnegie Mellon University

Acknowledgement:
Wei Dai, Qirong Ho, Jin Kyu Kim, Abhimanu Kumar, Seunghak Lee, Jinliang Wei, Pengtao Xie, Xun Zheng
Yaoliang Yu, James Cipar, Henggang Cui,
and, Phil Gibbons, Greg Ganger, Garth Gibson
"If a tree falls in a forest and no one is around to hear it, does it make a sound?" --- George Berkeley

Data ≠ Knowledge

- 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
How To Understand Big Data?
Machine Learning !!!
The Scalability Challenge

- Processing power/speed vs. Number of “machines”

- Perfect, Pathetic, Good!
for (t = 1 to T) {
  doThings()
  \(\theta^{t+1} = g(\theta^t, \Delta_f \theta(D))\)
  doOtherThings()
}

This computation needs to be parallelized!
Challenge 1 – Massive Data Scale

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

Source: Cisco Global Cloud Index

Source: The Connectivist
Issue: When is Big Data useful?

- Negative examples
  - “Simple” regression and classification models, with fixed parameter size
  - Intuition: the decrease in the variance of the estimator experiences diminishing returns with more data. At some point, the estimator is simply “good enough” for practical purposes, and additional data/computation is unnecessary

- Positive examples
  - Topic models (used all over internet industry)
  - DNNs (Google Brain, many others)
  - Collaborative filtering (again, used all over internet industry)
  - “Personalized” models
  - Practitioners of the above usually increase model size with more data

- Conjecture: how much data is useful really depends on model size/capacity
Challenge 2 – Gigantic Model Size

Maybe Big Data needs Big Models to extract understanding? But models with >1 trillion params also won’t fit!
Issue: Are Big Models useful?

- In theory
  - Possibly, but be careful not to over-extend

- Beware “statistical strength”
  - “When you have large amounts of data, your appetite for hypotheses tends to get even larger. And if it’s growing faster than the statistical strength of the data, then many of your inferences are likely to be false. They are likely to be white noise.” –Michael Jordan

- In practice
  - Some success stories - could there be theory justification?

- Many topics in topic models
  - Capture long-tail effects of interest; improved real-world task performance

- Many parameters in DNNs
  - Improved accuracy in vision and speech tasks
  - Publicly-visible success (e.g. Google Brain)
Classic algorithms used for decades

- K-means
- Logistic regression
- Decision trees
- Naive Bayes

Challenge 3 – Inadequate support for advanced methods
Challenge

Inadequate support for advanced methods

Today’s statistical models and ML algorithms

- Topic models
- Deep learning
- Lasso regression
Challenge 3

Inadequate support for advanced methods

Today’s statistical models and ML algorithms

Latent space network models

Tree ensembles

Constrained matrix factorization
Issue: Inference Algorithms, or Inference Systems?

- View: focus on inference algorithm
  - Scale up by refining the algorithm
    - Given fixed computation, finish inference faster
- A few examples
  - Quasi-Newton algorithms for optimization
  - Locality sensitive hashing for graphical models (Ahmed et al. 2012)

- View: focus on distributed systems for inference
  - Scale up by using more machines
    - Not trivial: real clusters are imperfect and unreliable; Hadoop not a fix-all
- A few platforms
  - Spark
  - GraphLab
  - Petuum
Issue: Theoretical Guarantees and Empirical Performance

- View: establishing theoretical consistency of estimators gives practitioners much-needed confidence
  - Motivated by empirical science, where guarantees are paramount

- Example: Lasso sparsistency and consistency (Wainwright 2009)
  - Theory predicts how many samples n needed for a Lasso problem with p dimensions and k non-zero elements
  - Simulation experiments show very close match with theory
  - Is there a way to analyze more complex models?

- View: empirical and industrial evidence can provide a strong driving force for experimental research
  - Motivated by industrial practice, particularly at internet companies

- Example: AB testing in industry
  - Principled experimental means of testing new algorithms or feature engineering; makes use of large user base for experimentation
  - Can show whether an new algorithm makes a significant difference to click-through rate, user adoption, etc.
Parallelization Strategies

\[ \theta_{t+1} = \theta_t + \Delta_f \theta(D) \]

New Model = Old Model + Update(Data)
Parallelization Strategies

\[ \vec{\theta}^{t+1} = \vec{\theta}^t + \Delta_f \vec{\theta}(D) \]

New Model = Old Model + Update(Data)

Data Parallel

\[ \Delta \theta(D_1) \quad \Delta \theta(D_2) \quad \Delta \theta(D_3) \]

\[ D \equiv \{ D_1, D_2, \ldots, D_n \} \]
Parallelization Strategies

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

New Model = Old Model + Update(Data)

Data Parallel

Model Parallel

$$\Delta \theta(D)$$

$$\Delta \theta(D_1)$$

$$\Delta \theta(D_2)$$

$$\Delta \theta(D_3)$$

$$\Delta \theta(D)$$

$$\Delta \theta_1(D)$$

$$\Delta \theta_2(D)$$

$$\Delta \theta_3(D)$$

$$\Delta \theta_k(D)$$

$$\tilde{\theta} = [\theta_1^T, \theta_2^T, \ldots, \theta_k^T]^T$$

$$\mathcal{D} \equiv \{D_1, D_2, \ldots, D_n\}$$
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

![Graph showing compute vs network time]

© Eric Xing @ CMU, 2015
Issue: How to approach distributed systems?

- **Idealist view**
  - Start with simplified view of distributed systems; develop elaborate theory

- **Issues being explored:**
  - Information theoretic lower bounds for communication (Zhang et al. 2013)
  - Provably correct distributed architectures, with mild assumptions (Langford et al. 2009, Duchi and Agarwal 2011)

- **How can we build practical solutions using these ideas?**

- **Pragmatist view**
  - Start with real-world, complex distributed systems, and develop a combination of theoretical guarantees and empirical evidence

- **Issues being explored:**
  - Fault tolerance and recovery (Zaharia et al. 2012, Spark, Li et al. 2014)
  - Impact of stragglers and delays on inference, and robust solutions (Ho et al. 2013, Dai et al. 2014, Petuum, Li et al. 2014)
  - Scheduling of inference computations for massive speedups (Low et al. 2012, GraphLab, Kim et al. 2014, Petuum)

- **How can we connect these phenomena to theoretical inference correctness and speed?**
Solution:
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 …
- **Model:** Generic building blocks: loss functions, structures, constraints, priors …
- **Algorithm:** Parallelizable and stochastic MCMC, VI, Opt, Spectrum …
- **Representation:** Compact and informative features
- **System:** Distributed architecture: DFS, parameter server, task scheduler …
- **Hardware:** GPU, flash storage, cloud …
- **Programming model & Interface:** High: Matlab/R, Medium: C/JAVA, Low: MPI
Outline: from sequential to parallel, algorithms and systems

- Optimization Algorithms
  - Algorithms:
    - Stochastic gradient descent
    - Coordinate descent
    - Proximal gradient methods: ISTA, FASTA, Smoothing proximal gradient
    - ADMM
  - Data-parallel
  - Model-Parallel

- Markov Chain Monte Carlo Algorithms
  - Data-parallel
    - Auxiliary Variable Dirichlet Process
    - Embarrassingly Parallel MCMC

- Distributed System Frameworks (aka, Big Learning systems)
Sparse Linear Regression

\[
\min_{\beta} \frac{1}{2} \| y - X\beta \|_2^2 + \lambda \Omega(\beta)
\]

Data fitting part:
- find \( \beta \) that fits into the data
- Squared loss, logistic loss, hinge loss, etc

Regularization part:
- induces sparsity in \( \beta \).
- incorporates structured information into the model
Sparse Linear Regression

\[
\min_{\beta} \frac{1}{2} \| y - X\beta \|_2^2 + \lambda \Omega(\beta)
\]

Examples of regularization \(\Omega(\beta)\):

\[
\Omega_{lasso}(\beta) = \sum_{j=1}^{J} | \beta_j |
\]

Sparsity

\[
\Omega_{group}(\beta) = \sum_{g \in G} \| \beta_g \|_2
\]

where \( \| \beta_g \|_2 = \sum_{j \in g} \sqrt{(\beta_j)^2} \)

Structured sparsity

\(\Omega_{tree}(\beta)\)

\(\Omega_{overlap}(\beta)\)

(sparsity + structured information)
Algorithm I: Stochastic Gradient Descent

- Consider an optimization problem:

\[
\min_x \mathbb{E}\{f(x, d)\}
\]

- Classical gradient descent:

\[
x^{(t+1)} \leftarrow x^{(t)} - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_x f(x^{(t)}, d_i)
\]

- Stochastic gradient descent:
  - Pick a random sample \(d_i\)
  - Update parameters based on noisy approximation of the true gradient

\[
x^{(t+1)} \leftarrow x^{(t)} - \gamma \nabla_x f(x^{(t)}, d_i)
\]
Stochastic Gradient Descent

- SGD converges almost surely to a global optimal for convex problems

- Traditional SGD compute gradients based on a single sample

- Mini-batch version computes gradients based on multiple samples
  - Reduce variance in gradients due to multiple samples
  - Multiple samples => represent as multiple vectors => use vector computation => speedup in computing gradients
Other usages:
e.g., SGD for Matrix Factorization

- Matrix factorization problem is given by

\[
\min_{W,H} \| A - WH^T \|_F^2 + \lambda \left( \|W\|_F^2 + \|H\|_F^2 \right)
\]

- MF approximates A with WH^T (W and H are rank-k matrices)
- SGD is shown be effective for MF [Koren and Bell, 2009]. MF SGD update rules are:

\[
\begin{align*}
    w_{i}^{(t+1)} & \leftarrow w_{i}^{(t)} - \gamma (\lambda w_{i}^{(t)} - R_{ij} h_{j}^{(t)}) \\
    h_{j}^{(t+1)} & \leftarrow h_{j}^{(t)} - \gamma (\lambda h_{j}^{(t)} - R_{ij} w_{i}^{(t)})
\end{align*}
\]

- Time complexity per MF SGD iteration is \(O(|\Omega|k)\)
  - Where \(\Omega\) is number of nonzero elements in matrix A
Parallel Stochastic Gradient Descent

- Parallel SGD: Partition data to different workers; all workers update full parameter vector

- Parallel SGD [Zinkevich et al., 2010]

- PSGD runs SGD on local copy of params in each machine
Hogwild!: Lock-free approach to PSGD

- MapReduce-like parallel processing frameworks have been a popular approach for parallel SGD

- However, MapReduce framework is not ideal for iterative algorithms
  - Difficult to express iterative algorithms in MapReduce
  - Overhead for fault tolerance
  - Overhead of locking or synchronization is a severe bottleneck

- Hogwild! Is a lock-free approach
  - It works well when data access is sparse, i.e., a single SGD step affects only a small number of variables
  - If multi processors write a parameter at the same time, break ties at random.
Hogwild!: Lock-free approach to PSGD

- Example:
  - Sparse SVM
    \[
    \min_{x} \sum_{\alpha \in E} \max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \|x\|_2^2
    \]
    - \( z \) is input vector, and \( y \) is a label; \((z,y)\) is an elements of \( E \)
    - Assume that \( z_\alpha \) are sparse
  - Matrix Completion
    \[
    \min_{W,H} \sum_{(u,v) \in E} (A_{uv} - W_u H_v^T)^2 + \lambda_1 \|W\|_F^2 + \lambda_2 \|H\|_F^2
    \]
    - Input \( A \) matrix is sparse
  - Graph cuts
    \[
    \min_{x} \sum_{(u,v) \in E} w_{uv} \|x_u - x_v\|_1 \text{ subject to } x_v \in S_D, v = 1, \ldots, n
    \]
    - \( W \) is a sparse similarity matrix, encoding a graph
Hogwild! Algorithm

- Hogwild! algorithm: iterate in parallel for each core
  - Sample $e$ uniformly at random from $E$
  - Read current parameter $x_e$; evaluate gradient of function $f_e$
  - Sample uniformly at random a coordinate $v$ from subset $e$
  - Perform SGD on coordinate $v$ with small constant step size

- Atomically update single coordinate, no mem-locking
- Hogwild! takes advantage of sparsity in ML problems
- Enables near-linear speedup on various ML problems
- Excellent on single machines, less ideal for distributed
  - Atomic update on multi-machine challenging to implement; inefficient and slow
  - Delay among machines requires explicit control… why? (see next slide)
The cost of uncontrolled delay – slower convergence

- Theorem: Given lipschitz objective $f_t$ and step size $\eta_t$,

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

where

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

$L$ is a lipschitz constant, and $\varepsilon_m$ and $\varepsilon_v$ are the mean and variance of the delay

- Intuition: distance between current estimate and optimal value decreases exponentially with more iters – but high variance in the delay $\varepsilon_v$ incurs exponential penalty

- Distributed systems have much higher delay variance than single machine
Theorem: the variance in the parameter estimate is

\[
\text{Var}_{t+1} = \text{Var}_t - 2\eta_t \text{cov}(x_t, \mathbb{E}^{\Delta t}[g_t]) + \mathcal{O}(\eta_t \xi_t) + \mathcal{O}(\eta_t^2 \rho_t^2) + \mathcal{O}_{\varepsilon_t}^*
\]

where

\[
\text{cov}(v_1, v_2) := \mathbb{E}[v_1^T v_2] - \mathbb{E}[v_1^T] \mathbb{E}[v_2]
\]

and \( \mathcal{O}_{\varepsilon_t}^* \) represents 5th order or higher terms as a function of the delay \( \varepsilon_t \).

Intuition: variance of the parameter estimate decreases near the optimum, but delay \( \varepsilon_t \) increases parameter variance \( \Rightarrow \) instability during convergence.

Distributed systems have much higher average delay than single machine.
PSGD with Parameter Server

- Parameter server allows us to parallelize SGD, consisting of:
  - Shared key-value store
  - Synchronization scheme

- Shared key-value store provides easy interface to read/write shared parameters

- Synchronization scheme determines how parameters are shared among multiple workers:
  - Bulk synchronous parallel (e.g., Hadoop)
  - Asynchronous parallel [Ahmed et al., 2012]
  - Stale synchronous parallel [Ho et al., 2013]
PSGD with Bounded Async PS

- Stale synchronous parallel supports synchronization with bounded staleness
- Fastest and the slowest workers are $\leq s$ clocks apart
Faster and better convergence

Objective function versus time
LDA 32 machines (256 threads), 10% data per iter

-9.00E+08
-9.50E+08
-1.00E+09
-1.05E+09
-1.10E+09
-1.15E+09
-1.20E+09
-1.25E+09
-1.30E+09

0 500 1000 1500 2000

Seconds

Log-Likelihood

BSP (stale 0)
stale 32
async
Algorithm II: Coordinate Descent

Update each regression coefficient in a cyclic manner

\[ \beta_1 \beta_2 \beta_3 \cdots \beta_J \]

1st iteration

2nd iteration

- **Pros and cons**
  - Unlike SGD, CD does not involve learning rate
  - If CD can be used for a model, it is often comparable to the state-of-the-art (e.g. lasso, group lasso)
  - However, as sample size increases, time for each iteration also increases
Example: Coordinate Descent for Lasso

\[
\hat{\beta} = \min_{\beta} \frac{1}{2} \| y - X\beta \|_2^2 + \lambda \sum_j |\beta_j |
\]

- Set a subgradient to zero:

\[-x_j^T (y - X\beta) + \lambda t_j = 0\]

- Assuming that \( x_j^T x_j = 1 \), we can derive update rule:

\[
\beta_j = S \left\{ x_j^T (y - \sum_{l \neq j} x_l \beta_l), \lambda \right\}
\]

\[S(x, \lambda) = \text{sign}(x)(|x| - \lambda)_+\]
Parallel Coordinate Descent

- Shotgun algorithm [Bradley et al. 2011] proposed parallel coordinate descent algorithm

- Shotgun algorithm
  - Choose parameters to update at random
  - Update the selected parameters in parallel
  - Iterate until convergence

- When features are nearly independent, Shotgun scales almost linearly
  - Shotgun scales linearly up to $P \leq \frac{d}{2\rho}$, where $\rho$ is the spectral radius of $A^T A$
  - For uncorrelated features, $\rho=1$; for exactly correlated features $\rho=d$
Block-greedy Coordinate Descent

- Block-greedy coordinate descent [Scherrer et al., 2012] extends Greedy-CD, Shortgun, Randomized-CD

- Alg: partition $p$ params into $B$ blocks; iterate:
  - Randomly select $P$ blocks
  - Greedily select one coordinate per $P$ blocks
  - Update each selected coordinate

- Sublinear convergence $O(1/k)$ for separable regularizer $r$:
  \[
  \min_x \sum_i f_i(x) + r(x_i)
  \]
  - Big-O constant depends on the maximal correlation among the $B$ blocks
- Hence greedily cluster features (blocks) to reduce correlation
Parallel Coordinate Descent with Dynamic Scheduler

- **STRADS (STRucture-Aware Dynamic Scheduler)** [Lee et al., 2014] is developed to schedule concurrent updates in CD
  - STRADS is a general scheduler for ML problems, applicable to CD as well as other ML algorithms such as Gibbs sampling

- STRADS improves the performance of CD, taking advantage of two key ideas
  - Dependency checking
    - update parameters which have a small degree of dependency. Thus, updating nearly independent parameters generate a small parallelization error
  - Priority-based updates
    - schedule the frequency of parameter updates based on their contributions to the decrease of objective function
Comparison: p-scheduling vs. u-scheduling

- Priority-based scheduling converged faster than the baseline with random scheduling

![Graph showing comparison between priority-based scheduling and other methods](image)

- Shotgun scheduling [Bradley et al. 2011]
- Priority-based scheduling + dep. checker

100M features
9 machines

Objective vs. Seconds
Advanced Optimization Tech.

- What if simple methods like SPG, CD are not adequate?

- Advanced techniques at hand
  - Complex regularizer: PG
  - Complex loss: SPG
  - Overlapping loss/regularizer: ADMM

- How to parallelize them? You must understand the MATH behind the algorithms
  - Which module should be at the server
  - Which module can be distributed to clients
  - ...
Proximal Gradient (a.k.a. forward-backward splitting, ISTA)

\[
\min_w f(w) + g(w)
\]

- \(f\): loss term, smooth (continuously differentiable)
- \(g\): regularizer, non-differentiable (e.g. 1-norm)

**Projected gradient**
- \(g\) represents some constraint
  
  \[
  g(w) = \iota_C(w) = \begin{cases} 
  0, & w \in C \\
  \infty, & \text{otherwise}
  \end{cases}
  \]

\[
\begin{align*}
  w & \leftarrow w - \eta \nabla f(w) \\
  w & \leftarrow \arg\min_z \frac{1}{2\eta} \|w - z\|^2 + \iota_C(z) \\
  &= \arg\min_{z \in C} \frac{1}{2} \|w - z\|^2
\end{align*}
\]

**Proximal gradient**
- \(g\) represents some simple function
  - e.g., 1-norm, constraint \(C\), etc.

\[
\begin{align*}
  w & \leftarrow w - \eta \nabla f(w) & \text{gradient} \\
  w & \leftarrow \arg\min_z \frac{1}{2\eta} \|w - z\|^2 + g(z) \\
  &= \arg\min_{z \in C} \frac{1}{2} \|w - z\|^2 & \text{proximal map}
\end{align*}
\]
Parallel (Accelerated) PG

- Bulk Synchronous Parallel Accelerated PG (exact)
  - Chen and Ozdaglar (2012, arXiv)
- Asynchronous Parallel (non-accelerated) PG (inexact)
  - Li et al. Parameter Server (2014, OSDI)
- General strategy:
  1. Compute gradients on workers
  2. Aggregate gradients on servers
  3. Compute proximal operator on servers
  4. Compute momentum on servers
  5. Send result $w^{t+1}$ to workers and repeat
- Can apply Hogwild-style asynchronous updates to non-accelerated PG, for empirical speedup
  - Open question: what about accelerated PG? What happens theoretically and empirically to accelerated momentum under asynchrony?
Outline: from sequential to parallel, algorithms and systems

- **Optimization Algorithms**
  - **Algorithms:**
    - Stochastic gradient descent
    - Coordinate descent
    - Proximal gradient methods: ISTA, FASTA, Smoothing proximal gradient
    - ADMM
  - Data-parallel
  - Model-Parallel

- **Markov Chain Monte Carlo Algorithms**
  - Data-parallel
    - Auxiliary Variable Dirichlet Process
    - Embarassingly Parallel MCMC

- **Distributed System Frameworks** (aka, Big Learning systems)
Posterior Inference Algorithms: MCMC and SVI

Markov Chain Monte Carlo: Randomly sample each variable in sequence

Stochastic Variational Inference: Gradient ascent on randomly-chosen variables
A Mixed Membership Triangular Model


Role mixed-membership vectors

\[ \theta_i \sim \text{Dirichlet}(\alpha) \]
\[ S_{i,j,k} \sim \text{Multinomial}(\theta_i) \]
\[ B_{xyz} \sim \text{Dirichlet}(\lambda) \]
\[ E_{ijk} \sim \text{TriangleDistribution}(B, S_{i,j,k}, S_{j,i,k}) \]

Rao-Blackwellized/Collapsed Gibbs Sampling for inference, with \( \theta \) and \( B \) integrated out

\[ p(s, \theta, B \mid E, \alpha, \lambda) \propto p(\theta \mid \alpha)p(B \mid \lambda)p(s \mid \theta) \}p(E \mid s, B). \]
Scalable Algorithms

- **Parsimonious model**: with linear $O(K)$ number of role parameters

- **$\delta$-subsampling**: down-sample neighborhood of high-degree nodes

- **Stochastic algorithms**: update small random subset of variables every iteration

- **More recent advancements of stochastic inference**:
  - Adaptive learning rate [R. Ranganath, C. Wang, D. Blei and E. P. Xing, ICML 2013]
  - Variance Reduction [C. Wang, X. Chen, A. Smola and E. P. Xing, NIPS 2013]
Gibbs Sampling (with $\delta$-subsampling) :

- Stanford web graph, $N \approx 280,000$
  - Converged in 500 Gibbs sampling iterations
  - Runtime: 18 hours using one processor core

Figure 5: $N = 281,903$ Stanford web graph, MMTM mixed-membership visualization.
SVI : Faster & More Accurate


### Real Networks — Statistics, Experimental Settings and Runtime

<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Edges</th>
<th>$\delta$</th>
<th>2,3-Tris (for $\delta$)</th>
<th>Frac. 3-Tris</th>
<th>Roles $K$</th>
<th>Threads</th>
<th>Runtime (10 data passes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightkite</td>
<td>58K</td>
<td>214K</td>
<td>50</td>
<td>3.5M</td>
<td>0.11</td>
<td>64</td>
<td>4</td>
<td>34 min</td>
</tr>
<tr>
<td>Brightkite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slashdot Feb 2009</td>
<td>82K</td>
<td>504K</td>
<td>50</td>
<td>9.0M</td>
<td>0.030</td>
<td>100</td>
<td>4</td>
<td>2.4 h</td>
</tr>
<tr>
<td>Slashdot Feb 2009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stanford Web</td>
<td>282K</td>
<td>2.0M</td>
<td>20</td>
<td>11.4M</td>
<td>0.57</td>
<td>5</td>
<td>4</td>
<td>10 min</td>
</tr>
<tr>
<td>Stanford Web</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Berkeley-Stanford Web</td>
<td>685K</td>
<td>6.6M</td>
<td>30</td>
<td>57.6M</td>
<td>0.55</td>
<td>100</td>
<td>8</td>
<td>15.2 h</td>
</tr>
<tr>
<td>Youtube</td>
<td>1.1M</td>
<td>3.0M</td>
<td>50</td>
<td>36.0M</td>
<td>0.053</td>
<td>100</td>
<td>8</td>
<td>9.1 h</td>
</tr>
</tbody>
</table>

**Stochastic VI MMSB (Gopalan et al, NIPS 2012)** took 8 days using 4 threads

*340x speedup!*

**Gibbs MMTM (Ho et al, NIPS 2012)** took 18.5 hours using 1 thread

*110x speedup!*

© Eric Xing @ CMU, 2015
The Need for Distributed Computation

- Triangular model SVI can handle 1M node networks with 100 roles in a few hours, on just one machine.

- What if we want to analyze 10K roles in a 100M-node network?

- Memory:
  - 100M * 10K = 1 trillion latent states = 4TB of RAM

- Computation:
  - SVI algorithm analyzes 1M nodes and 100 roles in a few hrs on one machine
  - 100M nodes and 10K roles would require 10K+ hrs on one machine, i.e. yrs!

- Need many machines to satisfy memory and computational requirements!
Parallel and Distributed MCMC

- Classic parallel MCMC solutions
  - Take multiple chains in parallel, take average/consensus between chains.
    - But what if each chain is very slow to converge?
    - Need full dataset on each process – no data parallelism!

- Naively run Gibbs sampling in parallel (i.e. parallelize a single MCMC chain)
  - Many distributed topic model implementations do this
  - But Parallel Gibbs sampling does not reach stationary distribution in general - it is incorrect! (Gonzalez et al. 2011 AISTATS)
  - Correct Parallel GS not possible on “collapsed” models like topic models … what to do?
Solution I: Induced Independence via Auxiliary Variables

[Dubey et al. ICML 2013, UAI 2014],

**Auxiliary Variable DP Inference**

- Conditioned on the restaurant allocation, data are distributed according to P independent Dirichlet process
- Each processor performs local collapsed Gibbs sampling on the independent DPs
- For the global parameters perform MH to migrate clusters across processors
  - Select a cluster ‘c’ and a processor ‘p’
  - Propose: move ‘c’ to ‘p’
  - Acceptance ratio depends on cluster size
- Can be done asynchronously in parallel without affecting performance
Auxiliary Variable Model for DP

- AV model (left) completely equivalent to standard DP (right)
  - Intuition: open up opportunity to parallelize MCMC via model reformulation

\[
D_j \sim \text{DP}\left(\frac{\alpha}{P}, H\right), \quad j = 1, \ldots, P
\]

\[
\phi \sim \text{Dirichlet}\left(\frac{\alpha}{P}, \ldots, \frac{\alpha}{P}\right)
\]

\[
\pi_i \sim \phi
\]

\[
\theta_i \sim D_{\pi_i}
\]

\[
x_i \sim f(\theta_i), \quad i = 1, \ldots, N.
\]
Correct Parallel MCMC via Auxiliary variable mixtures

- Idea: Dirichlet Mixture of Dirichlet processes are Dirichlet processes

\[ \phi \sim \text{Dirichlet} \left( \frac{\alpha}{P}, \ldots, \frac{\alpha}{P} \right) \]

\[ \pi_i \sim \phi \]
High-level idea:
- Run MCMC in parallel on data subsets; no communication between machines.
- Combine samples from machines to construct full posterior distribution samples.

Objective: recover full posterior distribution

\[ p(\theta | x^N) \propto p(\theta)p(x^N | \theta) = p(\theta) \prod_{i=1}^{N} p(x_i | \theta) \]

Definitions:
- Partition data into M subsets \( \{x^{n_1}, \ldots, x^{n_M}\} \)
- Define m-th machine’s “subposterior” to be \( p_m(\theta) \propto p(\theta) \frac{1}{M} p(x^{n_m} | \theta) \)
  - Subposterior: “The posterior given a subset of the observations with an underweighted prior”.

[Neiswanger, et al. UAI 14]
Embarassingly Parallel MCMC

- **Algorithm**
  1. For $m=1\ldots M$ independently in parallel, draw samples from each subposterior $p_m$.
  2. Estimate subposterior density product $p_1\cdots p_M(\theta) \propto p(\theta|x^N)$ (and thus the full posterior $p(\theta|x^N)$) by “combining subposterior samples”

- “Combine subposterior samples” via nonparametric estimation
  1. Given $T$ samples $\{\theta_{tm}^m\}_{t_m=1}^T$ from each subposterior $p_m$:
     - Construct Kernel Density Estimate (Gaussian kernel, bandwidth $h$):
       $\hat{p}_m(\theta) = \frac{1}{T} \sum_{t_m=1}^T \frac{1}{h^d} K \left( \frac{\|\theta - \theta_{tm}^m\|}{h} \right) = \frac{1}{T} \sum_{t_m=1}^T \mathcal{N}_d(\theta|\theta_{tm}^m, h^2 I_d)$
  2. Combine subposterior KDEs:
     \[
     \hat{p}_1 \cdots \hat{p}_M(\theta) = \frac{1}{T M} \prod_{m=1}^M \sum_{t_m=1}^T \mathcal{N}_d(\theta|\theta_{tm}^m, h^2 I_d) \propto \sum_{t_1=1}^T \cdots \sum_{t_M=1}^T w_{t_1} \cdots w_{t_M} \mathcal{N}_d\left( \theta|\bar{\theta}_{t_1} \cdots h^2 I_d \right)
     \]
     where
     \[
     \bar{\theta}_{t_1} = \frac{1}{M} \sum_{m=1}^M \theta_{tm}^m \quad w_{t_1} = \prod_{m=1}^M \mathcal{N}_d(\theta_{tm}^m|\bar{\theta}_{t_1}, h^2 I_d)
     \]
Embarassingly Parallel MCMC

- Theoretical guarantee: the nonparametric estimator generated by subposterior combination is consistent:

**Theorem 5.3.** If $h \leq T^{-1/(2\beta+d)}$, the mean-squared error of the estimator $\hat{p}_1\cdots\hat{p}_M(\theta)$ satisfies

$$\sup_{p_1,\ldots,p_M \in \mathcal{P}(\beta,L)} \mathbb{E} \left[ \int \left( \hat{p}_1\cdots\hat{p}_M(\theta) - p_1\cdots p_M(\theta) \right)^2 \, d\theta \right] \leq \frac{c}{T^{2\beta/(2\beta+d)}}$$

for some $c > 0$ and $0 < h \leq 1$.

- Simulations:
  - More subposteriors = tighter estimates
  - EPMCMC recovers correct parameter
  - Naïve subposterior averaging does not!

![Simulations Diagram](image-url)
Outline: from sequential to parallel, algorithms and systems

- Optimization Algorithms
  - Algorithms:
    - Stochastic gradient descent
    - Coordinate descent
    - Proximal gradient methods: ISTA, FASTA, Smoothing proximal gradient
    - ADMM
  - Data-parallel
  - Model-Parallel

- Markov Chain Monte Carlo Algorithms
  - Data-parallel
    - Auxiliary Variable Dirichlet Process
    - Embarrassingly Parallel MCMC

- Distributed System Frameworks (aka, Big Learning systems)
The systems interface of Big Learning

- Parallel Optimization and MCMC algorithms = “algorithmic interface” to Big Learning
  - Reusable building blocks to solve large-scale inferential challenges in Big Data and Big Models

- What about the systems (hardware, software platforms) to execute the algorithmic interface?
  - Hardware: CPU clusters, GPUs, Gigabit ethernet, Infiniband
    - Behavior nothing like single machine – what are the challenges?
  - Software platforms: Hadoop, Spark, GraphLab, Petuum
    - Each with their own “execution engine” and unique features
    - Different pros and cons for different data-, model-parallel styles of algorithms
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

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

---

**Compute vs Network**
- LDA 32 machines (256 cores)

- Network waiting time
- Compute time

Parallelize over worker threads
Share global model parameters via RAM
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

---

**Synchronization model** or **Programming model**

- **Non-uniform convergence**
- **Dynamic structures**
- **Error tolerance**
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()
}```

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 systems issues

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

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
The Big-ML “Stack” - More than just software

**Theory:** Degree of parallelism, convergence analysis, subsample complexity …

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

**Hardware:** GPU, flash storage, cloud …

**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
Parallelization Strategy

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

ML on epoch 1
ML on epoch 2
ML on epoch 3
ML on epoch m

Write outcome to KV store
Write outcome to KV store
Write outcome to KV store
Write outcome to KV store

Barrier?

Collect outcomes and aggregate
Do nothing
Do nothing
Do nothing

Network waiting time
Compute time

Compute vs Network
LDA 32 machines (256 cores)
Data Parallelism

\[ \Delta_1 = \Delta(A^{(t-1)}, D_1) \]

\[ \Delta_2 = \Delta(A^{(t-1)}, D_2) \]

\[ \Delta_3 = \Delta(A^{(t-1)}, D_3) \]

Additive Updates

\[ \Delta = \sum_{p=1}^{3} \Delta_p \]

\[ A^{(t)} = F(A^{(t-1)}, \Delta) \]
Model Parallelism

Read + Write

\[ \Delta_1 = \Delta_1(S_1 \in S, A^{(t-1)}, D) \]

\[ \Delta_p = \Delta_p(S_p \in S, A^{(t-1)}, D) \]

Concatenating updates

\[ \Delta = \{ \Delta_p \} \]

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

\[ S = S(A^{(t-1)}, D) \]

\[ S_1 \in S \]

\[ S_2 \in S \]

\[ S_3 \in S \]

\[ A^{(t-1)} \]

model parameters not updated in this iteration
A Dichotomy of Data and Model in ML Programs

Data Parallelism

\[ D_i \perp D_j \mid \theta, \forall i \neq j \]

Model Parallelism

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

Error persists and is not corrected

Sorting error: 2 after 5
ML Algorithms can Self-heal

Noisy trajectories

Initial True View

Optimum
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
- How do existing platforms (e.g., Spark, GraphLab) fit the above?
Why not Hadoop?

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!
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?

![GraphLab](image1)
![Spark](image2)
![PETUUM](image3)
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: 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

- **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: 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)
A New Framework for Large Scale Parallel Machine Learning
(Petuum.org)

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

- **Ready-to-run ML programs**
  - Earlier release: Topic Model (LDA), Deep Learning (DNN), Matrix Factorization (Collaborative Filtering), Lasso & Logistic Regression
  - Latest release: Random Forest, K-means, SVM, Deep Learning (CNN), Distance Metric Learning, Multiclass LR, Sparse Coding, Nonnegative MF, Topic Model (MedLDA)

- **Exploit ML properties, with theoretical guarantees**
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 **efficient** data-parallelism: model parameters become global
  - Special type of Distributed Shared Memory (DSM)

```
UpdateVar(i) {
  old = y[i]
  delta = f(old)
  y[i] += delta
}
```

```
UpdateVar(i) {
  old = PS.read(y,i)
  delta = f(old)
  PS.inc(y,i,delta)
}
```
Petuum Overview

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

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

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

```java
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.  
    ...  
}
```
## Lots of Advanced Apps

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN</td>
<td>Petuum Brain for mining images, videos, speech, text, biology</td>
</tr>
<tr>
<td>(Med)LDA</td>
<td>Web-scale analysis of docs, blogs, tweets</td>
</tr>
<tr>
<td>Regression</td>
<td>Linear and Logistic for intent prediction, stock/future hedging</td>
</tr>
<tr>
<td>(N)MF</td>
<td>Collaborative Filtering for recommending movies, products</td>
</tr>
<tr>
<td>MMTM</td>
<td>Societal/web-scale network analysis, community detection</td>
</tr>
<tr>
<td>SVM</td>
<td>General-purpose Classification</td>
</tr>
<tr>
<td>Ising</td>
<td>Model power and sensor grids</td>
</tr>
<tr>
<td>SIOR</td>
<td>Genome-wide association, stock/future hedging</td>
</tr>
<tr>
<td>ADMM</td>
<td>Constrained optimization for operations research, logistics management</td>
</tr>
<tr>
<td>Kalman</td>
<td>Kalman Filters for aviation control, dynamic system prediction</td>
</tr>
<tr>
<td>SC</td>
<td>Sparse Coding for web-scale, million-class classification</td>
</tr>
<tr>
<td>Metric</td>
<td>Distance Metric Learning to boost large-scale classification</td>
</tr>
</tbody>
</table>

© Eric Xing @ CMU, 2015
The Science Behind …

principles, design, and theory

- **Key insight**: ML algos have special properties
  - Error-tolerance, dependency structures, uneven convergence
  - How to harness for faster data/model-parallelism?
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**
  - 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
  - System Insight 1: Prioritize small # of vars => fewer deps to check
  - System Insight 2: Great synergy with Scheduler

![Diagram showing the process of prioritizing parameters and variables for update.](image)
How to speed up Data-Parallelism?

- Existing ways are either safe/slow (BSP), or fast/risky (Async)

- Need “Partial” synchronicity
  - Spread network comms evenly (don’t sync unless needed)
  - Threads usually shouldn’t wait – but mustn’t drift too far apart!

- Need straggler tolerance
  - Slow threads must somehow catch up

Is persistent memory really necessary for ML?
High-Performance Consistency Models for Fast Data-Parallelism


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

Thread 1 will always see these updates
Thread 1 may not see these updates (possible error)
Convergence Theorem

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

(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 \( \sigma = \frac{F}{L\sqrt{2(s+1)P}} \)

**SSP converges according to**

- Where \( T \) is the number of iterations

\[
R[X] := \left[ \frac{1}{T} \sum_{t=1}^{T} f_t(\tilde{x}_t) \right] - f(x^*) \leq 4FL \sqrt{\frac{2(s+1)P}{T}}
\]

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

Let observed staleness be $\gamma_t$

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

Theorem: Given L-Lipschitz objective $f_t$ and step size $h_t$,

Where $\gamma_t$ is the observed staleness, its mean, variance are $\mu_\gamma, \sigma_\gamma$.

$$P \left[ \frac{R[X]}{T} - \frac{1}{\sqrt{T}} \left( \eta L^2 + \frac{F^2}{\eta} + 2\eta L^2 \mu_\gamma \right) \geq \tau \right] \leq \exp \left\{ \frac{-T\tau^2}{2\bar{\eta}T\sigma_\gamma + \frac{2}{3}\eta L^2(2s+1)P\tau} \right\}$$

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

Explanation: the (E)SSP distance between true optima and current estimate decreases exponentially with more iterations. Lower staleness mean, variance $\mu_\gamma, \sigma_\gamma$ improve the convergence rate. Because ESSP has lower $\mu_\gamma, \sigma_\gamma$, it exhibits faster convergence than normal SSP.
Theorem: the variance in the (E)SSP estimate is

\[
\text{Var}_{t+1} = \text{Var}_t - 2\eta_t \text{cov}(x_t, E^\Delta_t [g_t]) + O(\eta_t \xi_t) + O(\eta_t^2 \rho_t^2) + O_{\gamma_t}^*
\]

where

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

and \(O_{\gamma_t}^*\) represents 5th order or higher terms in \(\gamma_t\).

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

Lower (E)SSP staleness \(\gamma_t\) \(\Rightarrow\) Lower variance in parameter \(\Rightarrow\) Less oscillation in parameter \(\Rightarrow\) More confidence in estimate quality and stopping criterion.

ESSP has lower staleness than SSP \(\Rightarrow\) higher quality estimates
Easy PS Programming

- Put global parameters in PS
  Examples:

- **Topic Modeling (MCMC)**
  - Topic-word table

- **Matrix Factorization (SGD)**
  - Factor matrices L, R

- **Lasso Regression (CD)**
  - Coefficients $\beta$

- PS supports many classes of algorithms
  - Above are just a few examples

```
UpdateVar(i) {
    old = PS.read(y,i)
    delta = f(old)
    PS.inc(y,i,delta)
}
```
Enjoys Async Speed, But BSP Guarantee across algorithms

- Massive Data Parallelism
- Effective across different algorithms

LDA on NYtimes Dataset
LDA 32 machines (256 cores), 10% discs per iter

Objective function versus time
Lasso 16 machines (128 threads)

Objective function versus time
MF 32 machines (256 threads)
Challenges in Model Parallelism

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

\[ y = X\beta \]

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

- Within group – synchronous (i.e., sequential) update
- Inter group – asynchronous update
Model Dependencies in Lasso

- Concurrent updates of $\beta$ may induce errors

Sequential updates

Concurrent updates

Induces parallelization error

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

$$\beta_1^{(t)} \leftarrow S(x_1^T y - x_1^T x_2 \beta_2^{(t-1)}, \lambda)$$
How to Model-Parallel?

- Again, existing ways are either safe but slow, or fast but risky
- Need to avoid processing whole-data just for optimal distribution
  - i.e., build expensive data representation on the whole data
  - Compute all variable dependencies
- Dynamic load balance

Is full consistency really necessary for ML?
Structure-Aware Parallelization (SAP)

- Smart model-parallel execution:
  - Structure-aware scheduling
  - Variable prioritization
  - Load-balancing

- Schedule()
- Push()
- Pull()

```c
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.
    ...
}
```
Structure-aware Dynamic Scheduler (STRADS)


- Priority Scheduling

\[ \{\beta_j\} \sim \left(\delta_\beta^{(t-1)}\right)^2 + \eta \]

- Block scheduling

[Block diagram showing worker synchronization and variable blocks]

[Kumar, Beutel, Ho and Xing, Fugue: Slow-worker agnostic distributed learning, AISTATS 2014]
Dynamic Scheduling Leads to Faster Convergence

Let $e := \frac{(P - 1)(\rho - 1)}{M} < 1$, where $P$ is the number of workers.

Let $M$ be the number of features.

Let $\rho$ be the spectral radius of $X$.

Theorem: the difference between the STRARD estimate and the true optima is

$$E[F(\beta^{(t)}) - F(\beta^*)] \leq \frac{CM}{P(1 - \epsilon)} \frac{1}{t} = O \left( \frac{1}{P \cdot t} \right)$$

Explanation: Dynamic scheduling ensures the gap between the objective at the $t$-th iteration and the optimal objective is bounded by $o \left( \frac{1}{P \cdot t} \right)$, which decreases as $t \rightarrow \infty$. Therefore dynamic scheduling ensures convergence.
Dynamic scheduling is close to ideal

Let $\mathcal{S}_{ideal}$ be an ideal model-parallel schedule
Let $\beta_{ideal}^{(t)}$ be the parameter trajectory by ideal schedule
Let $\beta_{dyn}^{(t)}$ be the parameter trajectory by dynamic schedule

**Theorem:** After $t$ iterations, we have

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

**Explanation:** Under dynamic scheduling, algorithmic progress is nearly as good as ideal model-parallelism. Intuitively, it is because both ideal and dynamic model-parallelism seek to minimize the parameter dependencies crossing between workers.
Faster, Better Convergence across algorithms

- STRADS+SAP achieves better speed and objective

---

**Objective**

100M features  
9 machines

- **STRADS**
- **Lasso−RR**

**RMSE**

80 ranks  
9 machines

- **STRADS**
- **GraphLab**

**Log−Likelihood**

2.5M vocab, 5K topics  
32 machines

- **STRADS**
- **YahooLDA**

---

© Eric Xing @ CMU, 2015
The Landscape of Big ML

- LDA - Topic Model
- MF - Matrix Factorization
- CNN - Convolutional Neural Network
- *GPU cores
The Landscape of Big ML

LDA - Topic Model
MF - Matrix Factorization
CNN - Convolutional Neural Network
*GPU cores

More cores, bigger models
The Landscape of Big ML
The Landscape of Big ML

LDA - Topic Model
MF - Matrix Factorization
CNN - Convolutional Neural Network
*GPU cores
The Landscape of Big ML

Bigger, more accurate models using reasonable # of cores
Open research topics

- Early days for data-, model-parallelism, and other ML properties
  - 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, partitioning, and scheduling strategies
  - Model of ML execution under error due to imperfect system?
  - Model not just “theoretical” ML costs, but also system costs?
Acknowledgements
Thank You!