A New Look at the System, Algorithm and Theory Foundations of Distributed Machine Learning

Eric P. Xing and Qirong Ho

1Carnegie Mellon University
2Institute for Infocomm Research, A*STAR

Acknowledgements:
Wei Dai, 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
Trees Falling in the Forest

"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 !!!
Massive Data

Facebook

1B+ USERS
30+ PETABYTES

Wikipedia

32 million pages

YouTube

100+ hours video uploaded every minute

twitter

645 million users
500 million tweets / day
The Scalability Challenge

![Graph showing the relationship between processing power/speed and the number of "machines". The graph compares "Perfect", "Pathetic", and "Good!" scenarios.]
An ML Program

\[
\arg\max_{\theta} \equiv \mathcal{L}(\{x_i, y_i\}_{i=1}^N; \theta) + \Omega(\theta)
\]

Solved by an iterative convergent algorithm

```plaintext
for (t = 1 to T) {
    doThings()
    \[
    \theta^{t+1} = g(\theta^t, \Delta_f \theta(D))
    \]
    doOtherThings()
}
```

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

Familiar problem: data from 50B devices, data centers won’t fit into memory of single machine
Maybe Big Data needs Big Models to extract understanding? But models with >1 trillion params also won’t fit!
Challenge 3 – Inadequate support for newer methods

Classic algorithms used for decades

- K-means
- Logistic regression
- Decision trees
- Naive Bayes
Growing Need for Big and Contemporary ML Programs

Google Brain
Deep Learning
for images:
1~10 Billion
model parameters

Multi-task Regression
for simplest whole-genome analysis:
100 million ~ 1 Billion
model parameters

Topic Models
for news article analysis:
Up to 1 Trillion
model parameters

Collaborative filtering
for Video recommendation:
1~10 Billion
model parameters
The Need for Distributed ML

- We had developed
  - a highly cost-effective model (MMTM [Ho et al., 2012]),
  - two generations of highly efficient algorithms
    (δ-subsampling Gibbs [Ho et al., 2012], SVI [Yin et al., 2013])
  - and highly specialized implementations

→ State-of-the-art results: 1M node networks with 100 roles in a few hours, on just one machine, 2-3 order’s of magnitudes speed-up

- But when we tried to do 10K roles in a 100M-node network:
  - Memory: 100M * 10K = 1 trillion latent states = 4TB of RAM
  - Computation: 10K+ hrs on one machine, i.e. yrs!
  - Attempt with Hadoop failed while in FB (see later) !!!

Say we want to analyze 10K roles in a 100M-node network, using a mixed membership model?
Many Open Questions:

- When is *Big Data* useful?
- Are *Big Models* useful?
  -- Both positive and negative answers exist …
- Inference algorithms, or inference systems?
- Theoretical guarantees, or empirical performance?
Current Solutions to Scalable ML

- Implementations of specific ML algorithms
  - YahooLDA, Vowpal Wabbit, Caffe, Torch, …
  - Provide a finely-tuned implementation of one (or a few) ML algorithms

- Platforms for general-purpose ML
  - Hadoop, Spark, GraphLab, Petuum, …
  - Allow others to write new ML programs

- Why this tutorial?
  - At first glance, ML problems seem radically different
  - We introduce a formal picture of ML to “bring order to the zoo”
  - We expose ML mathematical properties to be explored and later exploited
  - We note that many ML problems can be solved by a few “workhorse” algorithms
  - We explain how to design systems around these insights – thus achieving scalability, with both speed and solution quality guarantees
  - We provide theoretical guarantees for the system designs, and lay out roadmap for further analysis
Overview of Modern ML
A “Classification” of ML Models and Tools

- An ML program consists of:
  - A mathematical “ML model” (from one of many families)...
  - ... which is solved by an “ML algorithm” (from one of a few types)
A “Classification” of ML Models and Tools

- We can view ML programs as either
  - Probabilistic programs
  - Optimization programs

Probabilistic Programs

\[
\sum_{i=1}^{N} \sum_{j=1}^{N_i} \ln \mathbb{P}(x_{ij} \mid z_{ij}, B) + \sum_{i=1}^{N} \sum_{j=1}^{N_i} \ln \mathbb{P}(z_{ij} \mid \delta_i)
\]

Optimization Programs

\[
\sum_{i=1}^{N} \|y_i - X_i \beta\|_2^2 + \lambda \sum_{j=1}^{D} |\beta_j|
\]
Key building blocks of an ML program

- ML program: \( f(\theta, D) = L(\theta, D) + r(\theta) \)
- Objective or Loss function: \( L(\theta, D) \)
  - \( \theta = \) model, \( D = \) data
  - Common examples:
    - Least squares difference between predicted value and data
    - Log-likelihood of data
- Regularization / Prior / Structural Knowledge: \( r(\theta) \)
  - Common examples:
    - L2 regularization on \( \theta \) to prevent overfitting
    - L1 regularization on \( \theta \) to obtain sparse solution
    - (log of) Gaussian or Laplace priors over \( \theta \)
    - (log of) Dirichlet prior over \( \theta \) for smoothing

- Algorithm to solve for model given the data (cont’ next slide)
Iterative-convergent view of ML

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

New Model = Old Model + Update(Data)

- ML models solved via iterative-convergent ML algorithms
  - Iterative-convergent algorithms repeat until $\theta$ is stationary. Examples:
    - Probabilistic programs: MC, MCMC, Variational Inference
    - Optimization programs: Stochastic Gradient Descent, ADMM, Proximal Methods, Coordinate Descent
Optimization Example: Lasso Regression

- **Data, Model**
  - \( D = \{ \text{feature matrix } X, \text{ response vector } y \} \)
  - \( \theta = \{ \text{parameter vector } \beta \} \)

- **Objective** \( L(\theta, D) \)
  - Least-squares difference between \( y \) and \( X\beta \): \[ \sum_{i=1}^{N} \| y_i - X_i\beta \|_2^2 \]

- **Regularization** \( r(\theta) \)
  - L1 penalty on \( \beta \) to encourage sparsity: \[ \lambda \sum_{j=1}^{D} |\beta_j| \]
  - \( \lambda \) is a tuning parameter

- **Algorithms**
  - Coordinate Descent
  - Stochastic Proximal Gradient Descent
Optimization Example: Lasso Regression

Applications: Genetic Assays, Online Advertising

Model (Parameter Vector)  Update (CD algo)

\[
\beta_j^{(t)} = \beta_j^{(t-1)} - \beta_j^{(t-1)} + S(X_j^\top y - \sum_{k \neq j} X_j^\top X_k \beta_k^{(t-1)}, \lambda_n)
\]

\[
\hat{\theta}^{t+1} = \hat{\theta}^t + \Delta f \hat{\theta}(\mathcal{D})
\]

Data (Feature + Response Matrices)

\[
S(\cdot, \lambda) = \text{sign}(\cdot) (|\cdot| - \lambda)_+
\]
Probabilistic Example: Topic Models

- **Objective** $L(\theta, D)$
  - Log-likelihood of $D = \{\text{document words } x_{ij}\}$ given unknown $\theta = \{\text{document word topic indicators } z_{ij}, \text{ doc-topic distributions } \delta_i, \text{ topic-word distributions } B_k\}$:

$$
\sum_{i=1}^{N} \sum_{j=1}^{N_i} \ln \mathbb{P}_{\text{Categorical}}(x_{ij} \mid z_{ij}, B) + \sum_{i=1}^{N} \sum_{j=1}^{N_i} \ln \mathbb{P}_{\text{Categorical}}(z_{ij} \mid \delta_i)
$$

- **Prior** $r(\theta)$
  - Dirichlet prior on $\theta = \{\text{doc-topic, word-topic distributions}\}$

$$
\sum_{i=1}^{N} \ln \mathbb{P}_{\text{Dirichlet}}(\delta_i \mid \alpha) + \sum_{i=k}^{K} \ln \mathbb{P}_{\text{Dirichlet}}(B_k \mid \beta)
$$
  - $\alpha, \beta$ are “hyperparameters” that control the Dirichlet prior’s strength

- **Algorithm**
  - Collapsed Gibbs Sampling
Probabilistic Example: Topic Models

Applications: Natural Language Processing, Information Retrieval

Data (Docs) = x_{ij}

Model (Topics) = B_k

Update (Collapsed Gibbs sampling)

For each doc i, each token j:

Set \( k_{old} = z_{ij} \)

Gibbs sample new value of \( z_{ij} \), according to \( P(z_{ij} | x_{ij}, \delta_i, B) \)

Set \( k_{new} = z_{ij} \)

Perform updates to \( B, \delta: \)

\[
\begin{align*}
B_{k_{old}, w_{ij}} &= B_{k_{old}, w_{ij}} - 1 \\
B_{k_{new}, w_{ij}} &= B_{k_{new}, w_{ij}} + 1 \\
\delta_{i,k_{old}} &= \delta_{i,k_{old}} - 1 \\
\delta_{i,k_{new}} &= \delta_{i,k_{new}} + 1
\end{align*}
\]

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

```
1 6 7 3 5 4 8 2
1 6 3 7 4 5 2 8
1 3 6 7 4 5 2 8
1 3 4 5 6 7 2 8
```
... but ML Algorithms can Self-heal
More Intrinsic Properties of ML Programs

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

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

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

- Whereas traditional programs are **transaction-centric**, thus only guaranteed by **atomic correctness** at every step
Why come up with an ML classification?

- An ML classification helps to solve ML algorithm challenges systematically
  - No need to invent new algorithms for each new ML model or variant
  - Instead, re-use a smaller number of “workhorse” algorithms (engines) to solve entire classes of models
    - For each new ML model, determine which ML class it falls under
    - Then apply the most appropriate workhorse algorithm for that class

- Next tutorial section: Distributed ML Algorithms
  - We present a number of “workhorse” algorithms:
    - Basic form
    - Which units can be parallelized
    - What risks are incurred by parallelization (e.g. error or non-convergence)
    - Examples of scalable realizations (software)
Distributed ML Algorithms
An ML Program

\[
\arg \max_{\theta} \equiv \mathcal{L}(\{x_i, y_i\}_{i=1}^{N} ; \theta) + \Omega(\theta)
\]

Solved by an iterative convergent algorithm

```plaintext
for (t = 1 to T) {
    doThings()
    \[ \theta^{t+1} = g(\theta^t, \Delta f \theta(D)) \]
    doOtherThings()
}
```

This computation needs to be parallelized!
Challenge

- Optimization programs:

\[
\Delta \leftarrow \sum_{i=1}^{N} \left[ \frac{d}{d\theta_1}, \ldots, \frac{d}{d\theta_M} \right] f(x_i, y_i; \theta)
\]

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

A huge number of parameters (e.g.) \(J = 1B\)
Challenge

- Probabilistic programs

\[ z_{ij} \sim p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k) \cdot \frac{\beta_{x_{ij}} + B_{k,x_{ij}}}{V_\beta + \sum_{v=1}^{V} B_{k,v}} \]
Parallelization Strategies

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

New Model = Old Model + Update(Data)

Data Parallel
Parallelization Strategies

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

New Model = Old Model + Update(Data)

\[ \Delta \theta(D) \]

Data Parallel

\[ \Delta \theta(D_1), \Delta \theta(D_2), \Delta \theta(D_3), \ldots, \Delta \theta(D_n) \]

Model Parallel

\[ \Delta \theta_1(D), \Delta \theta_2(D), \Delta \theta_3(D), \ldots, \Delta \theta_k(D) \]
Outline: Optimization & MCMC Algorithms

- **Optimization Algorithms**
  - Stochastic gradient descent
  - Coordinate descent
  - Proximal gradient methods
    - ISTA, FASTA, Smoothing proximal gradient
  - ADMM

- **Markov Chain Monte Carlo Algorithms**
  - Auxiliary Variable methods
  - Embarrassingly Parallel MCMC
  - Parallel Gibbs Sampling
    - Data parallel
    - Model parallel
Example Optimization Program: 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
Example Optimization Program: Sparse Linear Regression

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

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

\[
\begin{align*}
\Omega_{lasso}(\beta) &= \sum_{j=1}^{J} |\beta_j| \\
\Omega_{group}(\beta) &= \sum_{g \in G} \|\beta_g\|_2 \\
\Omega_{tree}(\beta) &= \Omega_{overlap}(\beta)
\end{align*}
\]

Sparsity

Structured sparsity
(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
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 [Recht et al., 2011]

- Goal is to minimize a function in the form of

\[ f(x) = \sum_{e \in E} f_e(x_e) \]

- \( e \) denotes a small subset of parameter indices
- \( x_e \) denotes parameter values indexed by \( x_e \)

- Key observation:
  - Cost functions of many ML problems can be represented by \( f(x) \)
  - In SOME ML problems, \( f(x) \) is sparse. In other words, \(|E|\) and \( n \) are large but \( f_e \) is applied only a small number of parameters in \( x \)
Hogwild!: Lock-free approach to PSGD [Recht et al., 2011]

- **Example:**
  - **Sparse SVM**
    \[
    \min_x \sum_{\alpha \in E} \max(1 - y_\alpha x^T z_\alpha, 0) + \lambda \|x\|^2
    \]
    - \(z\) is input vector, and \(y\) is a label; \((z,y)\) is an element 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**
    \[
    \min_x \sum_{(u,v) \in E} w_{uv} \|x_u - x_v\|_1 \quad \text{subject to } x_v \in S_D, v = 1, \ldots, n
    \]
    - \(W\) is a sparse similarity matrix, encoding a graph
Hogwild! Algorithm [Recht et al., 2011]

- 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

- Advantages
  - Atomically update single coordinate, no mem-locking
  - Takes advantage of sparsity in ML problems
  - Near-linear speedup on various ML problems, on single machine

- Excellent on single machine, 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 [Dai et al. 2015]

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

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

- where $R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*)$
- Where $L$ is a lipschitz constant, and $\epsilon_m$ and $\epsilon_v$ are the mean and variance of the delay

- Intuition: distance between current estimate and optimal value decreases exponentially with more iterations
  - But high variance in the delay $\epsilon_v$ incurs exponential penalty!

- Distributed systems exhibit much higher delay variance, compared to single machine
The cost of uncontrolled delay – unstable convergence [Dai et al. 2015]

- Theorem: the variance in the parameter estimate is

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

- Where \(\text{cov}(v_1, v_2) := E[v_1^T v_2] - E[v_1^T]E[v_2]\)
- and \(O^{*}_{\epsilon_t}\) represents 5th order or higher terms, as a function of the delay \(\epsilon_t\)

- Intuition: variance of the parameter estimate decreases near the optimum
  - But delay \(\epsilon_t\) increases parameter variance \(\Rightarrow\) instability during convergence

- Distributed systems have much higher average delay, compared to single machine
Parallel SGD with Key-Value Stores

- We can parallelize SGD via
  - Distributed key-value store to share parameters
  - Synchronization scheme to synchronize parameters

- 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, Li et al., 2014]
  - Stale synchronous parallel [Ho et al., 2013, Dai et al., 2015]
Parallel SGD with Bounded Async KV-store

- Stale synchronous parallel (SSP) is a synchronization model with bounded staleness – “bounded async”
- Fastest and the slowest workers are $\leq s$ clocks apart
Example KV-Store Program: Lasso

- Lasso example: want to optimize
  \[ \sum_{i=1}^{N} \| y_i - X_i \beta \|^2 + \lambda \sum_{j=1}^{D} | \beta_j | \]

- Put \( \beta \) in KV-store to share among all workers

- Step 1: SGD: each worker draws subset of samples \( X_i \)
  - Compute gradient for each term \( \| y_i - X_i \beta \|^2 \) with respect to \( \beta \); update \( \beta \) with gradient
  \[ \beta(t) = \beta(t-1) + 2(y_i - X_i \beta^{(t-1)}) X_i^T \]

- Step 2: Proximal operator: perform soft thresholding on \( \beta \)
  \[ \beta_j = \text{sign}(\beta_j) (|\beta_j| - \lambda)_+ \]
  - Can be done at workers, or at the key-value store itself

- Bounded Asynchronous synchronization allows fast read/write to \( \beta \), even over slow or unreliable networks
Bounded Async KV-store: Faster and better convergence
Algorithm II: Coordinate Descent

Update each regression coefficient in a cyclic manner

1\text{st iteration}

\beta_1 \beta_2 \beta_3 \ldots \beta_J

2\text{nd iteration}

\beta_1 \beta_2 \beta_3 \ldots \beta_J

- 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

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

- Subgradient of our objective with respect to $\beta_j$ is:

$$-x_j^T (y - X\beta) + \lambda t_j$$

Subgradient of L1 norm:

$$\begin{cases} t_j = \text{sign}(\beta_j) & \text{if } \beta_j \neq 0 \\ t_j \in [-1,1] & \text{Otherwise} \end{cases}$$
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:

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

\[ S(x, \lambda) = \text{sign}(x)(|x| - \lambda)_+ \]
Example: Block Coordinate Descent for Group Lasso

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

- Set it to zero:

\[
-x_j^T (y - X\beta) + \lambda u_j = 0, \forall j \in g
\]

- In a similar fashion, we can derive update rule for group \( g \)

Iterate over each group of coefficients
Parallel Coordinate Descent

[Bradley et al. 2011]

- Shotgun, a parallel coordinate descent 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 \( P \leq \frac{d}{2\rho} \) workers, where \( \rho \) is spectral radius of \( A^TA \)
  - For uncorrelated features, \( \rho=1 \); for exactly correlated features \( \rho=d \)
  - No parallelism if features are exactly correlated!
Intuitions for Parallel Coordinate Descent

- Concurrent updates of parameters are useful when features are uncorrelated

- Updating parameters for correlated features may slow down convergence, or diverge parallel CD in the worst case
  - To avoid updates of parameters for correlated features, block-greedy CD has been proposed

Source: [Bradley et al., 2011]
Block-greedy Coordinate Descent

[Scherrer et al., 2012]

- Block-greedy coordinate descent generalizes various parallel CD strategies
  - e.g. Greedy-CD, Shotgun, 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
[Lee et al., 2014]

- STRADS (STRucture-Aware Dynamic Scheduler) allows scheduling of concurrent CD updates
  - STRADS is a general scheduler for ML problems
  - Applicable to CD, and other ML algorithms such as Gibbs sampling

- STRADS improves CD performance via
  - Dependency checking
    - Update parameters which are nearly independent => small parallelization error
  - Priority-based updates
    - More frequently update those parameters which decrease objective function faster
Example Scheduler Program: Lasso

- **Schedule step:**
  - **Prioritization:** choose next variables $\beta_j$ to update, with probability proportional to their historical rate of change
    $\Pr(\text{select } \beta_j) \sim (|\beta_j^{(t-1)} - \beta_j^{(t-2)}|)^2 + \epsilon$
  - **Dependency checking:** do not update $\beta_j$, $\beta_k$ in parallel if feature dimensions $j$ and $k$ are correlated
    $|\mathbf{x}_j^\top \mathbf{x}_k| < \rho$ for all $j \neq k$

- **Update step:**
  - For all $\beta_j$ chosen in Schedule step, in parallel, perform coordinate descent update
    $\beta_j^{(t)} = \beta_j^{(t-1)} - \beta_j^{(t-1)} + S(X_j^\top y - \sum_{k \neq j} X_j^\top X_k \beta_k^{(t-1)}, \lambda_n)$
  - Repeat from Schedule step
Comparison: priority vs. random-scheduling

- Priority-based scheduling converges faster than Shotgun (random) scheduling
Advanced Optimization Techniques

- 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? Must understand math behind algorithms
  - Which terms should be computed at server
  - Which terms can be distributed to clients
  - ...
When Constraints Are Complex:

-- Algorithm III: Proximal Gradient (a.k.a. ISTA)

\[
\min \limits_{w} f(w) + g(w)
\]

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

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

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

\[
\begin{align*}
\mathbf{w} &\leftarrow \mathbf{w} - \eta \nabla f(\mathbf{w}) \\
\mathbf{w} &\leftarrow \arg\min_{\mathbf{z}} \frac{1}{2\eta} \| \mathbf{w} - \mathbf{z}\|^2 + \iota_C(\mathbf{z}) \\
&= \arg\min_{\mathbf{z} \in C} \frac{1}{2} \| \mathbf{w} - \mathbf{z}\|^2
\end{align*}
\]
Algorithm III: Proximal Gradient (a.k.a. ISTA)

- PG hinges on the proximal map [Moreau, 1965]:
  \[ P^\eta_g(w) = \arg\min_z \frac{1}{2\eta} \|w - z\|^2 + g(z) \]
- Treated as black-box in PG
- Need proximal map efficiently computable, better closed-form
  - True when \( g \) is separable and “simple”, e.g. 1-norm (separable in each coordinate), non-overlapping group norm, etc.
- Can be demanding if \( g = g_1 + g_2 \), but vars in \( g_1, g_2 \) overlap
- [Yu, 2013] gave sufficient conditions for when \( g = g_1 + g_2 \) can be easily handled:
  \[ P^\eta_{g_1 + g_2}(w) = P^\eta_{g_1} \left( P^\eta_{g_2}(w) \right) \]
  - Useful when \( P^\eta_{g_1} \) and \( P^\eta_{g_2} \) available in closed-forms
  - E.g. fused lasso (Friedman et al.’07): \( P^\eta_{\| \cdot \|_1 + \| \cdot \|_{tv}}(w) = P^\eta_{\| \cdot \|_1} \left( P^\eta_{\| \cdot \|_{tv}}(w) \right) \)
Extensions of Proximal Gradient

- **Bregman**: replace the quadratic with bregman divergence
  \[ w^{t+1} \leftarrow \arg \min_w \langle w, \eta \nabla f(w^t) \rangle + \underbrace{D(w \| w^t)}_{d(w) - d(w^t) - \langle w - w^k, \nabla d(w^k) \rangle} + g(w) \]
  - \( d(w) = \frac{1}{2} \| w \|^2 \) recovers the usual PG; entropic: \( d(w) = \sum_i w_i \log w_i - w_i \)
  - Can be beneficial if \( d \) “aligns” well with \( g \) (e.g., leading to closed-form sol.)
  - Same theoretical guarantee (Tseng’10)

- **Subgradient**: replace grad with subgradient
  \[ w^{t+1} \leftarrow \arg \min_w \langle w, \eta_t \partial f(w^t) \rangle + D(w \| w^t) + g(w) \]
  - Removes differentiable assump. on \( f \)
  - Need diminishing step size \( \eta_t \to 0 \)
  - Slower \( O\left(\frac{1}{\sqrt{t}}\right) \) convergence (Duchi & Singer’09)
Accelerated PG (a.k.a. FISTA)

[Beck & Teboulle, 2009; Nesterov, 2013; Tseng, 2008]

- PG convergence rate $O(1/(\eta t))$
- Can be boosted to $O(1/(\eta t^2))$
  - Same Lipschitz gradient assumption on f; similar per-step complexity!
  - Lots of follow-up work to the papers cited above

**Proximal Gradient**

$$v^t \leftarrow w^t - \eta \nabla f(w^t)$$

$$u^t \leftarrow P_g^\eta(v^t)$$

$$w^{t+1} \leftarrow u^t + \left(0 \cdot (u^t - u^{t-1})\right)_{no \ momentum}$$

**Accelerated Proximal Gradient**

$$v^t \leftarrow w^t - \eta \nabla f(w^t)$$

$$u^t \leftarrow P_g^\eta(v^t)$$

$$w^{t+1} \leftarrow u^t + \frac{t - 1}{t + 2} (u^t - u^{t-1}) \approx 1 \ momentum$$

$$P_g^\eta(w) := \arg \min_z \frac{1}{2\eta} \|w - z\|^2_2 + g(z)$$
Parallel (Accelerated) PG

- Bulk Synchronous Parallel Accelerated PG (exact)
  - [Chen and Ozdaglar, 2012]
- Asynchronous Parallel (non-accelerated) PG (inexact)
  - [Li et al., 2014] Parameter Server

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?
When Objective Is Not Smooth:

-- Moreau Envelope Smoothing

- So far need \( f \) to have Lipschitz cont \( \text{grad} \), obtained \( O(1/t^2) \)
- What if not ?
- Can use subgradient, with diminishing step size \( O(1/\sqrt{t}) \)
  - Huge gap !
- Smoothing comes into rescue, if \( f \) itself is H-Lipschitz cont
  - Approx \( f \) with something nicer, like Taylor expansion in calculus 101
- Replace \( f \) with its Moreau envelope function

\[
M_f^n(w) := \min_z \frac{1}{2\eta} \|w - z\|^2_2 + f(z)
\]

Prop. \( \forall w \), \( 0 \leq f(w) - M_f^n(w) \leq \eta H^2/2 \)

- \( f(w) = |w| \), envelope \( M_f^n \) is Huber’s func (blue curve)
- Minimizer gives the proximal map \( P_f^n \) (red curve)
Smoothing Proximal Gradient

[Chen et al., 2012]

- Use Moreau envelope as smooth approximation
  - Rich and long history in convex analysis [Moreau, 1965; Attouch, 1984]
- Inspired by proximal point alg [Martinet, 1970; Rockafellar, 1976]
  - Proximal point alg = PG, when $f \equiv 0$
- Rediscovered in [Nesterov, 2005], led to SPG [Chen et al., 2012]

$$\min_w f(w) + g(w) \approx \min_w M_f^n(w) + g(w)$$

- With $\eta = O(1/t)$, SPG converges at $O(1/(\eta t^2)) = O(1/t)$
- Improves subgradient $O(1/\sqrt{t})$
- Requires both efficient $P_f^n$ and $P_g^n$

Smoothing Proximal Gradient

$$v^t \leftarrow w^t - \eta \nabla M_f^n(w^t)$$

$$u^t \leftarrow P_g^n(v^t)$$

$$w^{t+1} \leftarrow u^t + \frac{t-1}{t+2}(u^t - u^{t-1})$$

momentum
Parallel SPG?

- No known work yet
- Possible strategy:
  1. Compute smoothed gradients on workers
  2. Aggregate smoothed gradients on servers
  3. Compute proximal operator on servers
  4. Compute momentum on servers
  5. Send result $w^{t+1}$ to workers and repeat

- The above strategy is exact under Bulk Synchronous Parallel (just like accelerated PG).
  - Not clear how asynchronous updates impact smoothing+momentum
  - Open research topic
When Variables Are Coupled:

-- Algorithm IV: ADMM

Numerically challenging because

- Function f or g nonsmooth or constrained (i.e., can take value \( \infty \))
- Linear constraint couples the variables \( w \) and \( z \)
- Large scale, interior point methods NA

Naively alternating \( x \) and \( z \) does not work

- Min \( w^2 \) s.t. \( w + z = 1 \); optimum clearly is \( w = 0 \)
- Start with say \( w = 1 \) \( \rightarrow \) \( z = 0 \) \( \rightarrow \) \( w = 1 \) \( \rightarrow \) \( z = 0 \) …

However, without coupling, can solve separately \( w \) and \( z \)

- Idea: try to decouple vars in the constraint!

Canonical form:

\[
\min_{w,z} f(w) + g(z), \quad \text{s.t.} \quad Aw + Bz = c,
\]

where \( w \in \mathbb{R}^m, z \in \mathbb{R}^p, A : \mathbb{R}^m \to \mathbb{R}^q, B : \mathbb{R}^p \to \mathbb{R}^q, c \in \mathbb{R}^q \)
Example: Empirical Risk Minimization (ERM)

\[
\min_w g(w) + \sum_{i=1}^{n} f_i(w)
\]

- Each \( i \) corresponds to a training point \((x_i, y_i)\)
- Loss \( f_i \) measures the fitness of the model parameter \( w \)
  - least squares: \( f_i(w) = (y_i - w^T x_i)^2 \)
  - support vector machines: \( f_i(w) = (1 - y_i w^T x_i)_+ \)
  - boosting: \( f_i(w) = \exp(-y_i w^T x_i) \)
  - logistic regression: \( f_i(w) = \log(1 + \exp(-y_i w^T x_i)) \)
- \( g \) is the regularization function, e.g. \( \lambda_n \|w\|_2^2 \) or \( \lambda_n \|w\|_1 \)
- Vars coupled in obj, but not in constraint (none)
  - Reformulate: transfer coupling from obj to constraint
  - Arrive at canonical form, allow unified treatment later
How to: variable duplication

- Duplicate variables to achieve canonical form

\[
\min_w g(w) + \sum_{i=1}^{n} f_i(w)
\]

\[
v = [w_1, \ldots, w_n]^T
\]

\[
\min_{v,z} g(z) + \sum_i f_i(w_i), \quad \text{s.t.} \quad \begin{cases} w_i = z, \forall i \end{cases}
\]
\[
f(v) - [I,\ldots,I]^T z = 0
\]

- Global consensus constraint: \(\forall i, w_i = z\)
  - All \(w_i\) must (eventually) agree

- Downside: many extra variables, increase problem size
  - Implicitly maintain duplicated variables
Augmented Lagrangian

Canonical form: \( \min_{\mathbf{w}, \mathbf{z}} f(\mathbf{w}) + g(\mathbf{z}), \quad \text{s.t.} \quad A\mathbf{w} + B\mathbf{z} = \mathbf{c}, \)

where \( \mathbf{w} \in \mathbb{R}^m, \mathbf{z} \in \mathbb{R}^p, A : \mathbb{R}^m \rightarrow \mathbb{R}^q, B : \mathbb{R}^p \rightarrow \mathbb{R}^q, \mathbf{c} \in \mathbb{R}^q \)

- Intro Lagrangian multiplier \( \lambda \) to decouple variables

\[
\min_{\mathbf{w}, \mathbf{z}} \max_{\lambda} \left[ f(\mathbf{w}) + g(\mathbf{z}) + \lambda^\top (A\mathbf{w} + B\mathbf{z} - \mathbf{c}) + \frac{\mu}{2} \| A\mathbf{w} + B\mathbf{z} - \mathbf{c} \|_2^2 \right] \\
= L_\mu(\mathbf{w}, \mathbf{z}; \lambda)
\]

- \( L_\mu \): augmented Lagrangian
- More complicated min-max problem, but no coupling constraints
Algorithm IV: ADMM

\[
\min_{w,z} \max_{\lambda} \left( f(w) + g(z) + \lambda^\top (Aw + Bz - c) + \frac{\mu}{2} \| Aw + Bz - c \|^2 \right) \\
= L_\mu(w,z;\lambda)
\]

- Fix dual \( \lambda \), block coordinate descent on primal \( w, z \)
  \[
  w^{t+1} \leftarrow \arg \min_w L_\mu(w, z^t; \lambda^t) = f(w) + \frac{\mu}{2} \| Aw + Bz^t - c + \lambda^t / \mu \|^2
  \]
  \[
  z^{t+1} \leftarrow \arg \min_z L_\mu(w^{t+1}, z; \lambda^t) = g(z) + \frac{\mu}{2} \| Aw^{t+1} + Bz - c + \lambda^t / \mu \|^2
  \]
- Fix primal \( w, z \), gradient ascent on dual \( \lambda \)
  \[
  \lambda^{t+1} \leftarrow \lambda^t + \eta (Aw^{t+1} + Bz^{t+1} - c)
  \]
- Step size \( \eta \) can be large, e.g. \( \eta = \mu \)
  - Usually rescale \( \lambda \leftarrow \lambda / \eta \) to remove \( \eta \)
Row partition (data parallel)

\[ \min_z g(z) + \sum_{i=1}^{n} f_i(A_i z - c_i) \]

- each \( i \) corresponds to a (block of) training data \( A_i \)
- all summands \( f_i \) share the same global variable \( z \)
- all ERM in this form: SVM, lasso, logistic regression, etc.
- parallellize by duplicating \( z \) into \( w_1, \ldots, w_n \)

\[ \min_{w=[w_1, \ldots, w_n], z} g(z) + \sum_i f_i(A_i w_i - c_i), \quad \text{s.t.} \quad z - w_i = 0, \forall i \]

- **Exact Synchronization (bulk sync parallel) needed**
Column partition (model parallel)

\[
\min_{\mathbf{w}} f \left( \sum_{j=1}^{p} A_j \mathbf{w}_j - c \right) + \sum_{j=1}^{p} g_j(\mathbf{w}_j)
\]

- in columns data \( A = [A_1, \ldots, A_p] \), variables \( \mathbf{w} = [w_1, \ldots, w_p] \)
- Each function \( g_j \) have its own variable \( \mathbf{w}_j \)
- All variables \( \mathbf{w}_j \) coupled in \( f \)
- parallelize by adding auxiliary variable \( \mathbf{z} = [z_1, \ldots, z_p] \)

\[
\min_{\mathbf{w}, \mathbf{z}} f \left( \sum_{j} z_j - c \right) + \sum_{j} g_j(\mathbf{w}_j), \quad \text{s.t.} \quad A_j \mathbf{w}_j - z_j = 0, \forall j
\]

- Exact Synchronization (bulk sync parallel) needed
Asynchronous Parallel ADMM

[Zhang & Kwok, 2014]

- Only simplified consensus problem being studied:
  \[
  \min_{\mathbf{w}=[w_1,\ldots,w_n], \mathbf{z}} \sum_{i=1}^{n} f_i(w_i), \quad \text{s.t.} \quad w_i - z = 0, \forall i
  \]

- Can distribute the primal updates for each $w_i$
  \[
  (w_1, \ldots, w_n) \leftarrow \arg\min_{\mathbf{w}} L_\mu(w, z; \lambda)
  \]

- But dual update $\lambda \leftarrow \lambda + \sum_i w_i - z$ can happen only after all primal updates – barrier bottleneck

- How to alleviate the barrier bottleneck?
  - Asynchronously execute dual update after seeing $s$ out of $n$ primal updates
  - Condition: no machine is too far behind
    - Can be achieved with bounded staleness [Ho et al., 2013]
  - Asynchronous convergence proved in [Zhang & Kwok, 2014]
Outline: Optimization & MCMC Algorithms

- **Optimization Algorithms**
  - Stochastic gradient descent
  - Coordinate descent
  - Proximal gradient methods
    - ISTA, FASTA, Smoothing proximal gradient
  - ADMM

- **Markov Chain Monte Carlo Algorithms**
  - Auxiliary Variable methods
  - Embarrassingly Parallel MCMC
  - Parallel Gibbs Sampling
    - Data parallel
    - Model parallel
Example Probabilistic Program: Topic Models

- **Generative model**
  - Fit topics to each word $x_{ij}$ in each doc $i$
  - Uses categorical distributions with parameters $\delta$ and $B$

- **Parameter priors**
  - Induce sparsity in $\delta$ and $B$
  - Can also incorporate structure
    - E.g. asymmetric prior

Generative model of data

Priors on parameters
Inference for Probabilistic Programs: MCMC and SVI

Markov Chain Monte Carlo:
Randomly sample each variable in sequence
Next set of slides on this

Variational Inference:
Gradient ascent on variables
Can be treated as an optimization problem
Preliminaries: Speeding up sequential MCMC

- Technique 1: Alias tables
  - Sample from categorical distribution in amortized $O(1)$
  - “Throw darts at a dartboard”
  - Ex: probability distribution $[0.5, 0.25, 0.25]$
    - => alias table $\{1, 1, 2, 3\}$ => draw from table uniformly at random

- Technique 2: Cyclic Metropolis Hastings [Yuan et al., 2015]
  - Exploit Bayesian form $P(z=k) = P_{\text{evidence}}(k) \times P_{\text{prior}}(k)$
    - Propose $z_1$ from $P_{\text{evidence}}(k)$
    - Accept/Reject $z_1$
    - Propose $z_2$ from $P_{\text{prior}}(k)$
    - Accept/Reject $z_2$ … repeat
  - $P_{\text{prior}}(k), P_{\text{evidence}}(k)$ cheap to compute with alias table

- Other speedup techniques
  - Stochastic Gradient MCMC
  - Stochastic Variational Inference
Parallel and Distributed MCMC: Classic methods

- Classic parallel MCMC solution 1
  - 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!

![Diagram showing convergence of chains](chart.png)
Parallel and Distributed MCMC: Classic methods

- Classic parallel MCMC solution 2
  - Sequential Importance Sampling
  - Rewrite distribution over n variables as telescoping product over proposals q():
    \[ r(x_{1:n}) = r_1(x_1) \prod_{k=2}^{n} \alpha_k(x_{1:k}) \quad \text{where} \quad \alpha_n(x_{1:n}) = \frac{P_n'(x_{1:n})}{P_{n-1}(x_{1:n-1})q_n(x_n|x_{1:n-1})} \]
  - SIS algorithm:
    - Parallel draw samples \( x_i^n \sim q_n(x_n|x_{1:n-1}) \)
    - Parallel compute unnorm. wgts. \( r_n^i = r_{n-1}^i \alpha_n(x_{1:n}^i) = r_{n-1}^i \frac{P_n'(x_{1:n}^i)}{P_{n-1}(x_{1:n-1}^i)q_n(x_n^i|x_{1:n-1}^i)} \)
    - Compute normalized weights \( w_n^i \) by normalizing \( r_n^i \)
  - Drawback: variance of SIS samples increases exponentially with n
    - Need resampling + take many chains to control variance

- Let us look at newer solutions to parallel MCMC…
Solution I: Induced Independence via Auxiliary Variables [Dubey et al. 2013, 2014]

- Auxiliary Variable Inference: reformulate model as \( P \) independent models
  - Example below: Dirichlet Process for mixture models
  - Also applies to Hierarchical Dirichlet Process for topic models

- AV model (left) equivalent to standard DP model (right)

\[
\begin{align*}
D_j & \sim DP\left(\frac{\alpha}{P}, H\right), \quad j = 1, \ldots, P \\
\phi & \sim \text{Dirichlet}\left(\frac{\alpha}{P}, \ldots, \frac{\alpha}{P}\right) \\
\pi_i & \sim \phi \\
\theta_i & \sim D_{\pi_i} \\
x_i & \sim f(\theta_i), \quad i = 1, \ldots, N.
\end{align*}
\]
Solution I: Induced Independence via Auxiliary Variables [Dubey et al., 2013, 2014]

- Why does it work? A mixture over Dirichlet processes is equivalent to a Dirichlet processes

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

\[
\pi_i \sim \phi
\]
Solution I: Induced Independence via Auxiliary Variables [Dubey et al., 2013, 2014]

- Parallel inference algorithm:
  - Initialization: assign data randomly across $P$ Dirichlet Processes; assign each Dirichlet Process to one worker $p=1..P$
  - Repeat until convergence:
    - Each worker performs Gibbs sampling on local data within its DP
    - Each worker swaps its DP’s clusters with other workers, via Metropolis-Hastings:
      - For each cluster $c$, propose a new DP $q=1..P$
      - Compute proposal probability of $c$ moving to $p$
      - Acceptance ratio depends on cluster size

- Can be done asynchronously in parallel without affecting performance
Solution II: Embarrassingly Parallel (but correct) MCMC [Neiswanger et al., 2014]

- **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”.

\[ \text{IJCAI 15} \] © Eric Xing @ CMU, 2015
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:
   \[
   \hat{p}_1 \cdots \hat{p}_M(\theta) = \hat{p}_1 \cdots \hat{p}_M(\theta) = \frac{1}{T^M} \prod_{m=1}^M \sum_{t_m=1}^T \mathcal{N}_d(\theta | \theta_{tm}^m, h^2 I_d) \propto \sum_{t_1=1}^T \cdots \sum_{t_M=1}^T w_{t_1} \cdots w_{t_M} \mathcal{N}_d \left( \theta | \bar{\theta}_{t_1} + \frac{h^2}{M} I_d \right)
   \]
   - where
     \[
     \bar{\theta}_{t} = \frac{1}{M} \sum_{m=1}^M \theta_{tm}^m \quad w_{t} = \prod_{m=1}^M \mathcal{N}_d \left( \theta_{tm}^m | \bar{\theta}_{t} + \frac{h^2}{M} I_d \right)
     \]
Embarassingly Parallel MCMC

- Simulations:
  - More subposteriors = tighter estimates
  - EPMCMC recovers correct parameter
  - Naïve subposterior averaging does not!
Solution III: Parallel Gibbs Sampling

- Many MCMC algorithms
  - Sequential Monte Carlo [Canini et al., 2009]
  - Hybrid VB-Gibbs [Mimno et al., 2012]
  - Langevin Monte Carlo [Patterson et al., 2013]
  - ...

- Common choice in tech/internet industry:
  - Collapsed Gibbs sampling [Griffiths and Steyvers, 2004]
  - e.g. topic model Collapsed Gibbs sampler:

\[
p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k) \cdot \frac{\beta x_{ij} + B_{k,x_{ij}}}{V \beta + \sum_{v=1}^{V} B_{k,v}}
\]
Properties of Collapsed Gibbs Sampling (CGS)

- Simple equation: easy for system engineers to scale up
- Good theoretical properties
  - Rao-Blackwell theorem guarantees CGS sampler has lower variance (better stability) than naïve Gibbs sampling
- Empirically robust
  - Errors in $\delta, B$ do not affect final stationary distribution by much
- Updates are sparse: fewer parameters to send over network
- Model parameters $\delta, B$ are sparse: less memory used
  - If it were dense, even 1M word * 10K topic $\approx$ 40GB already!
CGS Example: Topic Model sampler

\[ p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k) \cdot \frac{\beta_{x_{ij}} + B_{k,x_{ij}}}{V \beta + \sum_{v=1}^{V} B_{k,v}} \]
Data Parallelization for CGS Topic Model Sampler

\[
p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k) \cdot \frac{\beta_{x_{ij}} + B_{k, x_{ij}}}{V \beta + \sum_{v=1}^{V} B_{k, v}}
\]
Data-Parallel Strategy 1: Approx. Distributed LDA

[Newman et al., 2009]

- Step 1: broadcast central model
Data-Parallel Strategy 1: Approx. Distributed LDA

[Newman et al., 2009]

- Step 1: broadcast central model
Data-Parallel Strategy 1: Approx. Distributed LDA

[Norman et al., 2009]

- Step 2: Perform Gibbs sampling in parallel
Data-Parallel Strategy 1: Approx. Distributed LDA

[Newman et al., 2009]

- Step 3: commit changes back to the central model
Data-Parallel Strategy 1: Approx. Distributed LDA

[Newman et al., 2009]

- Approximate
  - Convergence not guaranteed – Markov Chain ergodicity broken
  - Results generally “good enough” for industrial use

- Bulk synchronous parallel
  - CPU cycles are wasted while synchronizing the model

- How to overlap communication and computation for better efficiency?
Data-Parallel Strategy 2: Asynchronous LDA

[Smola et al., 2010; Ahmed et al., 2012]

- Also known as YahooLDA!
- Synchronize even while sampling is going on
Data-Parallel Strategy 2: Asynchronous LDA

[Smola et al., 2010; Ahmed et al., 2012]

- Multiple servers to share load
Data-Parallel Strategy 2: Asynchronous LDA

[Smola et al., 2010; Ahmed et al., 2012]

- Every machine keeps a local copy, updated asynchronously

Key-Value store

Model replica for sync

Model replica for sampling
Data-Parallel Strategy 2: Asynchronous LDA

[Smola et al., 2010; Ahmed et al., 2012]

- Asynchronous communication
  - Overlaps computation and communication – iterations are faster
  - But still approximate

- Also need to keep local copy of model
  - What if larger than machine capacity?
Data-Parallel Strategy 3: Petuum LDA v1 [Dai et al., 2015]

- Bounded-async protocol (SSP) + Least Recently Used Cache

Key-Value store

SSP protocol

LRU cache of the model
Data-Parallel Strategy 3: Petuum LDA v1 [Dai et al., 2015]

- Use of LRU and SSP protocol saves memory while retaining consistency
- But can we do better? Is the access pattern predictable?
- Recall the sampling equation

\[
p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k) \cdot \frac{\beta x_{ij} + B_{k,x_{ij}}}{V \beta + \sum_{v=1}^{V} B_{k,v}}
\]

- Not all of them need to be in memory throughout
- Can we schedule the sampling order?
Model-Parallel Strategy 1: GraphLab LDA

[Low et al., 2010; Gonzalez et al., 2012]

- Think graphically: token = edge

\[ p(z_{ij} = k | x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k) \cdot \frac{\beta x_{ij} + B_{k,x_{ij}}}{V\beta + \sum_{v=1}^{V} B_{k,v}} \]
Model-Parallel Strategy 1: GraphLab LDA

- Model-parallel via graph structure

[Low et al., 2010; Gonzalez et al., 2012]
Model-Parallel Strategy 1: GraphLab LDA

- Asynchronous communication
  - Overlaps computation and communication – iterations are faster
- Model-parallelism reduces error compared to data-parallelism
  - Disjoint words and docs on each machine => nearly-independent sampling
  - Exception: summary term \( (n^{-d_i}_k + \beta V) \) is inexact
- Model-parallelism means each machine only stores a subset of statistics
- Drawback: need to convert problem into a graph
  - Vertex-cut duplicates lots of vertices, canceling out savings
- Are there other ways to partition the problem?
Model-Parallel Strategy 2: LightLDA (Petuum LDA v2) [Yuan et al., 2015]

- Topic model matrix structure:

- Idea: non-overlapping matrix partition:

Source: [Gemulla et al., 2011]
Model-Parallel Strategy 2: LightLDA (Petuum LDA v2)

[Yuan et al., 2015]

- Non-overlapping partition of the word count matrix
- Fix data at machines, send model to machines as needed

Source: [Gemulla et al., 2011]
Model-Parallel Strategy 2: LightLDA (Petuum LDA v2) [Yuan et al., 2015]

- During preprocessing: determine set of words used in each data block
- Begin training: load each data block from disk
Model-Parallel Strategy 2: LightLDA (Petuum LDA v2) [Yuan et al., 2015]

- Pull the set of words from Key-Value store
Model-Parallel Strategy 2: LightLDA (Petuum LDA v2)

[Yuan et al., 2015]

- Sample and write the result to disk

![Diagram showing sequential read and write process]
Model-Parallel Strategy 2: LightLDA (Petuum LDA v2)

[Yuan et al., 2015]

- Disjoint words and docs on each machine
  - Gibbs sampling almost equivalent to sequential case
  - More accurate than data-parallel LDA
  - Fast, asynchronous execution possible

- Gibbs probability distributions very close to sequential case
  - Exception: summary term \( \left( n_k^{-d_l} + \bar{\beta} V \right) \) is slightly different on each machine
  - Mitigating factor: summary term very large for Big Data (typical size is >billions)
    - Differences in summary terms have small impact on Gibbs probability distributions
Model-Parallel Strategy 3: STRADS LDA (Petuum LDA v3)
[Lee et al., 2014]

- General-purpose system for “sending model to data” – scheduled model-parallelism
- Programmer writes code for data input, computation, and model output
- Communication and scheduling handled by the system

```c
func (input) {
    /* compute */
    ....
    return
    output;
}
```
Distributed ML Algorithms

Summary

- Many parallel algorithms for both Optimization and MCMC
- They share common parallelization themes
  - **Embarrassingly parallel**: combine results from multiple independent problems, e.g. PSGD, EP-MCMC
  - **Stochastic over data**: approximate functions/ gradients with expectation over subset of data, then parallelize over data subsets, e.g. SGD
  - **Model-parallel**: parallelize over model variables, e.g. Coordinate Descent
  - **Auxiliary variables**: decompose problem by decoupling dependent variables, e.g. ADMM, Auxiliary Variable MCMC

- Considerations
  - **Regularizers, model structure**: may need sequential proximal or projection step, e.g. Stochastic Proximal Gradient
  - **Data partitioning**: for data-parallel, how to split data over machines?
  - **Model partitioning**: for model-parallel, how to split model over machines? Need to be careful as model variables are not necessarily independent of each other.
Implementing Distributed ML Algorithms

- Implementing high-performance distributed ML is not easy
- If not careful, can end up slower than single machine!
  - System bottlenecks (load imbalance, network bandwidth & latency) are not trivial to engineer around

- Even if algorithm is theoretically sound and has attractive properties, still need to pay attention to system aspects
  - Bandwidth (communication volume limits)
  - Latency (communication timing limits)
  - Data and Model partitioning (machine memory limitation, also affects comms volume)
  - Data and Model scheduling (affects convergence rate, comms volume & timing)
  - Non-ideal systems behavior: uneven machine performance, other cluster users
Implementing Distributed ML Algorithms

- A number of ad-hoc or partial solutions, but sometimes lacking theoretical analysis
  - **Major barrier:** hard to analyze solutions because algorithm/systems sometimes not fully/transparently described in papers
  - **Possible solution:** a universal language and principles for design could facilitate theoretical analysis of existing and new solutions

- Let us look at some open-source platforms, which distributed ML algorithms can be implemented upon
Open-Source Platforms for Distributed ML
Modern Systems for Big ML

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

[Logos for GraphLab, Spark, and PETUUM]
Spark Overview [Zaharia et al., 2010]

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

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

MLlib algorithms (v1.4)

- Classification and regression
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression
- Collaborative filtering
  - alternating least squares (ALS)
- Clustering
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - streaming k-means
- Dimensionality reduction
  - singular value decomposition (SVD)
  - principal component analysis (PCA)
Spark Overview [Zaharia et al., 2010]

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

Source: Zaharia et al. (2012)
Spark Overview [Zaharia et al., 2010]

- RDD-based programming model
  - Similar in spirit to Hadoop Mapreduce
  - Functional style: manipulate RDDs via “transformations”, “actions”
    - E.g. map is a transformation, reduce is an action
  - Example: load file, count total number of characters
    ```scala
    val lines = sc.textFile("data.txt")
    val lineLengths = lines.map(s => s.length)
    val totalLength = lineLengths.reduce((a, b) => a + b)
    ```
  - Other transformations and actions:
    - union(), intersection(), distinct()
    - count(), first(), take(), foreach()
    - ...
  - Can specify if an RDD should be “persisted” to disk
    - Allows for faster recovery during cluster faults
Spark Overview [Zaharia et al., 2010]

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

Source: Zaharia et al. (2012)
Spark: Faster MapR on Data-Parallel

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

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

Source: ebaytechblog.com
Spark: Theoretical Considerations

- RDDs can be used to implement Bulk Synchronous programs
  - e.g. Map-Reduce programs

- No specific theory required
  - If a parallel algorithm is proven correct under synchronous execution, it will also be correct under Spark execution
GraphLab Overview [Low et al., 2012]

- Known as “GraphLab PowerGraph v2.2”
  - Different from commercial software “GraphLab Create” by Dato.com, who formerly developed PowerGraph v2.2

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

- Comes with ready-to-run “toolkits”
  - ML-centric toolkits: clustering, collaborative filtering, topic modeling, graphical models
GraphLab Overview [Low et al., 2012]

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

- Other toolkits available for computer vision, graph analytics, linear systems
GraphLab Overview [Low et al., 2012]

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

Source: Gonzalez (2012)
GraphLab Overview [Low et al., 2012]

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

Source: Gonzalez (2012)
GraphLab Overview [Low et al., 2012]

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

Source: Gonzalez (2012)
GraphLab Overview [Low et al., 2012]

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

Source: Gonzalez (2012)
GraphLab Overview [Low et al., 2012]

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

```cpp
// gather_nbrs: IN_NBRS
gather(D_u, D_{(u,v)}, D_v):
    return D_v.rank / #outNbrs(v)
sum(a, b): return a + b

apply(D_u, acc):
    rnew = 0.15 + 0.85 * acc
    D_u.delta = (rnew - D_u.rank) / #outNbrs(u)
    D_u.rank = rnew

// scatter_nbrs: OUT_NBRS
scatter(D_u, D_{(u,v)}, D_v):
    if (|D_u.delta|>\epsilon) Activate(v)
    return delta
```

Source: Gonzalez et al. (OSDI 2012)
Benefits of GraphLab

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

Source: Gonzalez (2012)
GraphLab: Model-Parallel via Graphs

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

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

Source: Low et al. (2010)
GraphLab: Theoretical Considerations

- GraphLab is an asynchronous system
  - Graph-consistency models used to enforce desirable graph-theoretic properties

- Using “Full Consistency”, Gibbs sampling is provably correct
  [Gonzalez et al., 2011]

**Proposition 3.1 (Graph Coloring and Parallel Execution).** Given $p$ processors and a $k$-coloring of an $n$-variable MRF, the parallel Chromatic sampler is ergodic and generates a new joint sample in running time:

$$O\left(\frac{n}{p} + k\right).$$

- No known results for Edge/Vertex Consistency
A New Framework for Large Scale Parallel Machine Learning
(Petuum.org)
Petuum Overview [Xing et al., 2015]

- **Key modules**
  - **Key-value store** (Parameter Server) for data-parallel ML algos
  - **Scheduler** for model-parallel ML algos

- **Program ML algos in iterative-convergent style**
  - ML algo = (1) write update equations + (2) iterate eqns via schedule
Petuum Overview [Xing et al., 2015]

- ML Library (Petuum v1.1):
  - Topic Modeling
    - LDA
    - MedLDA (supervised topic models)
  - Deep Learning
    - Fully-connected DNN
    - Convolutional Neural Network
  - Matrix Factorization
    - Least-squares Collaborative Filtering (with regularization)
    - Non-negative Matrix Factorization
    - Sparse Coding
  - Regression
    - Lasso Regression
  - Metric Learning
    - Distance Metric Learning
  - Clustering
    - K-means
  - Classification
    - Random Forest
    - Logistic Regression and SVM
    - Multi-class Logistic Regression
Petuum Overview [Xing et al., 2015]

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

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

```
ProcessDataPoint(i) {
  for j = 1 to M {
    old = PS.read(model, j)
    delta = f(model, data(i))
    PS.inc(model, j, delta)
  }
}
```
Petuum Overview [Xing et al., 2015]

- Key-Value store features:
  - ML-tailored consistency model: Stale Synchronous Parallel (SSP)
  - Asynchronous-like speed
  - Bulk Synchronous Parallel-like correctness guarantees for ML
Petuum Overview [Xing et al., 2015]

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

```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.
    ...
}
```
Petuum Overview [Xing et al., 2015]

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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Variables</th>
<th>Input Data</th>
<th>Finding dynamic block structures</th>
<th>Re-grouping</th>
<th>Dispatching blocks to workers</th>
<th>Dynamically revising</th>
</tr>
</thead>
</table>

![Graph showing objective vs seconds with a sharp drop due to SAP](chart.png)
Petuum: ML props = 1st-class citizen

- Error tolerance via Stale Sync Parallel KV-store
  - System Insight 1: ML algos bottleneck on network comms
  - System Insight 2: More caching => less comms => faster execution
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: Lowers computational cost of Scheduling
Petuum Architecture and Hadoop Ecosystem Integration

**Hadoop Ecosystem**
- Spark
- Hadoop MapReduce
- HBase
- Hive
- and others …

**Petuum Architecture**
- ML application library
- Data-Parallel API
- Model-Parallel API
- Bounded-Async KV-store (Bösen)
- Dynamic Scheduler (Strads)

**Hadoop Ecosystem**
- YARN (resource manager, fault tolerance)
- HDFS (distributed storage)
ML Programming Interface: Needs and Considerations

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

- ML programs are “stateful”
  - Model state $\theta$ updated every iteration; auxiliary local variables (e.g. summary statistics) often needed at each parallel worker
  - Natural fit for imperative programming style, coupled with distributed shared memory that automatically synchronizes model state via a consistency model
  - **Map-Reduce**: communicating model state through Map-reduce API can be expensive for Big Models; may require external distributed shared memory support (e.g. Cassandra, Memcached)
  - **Message-passing (e.g. MPI)**: efficient, but requires user to explicitly decide when to communicate updates
ML Programming Interface: Needs and Considerations

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

- ML programs can require explicit scheduling, e.g. model-parallel.
  - Programming interface should **separate update functions from schedule functions**.
  - GraphLab, Spark, Hadoop perform scheduling according to their own criteria; user-defined scheduling not currently available but could be implemented.
ML Programming Interface: Needs and Considerations

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

- ML shown to be efficient under non-blocking, bounded-asynchronous communication:
  - Distributed shared memory system (e.g. KV-store, parameter server) handles all communication.
  - **Ideal**: read/write model $\theta$ without worrying about communication; program correctness assured by bounded-async theoretical guarantees.
  - **Open question**: possible to adapt GraphLab (graph-async) and Hadoop/Spark (bulk synchronous) to bounded-async execution?
ML Programming Interface: Needs and Considerations

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

- What can be abstracted away?
  - Abstract away inter-worker communication/synchronization:
    - Automatic consistency models; bandwidth management through distributed shared memory
  - Abstract scheduling away from update equations:
    - Easy to change scheduling strategy, or use dynamic schedules
  - Abstract away worker management:
    - Let ML system decide optimal number and configuration of workers
  - Ideally, reduce programmer burden to just 3 things:
    - Declare model, write updates, write schedule
Systems, Architectures for Distributed ML
There Is No Ideal Distributed System!

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

**Unequal performance**

**Low bandwidth, High delay**

**Async execution: May diverge**

**BSP execution: Long sync time**
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. 2015, 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?
The systems interface of Big Learning

- Parallel Optimization and MC 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()
    parellelUpdate(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

---

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

- Non-uniform convergence
- Dynamic structures
- Error tolerance

---

**Synchronization model**

- or

**Programming model**
Why need new Big ML systems?

MLer’s view

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

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

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

Systems View:

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

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

Oversimplify ML issues and/or ignore ML opportunities
- ML algos “just work” without proof
- Conversion of ML algos across different program models (graph programs, RDD) is easy
Solution:

Machine Learning Models/Algorithms

- Graphical Models
- Nonparametric Bayesian Models
- Regularized Bayesian Methods
- Large-Margin Sparse Structured I/O Regression
- Sparse Coding
- Spectral/Matrix Methods
- Deep Learning

Hardware and infrastructure

- Network switches
- Infiniband
- Network attached storage
- Flash storage
- Server machines
- GPUs
- Desktops/Laptops
- NUMA machines
- Cloud compute (e.g. Amazon EC2)
- Virtual Machines
Solution: An Alg/Sys INTERFACE for Big ML
The Big-ML “Stack” - More than just software

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

**Representation:** Compact and informative features

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

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

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

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

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

© Eric Xing @ CMU, 2015
ML algorithms are Iterative-Convergent

Markov Chain Monte Carlo

Optimization
A General Picture of ML Iterative-Convergent Algorithms

Iterative Algorithm

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

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

\[ F() \] Aggregate + Transform Intermediate Updates

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

Model Parameters at iteration (t-1)
Issues with Hadoop and I-C ML Algorithms?

Naïve MapReduce not best for ML

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

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

- ML on epoch 1
  - Write outcome to KV store
  - Collect outcomes and aggregate

- ML on epoch 2
  - Write outcome to KV store
  - Do nothing

- ML on epoch 3
  - Write outcome to KV store
  - Do nothing

- ML on epoch m
  - Write outcome to KV store
  - Do nothing

Barrier?

Graph showing compute vs network time with 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 (Topics)

- gene 0.04
- dna 0.02
- genetic 0.01
- ...

- brain 0.04
- neuron 0.02
- nerve 0.01
- ...

- life 0.02
- evolve 0.01
- organism 0.01
- ...

- data 0.02
- number 0.02
- computer 0.01
- ...

Update (MCMC algo)

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

Data (Docs)

BIG DATA (billions of docs)
Example Data Parallel: Topic Models

Global shared model

gene 0.84
dna 0.02

gene 0.84
dna 0.02

gene 0.84
dna 0.02

life 0.62
evolve 0.01
organism 0.01

brain 0.04
neuron 0.02
nerve 0.01

data 0.02
number 0.02
computer 0.01

\( D \equiv \{ D_1, D_2, \ldots, D_n \} \)

MCMC algo

MCMC algo

MCMC algo

MCMC algo

MCMC algo
Model Parallelism

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

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

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

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

model parameters not updated in this iteration

© Eric Xing @ CMU, 2015
Example Model Parallel: Lasso Regression

Model (Parameter Vector)

Data (Feature + Response Matrices)

Update (CD algo)

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

BIG MODEL (100 billions of params)
Example Model Parallel: Lasso Regression

Not as easy as this picture suggests - will see why later.
A Dichotomy of Data and Model in ML Programs

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

\[ \tilde{\theta}_i \not\perp \tilde{\theta}_j \mid D, \quad \exists (i, j) \]
Data+Model Parallel: Solving Big Data+Model

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

Model (edge weights)

Data (images)

Update (backpropagation)

\[
\theta^{t+1} = \theta^t + \Delta f \hat{\theta}(D)
\]
Data+Model Parallel: Solving Big Data+Model

Tackle Deep Learning scalability challenges by combining data+model parallelism

Parameter Synchronization Channel
How difficult is data/model-parallelism?

- Certain mathematical conditions must be met

- Data-parallelism generally OK when data IID (independent, identically distributed)
  - Very close to serial execution, in most cases

- Naive Model-parallelism doesn’t work
  - NOT equivalent to serial execution of ML algo
  - Need carefully designed schedule
Intrinsic Properties of ML Programs

- ML is **optimization-centric**, and admits an **iterative convergent** algorithmic solution rather than a one-step closed form solution
  - **Error tolerance**: often robust against limited errors in intermediate calculations
  - **Dynamic structural dependency**: changing correlations between model parameters critical to efficient parallelization
  - **Non-uniform convergence**: parameters can converge in very different number of steps

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

- How do existing platforms (e.g., Spark, GraphLab) fit the above?
Challenges in Data Parallelism

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

- Challenge 1: Need “Partial” synchronicity
  - Spread network comms evenly (don’t sync unless needed)
  - Threads usually shouldn’t wait – but mustn’t drift too far apart!

- Challenge 2: Need straggler tolerance
  - Slow threads must somehow catch up

Is persistent memory really necessary for ML?
Is there a middle ground for data-parallel consistency?

- **Challenge 1: “Partial” synchronicity**
  - Spread network comms evenly (don’t sync unless needed)
  - Threads usually shouldn’t wait – but mustn’t drift too far apart!

- **Challenge 2: Straggler tolerance**
  - Slow threads must somehow catch up

![Diagram showing synchronization and straggler tolerance](image-url)
Stale Synchronous Parallel (SSP), a “bounded-asynchronous” model

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

Consequence:

- Asynchronous-like speed, BSP-like ML correctness guarantees
- Guaranteed age bound (staleness) on reads
- Contrast: no-age-guarantee Eventual Consistency seen in Cassandra, Memcached
Improving Bounded-Async via Eager Updates [Dai et al., 2015]

- Eager SSP (ESSP) protocol
  - Use spare bandwidth to push fresh parameters sooner

- Figure: difference in stale reads between SSP and ESSP
  - ESSP has fewer stale reads; lower staleness variance
  - Faster, more stable convergence (theorems later)
Enjoys Async Speed, yet BSP Guarantee, across algorithms

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

- Effective across different algorithms, e.g. LDA, Lasso, Matrix Factorization:
Example Petuum-PS Program: Tree-Structured Dirichlet Process

(b) Tree-structured stick breaking

$$\psi_\epsilon \sim \text{Beta}(1, \gamma) \quad \nu_\epsilon \sim \text{Beta}(1, \alpha)$$

$$\phi_{\epsilon,i} = \psi_{\epsilon,i} \prod_{j=1}^{i-1} (1 - \psi_{\epsilon,j})$$

$$\phi_\emptyset = 1$$

$$\pi_\epsilon = \nu_\epsilon \phi_\epsilon \prod_{\epsilon' < \epsilon} (1 - \nu_{\epsilon'}) \phi_{\epsilon'}$$

$$\pi_\emptyset = \nu_\emptyset$$

$$\sum_\epsilon \pi_\epsilon = 1$$

- Application: hierarchically-structured topic model
Example Petuum-PS Program: Tree-Structured Dirichlet Process

- Inference algorithm alternates between 2 phases
- Phase 1: Data-parallel parameter estimation
  - Fix tree structure; learn node parameters
  - Petuum-PS stores data-related sufficient statistics
  - Aggregate updates from different workers’ data samples
- Phase 2: Tree evolution
  - Create or merge tree nodes
  - Petuum-PS stores “operation records” that track tree changes
Example Petuum-PS Program: Tree-Structured Dirichlet Process

Algorithm 1 Distributed training for DNTs

1: % line 7,17: Data-parallel VI
2: % line 8-13: Model-parallel (odd/even) merge
3: % line 16,18-21: Model-parallel birth
4: Train:
5: Initialize $\{\Pi, \mu\}$ randomly
6: repeat
7:   Memoized VI
8:   Sample (odd/even) merge pairs $P \in$ assigned model part
9:   for all $(\epsilon_a, \epsilon_b) \in P$ do
10:     if $\mathcal{L}^{merge} > \mathcal{L}$ then
11:       Send merge record to PS
12:     end if
13:   end for
14:   Read all merge records from PS
15:   Update local model structure
16:   Sample birth nodes $Q \in$ assigned model part
17:   Memoized VI and collect target data
18:   for all $\epsilon \in Q$ do
19:     Restricted update $\epsilon$ and $\epsilon \cdot k$
20:     Send birth record to PS
21:   end for
22:   Read all birth records from PS
23:   Update local model structure
24: until convergence

Phase 1 data-parallel param estimation
Phase 2 merge moves
Phase 1 data-parallel param estimation
Phase 2 birth moves
Example Petuum-PS Program: Tree-Structured Dirichlet Process

- Linear speedup on Petuum-PS
  - From 1 to 4 machines

- Converged tree has 10k nodes (topics)

**Figure 3.** Convergence on PubMed  
**Figure 4.** Speedup on PubMed  
**Figure 6.** Growth of the tree
Challenges in Model Parallelism

- Recall Lasso regression:

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

A huge number of parameters (e.g.) \( J = 100M \)
Challenge 1: Model Dependencies

- Concurrent updates of $\beta$ may induce errors

Sequential updates

$\beta_1$

$\beta_2$

Concurrent updates

$\beta_1$

$\beta_2$

Sync

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

$\beta_1^{(t)} \leftarrow S(x_1^T y - x_1^T x_2 / \beta_2^{(t-1)}, \lambda)$

Induces parallelization error
Challenge 2: Uneven Convergence Rate on Parameters

- Using CD, update multiple parameters in parallel
  - Shotgun [Bradley et al. 2011] updated are chosen uniformly at random
  - Guaranteed to converge under certain conditions
  - However, parameters converge at different rates

Synchronization barrier

Round

Converged at 1 rounds
Converged at 2 rounds
Converged at 50 rounds
Converged at 100 rounds
Challenge 2: Uneven Convergence Rate on Parameters

- Convergence time determined by slowest parameters
- How to make slowest parameters converge more quickly?
Is there a middle ground for model-parallel consistency?

- Model partitioning can solve the two problems
  - Model dependencies and uneven parameter convergence
- Again, existing ways are either safe but slow, or fast but risky
- Option 1: process all data to find optimal model partitioning
  - Build full representation of data/model (e.g. via graph partitioning), explicitly compute all variable dependencies
- Option 2: randomly partition model

Is full consistency really necessary for ML?
Structure-Aware Parallelization (SAP) [Lee et al., 2014; Kumar et al., 2014]

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

- Simple programming:
  - Schedule()
  - Push()
  - Pull()

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

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

Shotgun [Bradley et al. 2011]

\[
p(j) \propto \left(\delta x_j^{(t-1)}\right)^2 + \epsilon
\]
Schedule 2: Block-based (with load balancing) [Kumar et al., 2014]

Partition data & model into $d \times d$ blocks
Run different-colored blocks in parallel

Blocks with less data run more iterations
Automatic load-balancing + better convergence
Structure-aware Dynamic Scheduler (STRADS) [Lee et al., 2014, Kumar et al., 2014]

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

- Priority Scheduling
- Block scheduling

[Kumar, Beutel, Ho and Xing, Fugue: Slow-worker agnostic distributed learning, AISTATS 2014]
Avoids dependent parallel updates, attains near-ideal convergence speed

- STRADS+SAP achieves better speed and objective
Efficient for large models

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

![Graphs showing performance of Lasso, MF, and LDA](image)

- Lasso: 9 machines
- MF: 9 machines
- LDA: 64 machines
Example Petuum-Scheduler Program: Lasso

- Application: feature selection in high dimensional data

```cpp
// STRADS Lasso

schedule() {
    // Priority-based scheduling
    for all j // Get new priorities
        c_j = f_1(j)
    for a=1..L' // Prioritize betas
        random draw s_a using [c_1, ..., c_J]
    // Get 'safe' betas
    (j_1, ..., j_L) = f_2(s_1, ..., s_L')
    return (b[j_1], ..., b[j_L])
}

push(worker = p, pars = (b[j_1],...,b[j_L])) {
    z = [] // Empty list
    for a=1..L // Compute partial sums
        z.append( f_3(p, j_a) )
    return z
}

pull(workers = [p], pars = (b[j_1],...,b[j_L]),
    updates = [z]) {
    for a=1..L // Aggregate partial sums
        b[j_a] = f_4(j_a, [z])
}
```
Lasso schedule() has two parts:

1. Priority selection:
   \[ \mathcal{U} = \{x_j\} \sim \left( \delta x_j^{(t-1)} \right)^2 + \epsilon \]

2. Dependency checking:
   \[ |a_j^T a_k| < \rho \text{ for all } j \neq k \in \mathcal{U} \]
   Discard parameters that violate the above condition

Once selection and checking are finished, dispatch parameters to workers
Example Petuum-Scheduler Program: Lasso

1. Compute `schedule()`, then dispatching parameters to be updated

2. Each machine computes partial results for assigned parameters using `push()`

3. Aggregating partial results using `pull()`

Priority-based selection

Scheduler

\{10, 12, \ldots, 20\}
\{10\} \{12\} \{20\}

Machine 1

\(f\{a_{10}\}\)

Machine 2

\(f\{a_{12}\}\)

Machine M

\(\ldots\)

\(f\{a_{20}\}\)

Priority-based scheduling w/ dependency checker
For high-dim problems, schedule() greatly improves convergence rate of Lasso

- Sharp drop due to prioritization and dependency checking

Uneven, power-law-like parameter convergence is a big reason for the speedup

- 85% parameters converged in 5 iterations, but need 100+ iterations for the remaining 15%!
Theory of (Ideal) ML Systems
Theory of sequential ML

- Sequential algos well studied in literature

- What are desired properties for ML algorithm?
- Convergence property
  - Optimization
    - Model param $\theta$ gets closer to true optimum $\theta^*$ with more iterations or samples
  - MCMC and Stochastic Variational Inference
    - Learned distribution eventually matches true model posterior, after enough steps
  - Convergence may be “in expectation” – each step not guaranteed to get closer to true optimum or posterior

- Stability property
  - Optimization
    - Model parameter $\theta$ does not move much (low variance) when an optimum is reached
  - Important for stochastic algorithms, which may “fluctuate” near convergence
    - high stability $\Rightarrow$ less fluctuation $\Rightarrow$ easier to determine convergence
Why study parallel ML theory?

- What sequential guarantees still hold in parallel setting?
  - Under what conditions?

- Growing body of literature for “ideal” parallel systems
  - Serializable—equivalent to single-machine execution in some sense
  - Focused on per-iteration analysis
    - Abstract away computational/comms cost
    - Predicting real-world running time requires these costs to be put back

- “Real-world” parallel systems a work in progress
  - Asynchronous or bounded-async approaches can empirically work better than synchronous approaches
    - Need additional theoretical analysis to understand why
    - Async => no serializability… why does it still work?
  - Parallelization requires data and/or model partitioning… many strategies exist
    - Want partitioning strategies that are provably correct
    - Need to determine when/where independence is violated, and what impact such violation has on algorithm correctness
Types of ML systems

- **Sequential (single-worker) learning**
  - Non-parallel, but rich body of theoretical work

- **Methods for “ideal” systems**
  - “Embarrassingly-parallel” (EP) learning
    - Distributed learning with little to no communication; easy to implement
  - Synchronous execution
    - Data-parallel execution serializable, conventional sequential guarantees usually hold
    - Can deploy on Hadoop & Spark without worrying about correctness
    - Expensive under load imbalance or stragglers! (curse of the last reducer)
Types of ML systems

- **Sequential (single-worker) learning**
  - Non-parallel, but rich body of theoretical work

- **Methods for real systems**
  - Scheduled or slow-worker agnostic execution
    - Rebalance worker sample size to deal with load imbalance or stragglers
  - Bounded-asynchronous execution
    - Allow parameters to be stale, and workers to be (temporarily slow)
    - Why correct? Bounded loss of serializability => “close” to sequential execution
  - Model-parallelization
    - Data-parallel provably safe because of IID data assumption
    - Model parameters are not always independent of each other; must schedule to avoid updating dependent parameters in parallel
Correctness of Embarrassingly Parallel Learning

- Sequential algorithms known to converge

- Embarrassingly Parallel learning compensates for non-ideal behavior of real systems, by eliminating communication
  - No communication until end of algorithm
  - Intuition: just average parameters once all (independent) workers have finished

- Does EP learning lead to convergence?
  - For MCMC?
  - For optimization, e.g. SGD?
EP-MCMC convergence guarantee
[Neiswanger et al., 2014]

\textbf{Theorem 5.3.} \textit{If } $h \propto T^{-1/(2\beta+d)}$, \textit{the mean-squared error of the estimator } $p_1\cdots p_M(\theta)$ \textit{satisfies}

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

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

- \textbf{Explanation}: the nonparametric estimator generated by subposterior combination is consistent

- \textbf{Benefit}: no comms needed by EP-MCMC; Hadoop-friendly

- \textbf{Drawback}: subposterior combination requires costly product of sums

$$
p_1\cdots p_M(\theta) = \hat{p}_1\cdots \hat{p}_M(\theta) = \frac{1}{T^M} \prod_{m=1}^{M} \sum_{t_m=1}^{T} \mathcal{N}_d(\theta|\theta_{t_m}^m, h^2 I_d) \propto \sum_{t_1=1}^{T} \cdots \sum_{t_M=1}^{T} w_{t_1}\cdots w_{t_M} \mathcal{N}_d(\theta|\bar{\theta}_{t_1}, \frac{h^2}{M} I_d)
$$
EP-Stochastic Gradient Descent convergence guarantee

[Zinkevich et al., 2011]

- Setting: perform SGD independently on k machines, each using T data points. Then, average the parameter vectors w.
  - Theorem [Zinkevich et al., 2011]: Let $D_{n,k}^T$ be the output after T samples on each of k machines, with learning rate $\eta$. Then, for constants $G$ and $\lambda$, and for strongly convex objective:

$$
\text{If } \eta \leq \eta^* \text{ and } T = \frac{\ln k - (\ln \eta + \ln \lambda)}{2\eta \lambda}
$$

$$
E_{w \in D_{n,k}^T}[c(w)] - \min_w c(w) \leq \frac{8\eta G^2}{\sqrt{k} \lambda} \sqrt{\|\nabla c\|_L} + \frac{8\eta G^2}{k \lambda} \|\nabla c\|_L + (2\eta G^2)
$$

- Explanation: EP-SGD parameter vector $w$ is close to the true minimum (in terms of objective function $c(w)$)
  - More machines $k$ => terms 1,2 shrink => faster convergence
  - However, term 3 eventually dominates => further parallelization does not help
Weakness of EP Learning?

- Multimodal or non-concave functions can be problematic
  - Average of two modes may not be a mode!
  - EP-SGD faces this issue in practice; many ML problems are non-convex/concave
  - EP-MCMC addresses this issue by subposterior combination, but this is computationally expensive
Correctness of Synchronous Learning

- Embarrassingly Parallel learning converges…
  - But cannot handle multimodal/non-concave functions without special care
  - Weakness of averaging/communicating just once

- Synchronous learning – workers can communicate many times before termination
  - For example, at a barrier placed at the end of each iteration
  - Assumes “ideal” system properties:
    - Communication is not expensive relative to computation (algorithm execution)
    - Workers arrive at barrier at the same time (otherwise they must wait for each other)

- Does Synchronous learning converge?
  - For optimization?
  - For MCMC?
Synchronous ADMM convergence guarantee

[Eckstein & Bertsekas, 1992; Boyd et al., 2010]

- Setting: perform ADMM on $f() + g()$, where $f$, $g$ are closed, proper, and convex, and Lagrangian has a saddle-point

Theorem [Eckstein & Bertsekas, 1992; Boyd et al., 2010]

ADMM iterates $(w^t, z^t; \lambda^t)$ satisfy:

- primal optimality: $f(w^t) + g(z^t) \rightarrow p^* := \min_{Aw+Bz=c} f(w) + g(z)$
- dual convergence: $\lambda^t \rightarrow \lambda^*$ for some dual maximizer $\lambda^*$
- feasibility: $r^t := Aw^t + Bz^t - c \rightarrow 0$
- primal convergence: $(w^t, z^t) \rightarrow (w^*, z^*)$ if bounded or unique

- Explanation: ADMM has same optimal solution as original problem; convergence is eventually guaranteed
  - Accelerated variants converge as fast as $O(1/t^2)$ [Goldfarb and Ma, 2012]
  - After $t$ iterations, $[f(x^{(t)})+g(x^{(t)})] - [f(x^*)+g(x^*)]$ has shrunk to a factor of $O(1/t^2)$
Synchronous Auxiliary Variable
MCMC Exactness Guarantee

[Dubey et al., 2013, 2014]

- Recall: Auxiliary Variable Inference
- Reformulate single model as P independent models, and then parallelize over P workers
  - Dirichlet Process example:

\[
D_j \sim DP\left(\frac{\alpha}{P}, H\right), \quad j = 1, \ldots, P
\]
\[
\phi \sim \text{Dirichlet}\left(\frac{\alpha}{P}, \ldots, \frac{\alpha}{P}\right)
\]
\[
\pi_i \sim \phi
\]
\[
\theta_i \sim D_{\pi_i}
\]
\[
x_i \sim f(\theta_i), \quad i = 1, \ldots, N.
\]
\[
D \sim DP(\alpha, H),
\]
\[
\theta_i \sim D,
\]
\[
x_i \sim f(\theta_i)
\]
Synchronous Auxiliary Variable
MCMC Exactness Guarantee

[Dubey et al., 2013, 2014]

- Auxiliary Variable reformulation is exact:

  **Theorem 1** (Auxiliary variable representation for the DPMM). We can re-write the generative process for a DPMM (given in Eq. 1) as

  \[
  D_j \sim DP\left(\frac{\alpha}{P}, H\right), \quad \phi \sim Dirichlet\left(\frac{\alpha}{P}, \ldots, \frac{\alpha}{P}\right),
  \]
  \[
  \pi_i \sim \phi, \quad \theta_i \sim D_{\pi_i}, \quad x_i \sim f(\theta_i),
  \]

  for \( j = 1, \ldots, P \) and \( i = 1, \ldots, N \). The marginal distribution over the \( x_i \) remains the same.

- **Explanation**: AV reformulation has identical marginal distribution over data \( x_i \) \( \Rightarrow \) sampling from AV reformulation equivalent to sampling original model

- **Advantage**: Collapsed gibbs sampler for AV reformulation can be correctly parallelized, unlike original model!
Weakness of Synchronous Learning?

- Speedup rarely P-fold in practice. Two reasons:
  - Slow workers exist in real clusters
    - Causes: background jobs, other users' jobs, datacenter environment, etc.
    - Faster workers wait at barriers for slower workers to catch up
  - Communication at barrier non-zero
    - Can take as long as, or even longer, than the computation done by workers
Theory of Real Distributed ML Systems
Challenges in real-world distributed systems

- Real-world systems need asynchronous execution and load balancing
  - Synchronous system: load imbalances => slow workers => waiting at barriers
  - Need load balancing to reduce load at slow workers
  - Need asynchronous execution so faster workers can proceed without waiting

- Solution 1: key-value stores
  - Automatically manages communication with bounded asynchronous guarantees

- Solution 2: scheduling systems
  - Automatically balances workload across workers; also performs prioritization and dependency checking
Communication strategies

- **Data parallel**
  - Partition data across workers
    - Or fetch small batches of data in an online/streaming fashion
  - Communicate model as needed to workers
    - e.g. key-value store with bounded asynchronous model – theoretical consequences?

- **Model parallel**
  - Partition model across workers
    - Model partitions can change dynamically during execution – theoretical consequences?
  - Send data to workers as needed (e.g. from shared database)
    - Or place full copy of data on each worker (since data is immutable)

- **Data + Model parallel?**
  - Partition both data and model across workers
  - Wide space of strategies; need to reduce model and data communication
    - Reduce model communication by exploiting independence between variables
    - Reduce data and model communication via broadcast strategies, e.g. Halton sequence
Bridging Models for Parallel Programming

- **Bulk Synchronous Parallel [Valiant, 1990]** is a bridging model
  - Bridging model specifies how/when parallel workers should compute, and how/when workers should communicate
  - Key concept: barriers
    - No communication before barrier, only computation
    - No computation inside barrier, only communication
  - Computation is “serializable” – many sequential theoretical guarantees can be applied with no modification

- **Bounded Asynchronous Parallel (BAP) bridging model**
  - Key concept: bounded staleness [Ho et al., 2013; Dai et al., 2015]
    - Workers re-use old version of parameters, up to s iterations old – no need to barrier
    - Workers wait if parameter version older than s iterations
Types of Convergence Guarantees

- Regret/Expectation bounds on parameters
  - Better bounds $\Rightarrow$ better convergence progress per iteration

- Probabilistic bounds on parameters
  - Similar meaning to regret/expectation bounds, usually stronger in guarantee

- Variance bounds on parameters
  - Lower variance $\Rightarrow$ higher stability near optimum $\Rightarrow$ easier to determine convergence

- For data parallel?
- For Model parallel?
- For Data + model parallel?
BAP Data Parallel: Can we do value-bounding?

- **Idea**: limit model parameter difference \( \Delta \theta_{i,j} = ||\theta_i - \theta_j|| \) between machines \( i,j \) to < a threshold

- Does not work in practice!
  - To guarantee that \( \Delta \theta_{i,j} \) has not exceeded the threshold, machines must wait to communicate with each other
  - No improvement over synchronous execution!

- Rather than controlling parameter difference via magnitude, what about via iteration count?
  - This is the (E)SSP communication model…
BAP Data Parallel: (E)SSP model [Ho et al., 2013; Dai et al., 2015]

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

### Consequence:
- Asynchronous-like speed, BSP-like ML correctness guarantees
- Guaranteed age bound (staleness) on reads
- Contrast: no-age-guarantee Eventual Consistency seen in Cassandra, Memcached
BAP Data Parallel: (E)SSP Regret Bound [Ho et al., 2013]

- **Goal**: minimize convex \( f(x) = \frac{1}{T} \sum_{t=1}^{T} f_t(x) \)
  
  (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 \( \sigma = \frac{F}{L \sqrt{2(s+1)P}} \)

- (E)SSP converges according to
  
  - Where \( T \) is the number of iterations

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

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

- Stronger guarantees on means and variances can also be proven

© Eric Xing @ CMU, 2015
Intuition: Why does (E)SSP converge?

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

\[ \epsilon \leq C(2s - 1) \]
SSP versus ESSP: What is the difference?

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

- What impact does ESSP have on convergence speed and stability?
Theorem: Given $L$-Lipschitz objective $f_t$ and stepsize $h_t$,

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

Let real staleness observed by system be $\gamma_t$. Let its mean, variance be $\mu_\gamma = \mathbb{E}[\gamma_t], \sigma_\gamma = \text{var}(\gamma_t)$

**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.**

**Take-away:** controlling staleness mean $\mu_\gamma$, variance $\sigma_\gamma$ (on top of max staleness $s$) is needed for faster ML convergence, which ESSP does.
Theorem: the variance in the (E)SSP estimate is

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

where

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

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 \( \gamma_t \) => Lower variance in parameter => Less oscillation in parameter => More confidence in estimate quality and stopping criterion.

Take-away: Lower average staleness (via ESSP) not only improves convergence speed, but also yields better parameter estimates.
ESSP vs SSP: Increased stability helps empirical performance

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

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

**STRADS**

- **Check Variable Dependency**
- **Generate Blocks of Variables**
- **All Variables**
- **Sample Variables to be Updated \( \sim p(j) \)**

- **Priority Scheduling**

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

- **Block scheduling**

- **Worker 1**
- **Worker 2**
- **Worker 3**
- **Worker 4**

- **Round 1**
- **Round 2**
- **Round 3**
- **Round 4**
Scheduled Model Parallel: Dynamic Scheduling Expectation Bound

[Lee et al. 2014]

- **Goal:** solve sparse regression problem
  \[
  \min_{\beta} \| y - X\beta \|_2^2 + \lambda \sum_j |\beta_j|
  \]
  - Via coordinate descent over “SAP blocks” \( X^{(1)}, X^{(2)}, ..., X^{(B)} \)
  - \( X^{(b)} \) are the data columns (features) in block \((b)\)
  - \( P \) parallel workers, \( M \)-dimensional data
  - \( \rho = \text{SpectralRadius}[\text{BlockDiag}[(X^{(1)})^T X^{(1)}, ..., (X^{(t)})^T X^{(t)}]]; \) this block-diagonal matrix quantifies the maximum level of correlation (and hence problem difficulty) within all the SAP blocks \( X^{(1)}, X^{(2)}, ..., X^{(t)} \)

- **SAP converges according to**
  - Where \( t \) is # of iterations
  - Gap between current parameter estimate and optimum
    \[
    \mathbb{E} \left[ f(X^{(t)}) - f(X^*) \right] \leq \frac{\mathcal{O}(M)}{P - \mathcal{O}(P^2 \rho)} \frac{1}{t} = \mathcal{O} \left( \frac{1}{Pt} \right)
    \]
    - SAP explicitly minimizes \( \rho \), ensuring as close to \( 1/P \) convergence as possible
  - **Take-away:** SAP minimizes \( \rho \) by searching for feature subsets \( X^{(1)}, X^{(2)}, ..., X^{(B)} \) without cross-correlation => as close to \( P \)-fold speedup as possible
Scheduled Model Parallel:
Dynamic Scheduling Expectation Bound is near-ideal
[Xing et al. 2015]

Let $S^{ideal}()$ be an ideal model-parallel schedule
Let $\beta_{ideal}^{(t)}$ be the parameter trajectory due to ideal scheduling
Let $\beta_{dyn}^{(t)}$ be the parameter trajectory due to SAP scheduling

**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, this is because both ideal and SAP model-parallelism minimize the parameter dependencies between parallel workers.
Scheduled Model Parallel: Dynamic Scheduling Empirical Performance

- Dynamic Scheduling for Lasso regression (SMP-Lasso): almost-ideal convergence rate, much faster than random scheduling (Shotgun-Lasso)
Scheduled Data+Model Parallel: Block-based Scheduling (with load balancing)

[Kumar et al. 2014]

Partition data & model into $d \times d$ blocks
Run different-colored blocks in parallel

Blocks with less data run more iterations
Automatic load-balancing + better convergence
Scheduled Data+Model Parallel: Block-based Scheduling Variance Bound 1

[Kumar et al. 2014]

- Variance between iterations $S_{n+1}$ and $S_n$ is:

$$Var(\Psi_{S_{n+1}}) = Var(\Psi_{S_n}) - \left(2\eta S_n\right) \sum_{i=1}^{w} n_i \Omega_0^i Var(\psi^i_{S_n})$$

$$- 2\eta S_n \sum_{i=1}^{w} n_i \Omega^i_0 Cov(\psi^i_{S_n}, \delta^i_{S_n}) + \eta^2 S_n \sum_{i=1}^{w} n_i \Omega^i_1 + O(\Delta S_n)$$

- Explanation:
  - higher order terms (red) are negligible
  - $\Rightarrow$ parameter variance decreases every iteration

- Every iteration, the parameter estimates become more stable
Intra-block variance: Within blocks, suppose we update the parameters $\psi$ using $n_i$ data points. Then, variance of $\psi$ after those $n_i$ updates is:

$$\text{Var}(\psi^{t+n_i}) = \text{Var}(\psi^t) - 2\eta_t n_i \Omega_0 (\text{Var}(\psi^t))$$

$$- 2\eta_t n_i \Omega_0 \text{Cov}(\psi_t, \delta_t) + \frac{\eta_t^2 n_i \Omega_1}{\Delta_t}$$

$$+ \underbrace{O(\eta_t^2 \rho_t) + O(\eta_t \rho_t^2) + O(\eta_t^3) + O(\eta_t^2 \rho_t^2)}_{\Delta_t}$$

**Explanation:**
- Higher order terms (red) are negligible
- $\Rightarrow$ doing more updates within each block decreases parameter variance, leading to more stable convergence

**Load balancing by doing extra updates is effective**
Scheduled Data+Model Parallel: Block-Scheduling Empirical Performance

- Slow-worker Agnostic Block-Scheduling (Fugue) faster than:
  - Embarrassingly Parallel SGD (PSGD)
  - Non slow-worker Agnostic Block-Scheduling (Barriered Fugue)

- Slow-worker Agnostic Block-Scheduling converges to a better optimum than asynchronous GraphLab
  - Reason: more stable convergence due to block-scheduling

- Task: Imagenet Dictionary Learning
  - 630k images, 1k features
Future work: BAP Model-Parallel Guarantees

- Model-parallel under synchronous setting:
  - Dynamic scheduling
  - Slow-worker block-based scheduling

- Synchronous slow-worker problem solved by:
  - Load balancing (for dynamic scheduling)
  - Allow additional iters while waiting for other workers (slow-worker scheduling)

- Work in progress: theoretical guarantees for bounded-async model-parallel execution
  - Intuition: model-parallel sub-problems are nearly independent (thanks to scheduling)
  - Perhaps better per-iteration convergence than bounded-async data-parallel learning?
What parameters to communicate?

- From a systems perspective, distributed learning is challenging because networks are slow
  - Communication time can easily dominate useful computation time

- Bounded-async strategies are one solution
  - Use stale parameters and async communication to reduce communication time

- What else can we do to mitigate network slowness?
  - Idea: communicate fewer, but more important, parameters
Communication Strategy 1: Stochastic, Prioritized Sending

[Lee et al., 2014, Wei et al., 2015]

**Petuum-STRADS model-parallelism**
- Sub-problems have nice property: only touch small fraction of parameters $\theta$
- Therefore, only send necessary subset of $\theta$ to each worker
- Similar to model-circulation in topic model (LDA) research

**Petuum-PS (Bösen) data-parallelism**
- In principle, data-parallel requires all parameters to be communicated…
- … but can afford to be “more stale” on parameters that are changing slowly
- Therefore, prioritize which params to send based on rate of change
- Better final result compared to plain (E)SSP
- Approximately 2x convergence speed on LDA algorithm
Communication Strategy 2: Sufficient Factor Broadcasting

[In publication]

- Some ML models have matrix-shaped parameters
  - Sparse coding, multiclass LR, distance metric learning

- Common property: for every data sample, the (stochastic) parameter update is low-rank or rank-1:
  - Example: $\Theta^{(t+1)} = \Theta^{(t)} + \sum_i u_i v_i^T$
  - where $i$ indexes data samples, and $u,v$ are vectors
  - $u_i v_i^T = \Delta\Theta_i$, which is the update to $\Theta$ due to sample $i$
Communication Strategy 2: Sufficient Factor Broadcasting

[In publication]

- Sending $\Theta$ is expensive; $\Theta$ can be very large:
  - e.g. Multiclass LR on 10000-class Imagenet challenge: $|\Theta|$ is almost 10 billion!

- But the “sufficient factors” $(u_1,v_1), \ldots, (u_M,v_M)$ are much smaller! (M is minibatch size)
  - For reasonable minibatch sizes $M=100$ to $1000$, $M$ sufficient factors is 3+ orders of magnitude smaller than $\Theta$!
  - Sufficient Factor Broadcasting (SFB) 4x faster runtime than sending full matrix $\Theta$ (Full Matrix Sync, FMS)

![Runtime comparison graph](image-url)
Open Research
Issues and Topics
The Landscape of Big ML

LDA - Topic Model
MF - Matrix Factorization
CNN - Convolutional Neural Network

*GPU cores

Number of Cores

Number of Model Parameters
The Landscape of Big ML

Trend over last 5 years:
More cores, bigger models

- 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*
The Landscape of Big ML

Possible to learn bigger, more powerful models with only reasonable # of cores?
Issue: When is Big Data useful?

- **Negative examples**
  - “Simple” regression and classification models, with fixed parameter size
  - **Intuition:** decrease estimator variance has diminishing returns with more data. Estimator eventually becomes “good enough”, and additional data/computation is unnecessary

- **Positive examples**
  - Topic models (internet/tech industry)
  - DNNs (Google, Baidu, Microsoft, Facebook, etc.)
  - Collaborative filtering (internet/tech industry)
  - Personalized models
  - **Industry practitioners sometimes increase model size with more data**

- **Conjecture:** how much data is useful really depends on model size/capacity
Issue: Are Big Models useful?

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

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

- 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

- 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)
Issue: Inference Algorithms, or Inference Systems?

- View: focus on inference algorithm
  - Scale up by refining the algorithm
    - Given fixed computation, finish inference faster

- Some 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

- Some examples
  - Spark
  - GraphLab
  - Petuum
Issue: Theoretical Guarantees and Empirical Performance

- View: establishing theoretical guarantees gives practitioners 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, industrial evidence can provide strong driving force for experimental research
  - Motivated by industrial practice, particularly at internet companies

- Example: AB testing in industry
  - Principled means of testing new algorithms, feature engineering; by experimenting on user base
  - Determine if new method makes a significant difference to click-through rate, user adoption, etc.
Open research topics

- Future of data-, model-parallelism, and other ML properties
  - New properties, principles still undiscovered
  - Potential to accelerate ML beyond naive strategies

- Deep analysis of BigML systems still limited to few ML algos
  - Model of ML execution under error due to imperfect system?

- How to express more ML algorithms in table form (Spark, Petuum), or graph form (GraphLab)
  - Tree-structured algorithms? Infinite-dimensional Bayesian nonparametrics?
  - What are the key elements of a generic ML programming interface?
Acknowledgements
Thank You!