How to Go Really Big in AI:
Strategies & Principles for Distributed Machine Learning

Eric Xing
epxing@cs.cmu.edu
School of Computer Science
Carnegie Mellon University

Acknowledgement:
Wei Dai, Qirong Ho, Jin Kyu Kim, Abhimanu Kumar, Seunghak Lee, Jinliang Wei, Pengtao Xie, Yaoliang Yu, Hao Zhang, Xun Zheng
James Cipar, Henggang Cui,
and, Phil Gibbons, Greg Ganger, Garth Gibson
Machine Learning:
-- a view from outside
Inside ML ...

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

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
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
Growing Model Complexity

- **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 Scalability Challenge

![Graph showing scalability challenge with processing power/speed vs. number of "Machines". The graph includes lines for Perfect, Pathetic, and Good! scenarios.]
Why need new Big ML systems?

Today’s AI & ML imposes high CAPEX and OPEX

- Example: The Google Brain AI & ML system

- High CAPEX
  - 1000 machines
  - $10m+ capital cost (hardware)
  - $500k+/yr electricity and other costs

- High OPEX
  - 3 key scientists ($1m/year)
  - 10+ engineers ($2.5m/year)

- Total 3yr-cost = $20m+

- Small to mid companies and the Academic do not have such luxury

- 1000 machines only 100x as good as 1 machine!
Why need some new thinking?

MLer’s view

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

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

### Compute vs Network
LDA 32 machines (256 cores)

- Network waiting time
- Compute time

- Parallelize over worker threads
- Share global model parameters via RAM
Why need some new thinking?

**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**

```
[Diagram showing synchronization model]
```

**Programming model**

```
[Diagram showing programming model]
```
Why need some new thinking?

MLer’s view

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

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

Oversimplify systems issues

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

Systems View:

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

Oversimplify ML issues and/or ignore ML opportunities

- **ML algos “just work” without proof**
- Conversion of ML algos across different program models (graph programs, RDD) is easy
Existing Solution:

Machine Learning Models/Algorithms

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

Hardware and infrastructure

- Network switches
- Network attached storage
- Infiniband
- Flash storage
- Server machines
- GPUs
- Desktops/Laptops
- NUMA machines
- Cloud compute (e.g. Amazon EC2)
- Virtual Machines
How about this … [Xing et al., 2015]
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

\[
\text{for } (t = 1 \text{ to } T) \{
\text{doThings()}
\quad \tilde{\theta}^{t+1} = g(\tilde{\theta}^t, \Delta_f \tilde{\theta}(\mathcal{D}))
\quad \text{doOtherThings()}
\}
\]

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

Familiar problem: data from 50B devices, data centers won’t fit into memory of single machine
Challenge #2
– Gigantic Model Size

Big Data needs Big Models to extract understanding
But ML models with >1 trillion params also won’t fit!
Typical ML Programs (about the “f”)

- **Optimization programs:**

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

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

A huge number of parameters (e.g.) \( M = 1B \)
Typical ML Programs (about the “f”)  

- Probabilistic programs  

\[ z_{di} \sim p(z_{di} = k | \text{rest}) \propto (n_{kd}^{-di} + \alpha_k) \cdot \frac{n_{kw}^{-di} + \beta_w}{n_k^{-di} + \beta_V} \]
for (t = 1 to T) {
  doThings()
  parallelUpdate(x, θ)
  doOtherThings()
}
Usually, we worry ...

A sequential program

\[ \beta_1 \overset{?}{\rightarrow} \beta_2 \]

A parallel program

\[ \beta_1 \overset{?}{\rightarrow} \beta_2 \rightarrow \beta_1 \rightarrow \beta_2 \]

but assuming an ideal system, e.g.,
- zero-cost sync,
- zero-cost fault recovery
- uniform local progress
- ...

for \( t = 1 \) to \( T \)

\[
\begin{align*}
&\text{doThings()} \\
&\text{parallelUpdate(x,0)} \\
&\text{doOtherThings()}
\end{align*}
\]
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

Sorting error: 2 after 5

Error persists and is not corrected
ML Algorithms can Self-heal

for (t = 1 to T) {
  doThings()
  \( \tilde{\theta}^{t+1} = g(\tilde{\theta}^t, \Delta f(\tilde{\theta}(D))) \)
  doOtherThings()
}
Intrinsic Properties of ML Programs
[Xing et al., 2015]

- **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
Intrinsic Properties of ML Programs
[Xing et al., 2015]

- 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 Big Data platforms fit the above?
Two Parallel Strategies for ML
A Dichotomy of Data and Model in ML Programs

\[ \bar{\theta}^{t+1} = \bar{\theta}^t + \Delta f \bar{\theta}(D) \]
A Dichotomy of Data and Model in ML Programs

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

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

Data Parallel

\[
\Delta \theta(D_i) \quad \Delta \theta(D_j) \quad \Delta \theta(D_k)
\]

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

Model Parallel

\[
\bar{\theta}_i \neq \bar{\theta}_j \mid D, \exists (i, j)
\]
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 \( X\beta \):
    \[
    \sum_{i=1}^{N} \left\| y_i - X_i \beta \right\|_2^2
    \]

- Regularization \( \mathcal{R}(\theta) \)
  - \( L_1 \) 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:

Model (Parameter Vector) → Update (prox SGD or CD algo) → Data (Feature + Response Matrices)

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

\[ \overrightarrow{\theta}^{t+1} = \overrightarrow{\theta}^t + \Delta f \overrightarrow{\theta}(D) \]
Data-Parallel Lasso

Proximal SGD:

Global shared model

Partition rows of Feature+Response Matrices across workers
Model-Parallel Lasso

Coordinate Descent:

\[ \theta \equiv [\theta_1^T, \theta_2^T, \ldots, \theta_k^T]^T \]

Worker machines with local model
Probabilistic Example:

Topic Models

- **Objective** $L(\theta, D)$
  - Log-likelihood of $D = \{\text{document words } x_{ij}\}$ given unknown $\theta = \{\text{document word topic indicators } z_{ij}, \text{ doc-topic distributions } \delta_i, \text{ topic-word distributions } B_k\}$:
    \[
    \sum_{i=1}^{N} \sum_{j=1}^{N_i} \ln p_{\text{Categorical}}(x_{ij} \mid z_{ij}, B) + \sum_{i=1}^{N} \sum_{j=1}^{N_i} \ln p_{\text{Categorical}}(z_{ij} \mid \delta_i)
    \]

- **Prior** $\rho(\theta)$
  - Dirichlet prior on $\theta = \{\text{doc-topic, word-topic distributions}\}$
    \[
    \sum_{i=1}^{N} \ln p_{\text{Dirichlet}}(\delta_i \mid \alpha) + \sum_{i=k}^{K} \ln p_{\text{Dirichlet}}(B_k \mid \beta)
    \]
  - $\alpha, \beta$ are “hyperparameters” that control the Dirichlet prior’s strength

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

Collapsed Gibbs sampling

Data (Docs) = $x_{ij}$

Model (Topics) = $B_k$

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 $B, \delta$:

$B_{k_{old}, w_{ij}} = B_{k_{old}, w_{ij}} - 1$

$B_{k_{new}, w_{ij}} = B_{k_{new}, w_{ij}} + 1$

$\delta_{i, k_{old}} = \delta_{i, k_{old}} - 1$

$\delta_{i, k_{new}} = \delta_{i, k_{new}} + 1$

\[
\bar{\theta}^{t+1} = \bar{\theta}^t + \Delta_f \bar{\theta}(\mathcal{D})
\]
Data Parallel Gibbs

Global shared model

gene 0.84
dna  0.82
genetic 0.81
...
life 0.82
evolve 0.81
organism 0.81
...
brain 0.84
neuron 0.82
nerve 0.81
...
data 0.02
number 0.02
computer 0.01
...

\[ \mathcal{D} = \{ \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n \} \]
D+M Parallel Gibbs

Pair up vocabulary words with documents, divide across workers

Parameter Synchronization Channel
What’s Next?

First-timer’s “Ideal View” of ML

```
global model = (a,b,c,...)  
global data = load(file)

Update(var a):  
a = doSomething(data,model)

Main:  
do Update() on all var in model until converged
```

Reality of High-performance implementations

Many considerations
- What data batch size?
- How to partition model?
- When to sync up model?
- How to tune step size?
- What order to Update()?

1000s of lines of extra code

Need a System Interface for Parallel ML

– Does ML really Stop at the Ideal View?
4 Principles of ML System Design

How to execute distributed-parallel ML programs?
ML program equations tell us “What to Compute”. But…

1. **How to Distribute?**

2. **How to Bridge Computation and Communication?**

3. **How to Communicate?**

4. **What to Communicate?**
Principles of ML system Design [Xing et al., to appear 2016]

1. How to Distribute: Scheduling and Balancing workloads
Example: Model Distribution

Lasso via coordinate descent:

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

- How to correctly divide computational workload across workers?
- What is the best order to update parameters?

A huge number of parameters (e.g.) \( M > 100 \text{ million} \)
Beware Model Dependencies

- Concurrent updates of $\beta$ may induce errors

Sequential updates

$$\beta_1 \rightarrow \beta_2$$

Concurrent updates

$$\beta_1 \rightarrow \beta_2 \rightarrow \beta_1 \rightarrow \beta_2$$

Sync

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

Decreases iteration progress

$$\beta_1^{(t)} \leftarrow S(x_1^T y - x_1^T x_2 \beta_2^{(t-1)}, \lambda)$$
Avoid Dependency Errors via Data+Model Scheduling [Lee et al., 2014]

**Task**: subset of data \((X, y)\) & model \((\beta)\)

**Schedule**: find parallel task plan that minimizes parameter dependencies

\[ x_1^T x_2 \beta_2^{(t-1)} \]
Structure-Aware Parallelization (SAP)

[Lee et al., 2014] [blinded, to appear]

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

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

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

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

pull(workers = [p], vars = (x[j_1],...,x[j_U]),
     updates = [z]) {
    // Use partial updates z from workers p to
    // update U vars x[j]. sync() is automatic.
    ....
}
```
Structure-aware Dynamic Scheduler (Strads) [Lee et al., 2014]

Strads System

(1) Partition Data + Model into Tasks
(2) Schedule & Prioritize Tasks onto Workers
(3) Balance Task Load on each Worker

• 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

Load-balanced Tasks
Sync. barrier

Kumar, Beutel, Ho and Xing, Fugue: Slow-worker agnostic distributed learning, AISTATS 2014
SAP Scheduling: Faster, Better Convergence across algorithms

- SAP on Strads achieves better speed and objective

**Graphs:***
- Lasso
  - Objective: STRADS vs. Lasso-RR
  - Y-axis: Objective
  - X-axis: Seconds
  - 100M features, 9 machines

- MF
  - RMSE: STRADS vs. GraphLab
  - Y-axis: RMSE
  - X-axis: Seconds
  - 80 ranks, 9 machines

- LDA
  - Log-Likelihood: STRADS vs. YahooLDA
  - Y-axis: Log-Likelihood
  - X-axis: Seconds
  - 2.5M vocab, 5K topics, 32 machines
SAP gives Near-Ideal Convergence Speed [Xing et al., 2015]

- **Goal:** solve sparse regression problem
  - Via coordinate descent over “SAP blocks” $X^{(1)}, X^{(2)}, \ldots, X^{(B)}$
    - $X^{(b)}$ are data columns (features) in block $(b)$
  - $P$ parallel workers, $M$-dimensional data
- $\rho =$ Spectral Radius[BlockDiag$[(X^{(1)})^{T}X^{(1)}, \ldots, (X^{(t)})^{T}X^{(t)}]]$; this block-diagonal matrix quantifies max level of correlation within all SAP blocks $X^{(1)}, X^{(2)}, \ldots, X^{(t)}$

- **SAP converges according to**

\[
\mathbb{E} \left[ f(X^{(t)}) - f(X^*) \right] \leq \frac{O(M)}{P - O(P^2\rho) \frac{1}{M}} \frac{1}{t} = O \left( \frac{1}{Pt} \right)
\]

where $t$ is # of iterations

- **Take-away:** SAP minimizes $\rho$ by searching for feature subsets $X^{(1)}, X^{(2)}, \ldots, X^{(B)}$ w/o cross-correlation => as close to $P$-fold speedup as possible
How to SAP-LDA
[Zheng et al., to appear 2015]

At iteration \((t)\):
- Worker 1 samples docs+words in \(Z_1^{(t)}\)
- Worker 2 \(\leftarrow Z_2^{(t)}\), Worker 3 \(\leftarrow Z_3^{(t)}\) and so on...
- Use different-sized \(Z_p^{(t)}\) to load balance power-law tokens
SAP-LDA performance [blinded, to appear]

- **Ideal rate**: progress per iter preserved from 25 → 100 machines
  - Thanks to dependency checking

- **Near-ideal throughput**: data rate 1x → 3.5x from 25→100 machines
  - Thanks to load balancing

- **Convergence Speed = rate x throughput**
  - Therefore near-ideal 3.5x speedup from 25→100 machines

### 80GB data, 2M words, 1K topics, 100 machines

<table>
<thead>
<tr>
<th>Machines</th>
<th>SAP-LDA data throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 machines</td>
<td>58.3 M/s (1x)</td>
</tr>
<tr>
<td>50 machines</td>
<td>114 M/s (1.96x)</td>
</tr>
<tr>
<td>100 machines</td>
<td>204 M/s (3.5x)</td>
</tr>
</tbody>
</table>
SAP-LDA performance [blinded, to appear]

YahooLDA progress per iteration

80GB data, 2M words, 1K topics, 100 machines

<table>
<thead>
<tr>
<th>Iterations</th>
<th>YahooLDA data throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 machines</td>
<td>39.7 M/s (1x)</td>
</tr>
<tr>
<td>50 machines</td>
<td>78 M/s (1.96x)</td>
</tr>
<tr>
<td>100 machines</td>
<td>151 M/s (3.8x)</td>
</tr>
</tbody>
</table>

- YahooLDA attains near-ideal throughput (1 → 3.8x)…
- … but progress per iteration gets worse with more machines
- YahooLDA only <2x speedup from 25 → 100 machines
  - 6.7x slower compared to SAP-LDA
How to SAP-Lasso

Model Parallel Lasso

- Compute $M$-by-$M$ correlation matrix $X^T X$
  - Group indices $(i,j)$ with high correlation $(X^T)_i X_j > \rho$
  - Load-balance different-sized groups to parallel workers
SAP-Lasso Performance

- Even for small # machines (4)...
- ... 2x difference between SAP and non-SAP Lasso

6.4GB data, 500M features, 4 machines

![Graph showing 2x speedup between SAP-Lasso and Non-SAP Lasso](image)
Improving SAP through Prioritization
[Lee et al., 2014]

- Choose next param to update via convergence speed
  - Lasso: sample params proportional to recent change
  - Approximately maximizes convergence progress per iteration

Shotgun [Bradley et al. 2011]

Priority-based scheduling

\[ p(j) \propto \left( \delta x_j^{(t-1)} \right)^2 + \epsilon \]
Prioritized SAP-Lasso vs non-SAP Lasso

- Prioritized SAP-Lasso convergences much faster than non-SAP Lasso (Shotgun algorithm)

Graph showing convergence over time:

- Objective vs Seconds
- Prioritized SAP-Lasso
- Shotgun scheduling [Bradley et al. 2011]

Initialization phase to collect info on Convergence progress

Mathematical expressions:

\[ p(j) \sim \left( \delta \beta_j^{(t-1)} \right)^2 + \rho \]
Principles of ML system Design [Xing et al., to appear 2016]

2. How to Bridge Computation and Communication: Bridging Models and Bounded Asynchrony
The Bulk Synchronous Parallel Bridging Model [Valiant & McColl]

- Perform barrier in order to communicate parameters
- Mimics sequential computation – “serializable” property
- Enjoys same theoretical guarantees as sequential execution
The success of the von Neumann model of sequential computation is attributable to the fact it is an efficient bridge between software and hardware… an analogous bridge is required for parallel computation if that is to become as widely used – Leslie G. Valiant

- Numerous implementations since 90s (list by Bill McColl):
  - Oxford BSP Toolset (‘98), Paderborn University BSP Library (‘01), Bulk Synchronous Parallel ML (‘03), BSPonMPI (‘06), ScientificPython (‘07), Apache Hama (‘08), Apache Pregel (‘09), MulticoreBSP (‘11), BSPedupack (‘11), Apache Giraph (‘11), GoldenOrb (‘11), Stanford GPS Project (‘11) …
But There Is No Ideal Distributed System!

- **Two distributed challenges:**
  - Networks are slow
  - "Identical" machines rarely perform equally

**Result:** BSP barriers can be slow

---

Compute vs Network
LDA 32 machines (256 cores)

- Network waiting time
- Compute time

---

Unequal performance
Low bandwidth, High delay
Is there a better way to interleave computation and communication?

- Safe/slow (BSP) vs. 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?
A Stale Synchronous Parallel Bridging Model [Ho et al., 2013]

Stale Synchronous Parallel (SSP)

- Fastest/slowest workers not allowed to drift $>s$ iterations apart

Consequence

- Fast like async, yet correct like BSP
- Why? Workers’ local view of model parameters “not too stale” ($\leq s$ iterations old)
Data-Parallel Proximal Gradient under SSP

- Model (e.g. SVM, Lasso ...):
  \[
  \min_{a \in \mathbb{R}^d} \mathcal{L}(a, D), \quad \text{where } \mathcal{L}(a, D) = f(a, D) + g(a)
  \]
  Input $D$, model $a$

- Algorithm:
  - Update
    \[
    a(t) := \text{prox}_g \left( a^p(t) - \eta(t) \sum_{(p', t') \in \text{Recv}(t)} \Delta(a^{p'}(t'), D_{p'}) \right)
    \]
  - sub-update
    \[
    \Delta(a^p(t), D_p) := \nabla f(a^p(t), D_p)
    \]
  - proximal step wrt $g$
  - stale sub-updates $\Delta()$ received by worker $p$ at iteration $t$
  - gradient step wrt $f$

- Data parallel:
  - Data $D$ too large to fit in a single worker, divide among $P$ workers

\[\]
SSP Data-Parallel
Async Speed, BSP Guarantee

- Massive Data Parallelism
- Effective across different algorithms
SSP Data Parallel Convergence Theorem

[Ho et al., 2013, Dai et al., 2015]

Let observed staleness be $\gamma_t$

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

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

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

where

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

Explanation: the distance between true optima and current estimate decreases exponentially with more SSP iterations. Lower staleness mean, variance $\mu_\gamma, \sigma_\gamma$ improve the convergence rate.
Model-Parallel Proximal Gradient under SSP

- Model (e.g. SVM, Lasso ...):
  \[
  \min_{a \in \mathbb{R}^d} \mathcal{L}(a, D), \quad \text{where} \quad \mathcal{L}(a, D) = f(a, D) + g(a)
  \]
  data \( D \), model \( a \)

- Model parallel
  - Model dimension \( d \) too large to fit in a single worker
  - Divide model among \( P \) workers \( a = (a_1, a_2, \ldots, a_P) \)

- Algorithm:
  \[
  \forall p, \quad a_p(t + 1) = a_p(t) + \gamma_p(t) \cdot F_p(a^p(t))
  \]
  \[
  = a_p(0) + \sum_{k=0}^{t} \gamma_p(k) \cdot F_p(a^p(t))
  \]
  (local) \( a^p(t) = (a_1(\tau^p_1(t)), \ldots, a_P(\tau^p_P(t))) \)
  (global) \( a(t) = (a_1(t), \ldots, a_P(t)) \)

  \[
  a^p(t+1) := F_p(a^p(t)) = \text{prox}_{g_p}^p \left( a_p(t) - \eta \nabla_p f(a^p(t)) \right) - a_p(t)
  \]

- worker \( p \) keeps local copy of the full model (can be avoided for linear models)
SSP Model-Parallel
Async Speed, BSP Guarantee

Lasso: 1M samples, 100M features, 100 machines

- Massive Model Parallelism
- Effective across different algorithms
SSP Model Parallel Convergence Theorem
[Zhou et al., to appear 2016]

**Theorem:** Given that the SSP delay is bounded, with appropriate step size and under mild technical conditions, then

$$\sum_{t=0}^{\infty} \|a(t+1) - a(t)\| < \infty \quad \sum_{t=0}^{\infty} \|a^p(t+1) - a^p(t)\| < \infty$$

In particular, the global and local sequences converge to the same critical point, with rate $O(t^{-1})$:

$$\mathcal{L}\left(\frac{1}{t} \sum_{k=1}^{t} a(k)\right) - \inf \mathcal{L} \leq O\left(t^{-1}\right)$$

**Explanation:** Finite length guarantees that the algorithm stops (the updates must eventually go to zero). Furthermore, the algorithm converges at rate $O(t^{-1})$ to the optimal value; same as BSP model parallel.
Principles of ML system Design [Xing et al., to appear 2016]

3. How to Communicate:
*Managed Communication and Topologies*
Managed Communication [Wei et al., 2015]

- SSP only
  - Communicates only at iteration boundary
  - Ensures bounded staleness consistency

- SSP + Managed Communication
  - Continuous communication/synchronization
  - Update prioritization
  - Same consistency guarantees as SSP
MatrixFact: Managed Communication Speedup

- Matrix Factorization, Netflix data, rank = 400
- 8 machines * 16 cores, 1GbE ethernet
LDA: Managed Communication Speedup

- Latent Dirichlet Allocation, NYTimes, # topics = 1000,
- 16 machines * 16 cores, 1GbE ethernet

Already enjoying
SSP speedup

3x additional speed up from comms management
25% additional speedup from comms prioritization
Managed Communication: Wait-free back-propagation [Zhang et al., 2015]

- Distributed wait-free back-propagation for Deep Learning
  - Exploit the chain rule in BP
  - Overlap comms/compute every layer
    - Send out gradients once it’s ready
    - Sync updated params once it’s ready
    - Wait-free!
Managed Communication: Wait-free back-propagation

- Effective for today’s CNN structures

### Params/FLOP distribution of modern CNNs

<table>
<thead>
<tr>
<th></th>
<th>CONV Layers (#/%)</th>
<th>FC Layers (#/%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>2.3M / 3.75</td>
<td>59M / 96.25</td>
</tr>
<tr>
<td>VGG-16</td>
<td>7.15M / 5.58</td>
<td>121.1M / 94.42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CONV Layers (#/%)</th>
<th>FC Layers (#/%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>1,352M / 92.0</td>
<td>117M / 8.0</td>
</tr>
<tr>
<td>VGG-16</td>
<td>10,937M / 91.3</td>
<td>121.1M / 8.7</td>
</tr>
</tbody>
</table>

- 90% computation happens at bottom (Conv) layers
- 90% communication happens at top (FC) layers
- Which means… 90% are overlapped with 90%!
Parameter Storage Paradigms

Centralized Storage

Server

Worker

Send change $\Delta W$

Send $W$ itself

Decentralized Storage

Worker

Send change $\Delta W$

Send change $\Delta W$

Worker

- **Centralized**: send parameter $W$ itself from server to worker
  - Advantage: allows compact comms topology, e.g. bipartite
- **Decentralized**: always send changes $\Delta W$ between workers
  - Advantage: can exploit special structure of $\Delta W$ to reduce comms
Topology: Master-Slave

- Used with centralized storage paradigm
- Topology = bipartite graph: Servers (masters) to Workers (slaves)
- Disadvantage: need to code/manage clients and servers separately
- Advantage: bipartite topology far smaller than full $N^2$ P2P connections
Topology: Peer-to-Peer (P2P)

- Used with **decentralized storage** paradigm
- Workers update local parameter view by broadcasting/receiving
- **Disadvantage:** expensive unless updates $\Delta W$ are lightweight; expensive for large # of workers
- **Advantage:** only need worker code (no central server code); if $\Delta W$ is low rank, comms reduction possible
Halton Sequence Topology [Li et al., 2015]

- Used with **decentralized storage** paradigm
- Like P2P topology, but route messages through many workers
  - e.g. to send message from 1 to 6, use 1->2->3->6
- **Disadvantage**: incur higher SSP staleness due to routing, e.g. 1->2->3->6 = staleness 3
- **Advantage**: support bigger messages; support more machines than P2P topology
Principles of ML system Design [Xing et al., to appear 2016]

4. What to Communicate: *Exploiting Structure in ML Updates*
Matrix-Parameterized Models (MPMs)

\[
\min_W \frac{1}{N} \sum_{i=1}^{N} f_i(Wa_i; b_i) + h(W)
\]

Matrix parameter W

Loss function

Regularizer

Distance Metric Learning, Sparse Coding, Distance Metric Learning, Group Lasso, Neural Network, etc.
Big MPMs

Multiclass Logistic Regression on Wikipedia
- Feature dim. = 20K
- #classes=325K
- 6.5B

Distance Metric Learning on ImageNet
- Feature dim. = 172K
- Latent dim. = 50K
- 8.6B

Sparse Coding on ImageNet
- Feature dim. = 172K
- Dic. Size=50K
- 8.6B

Neural Network of Google Brain
- #neurons in layer 0 = 40K
- #neurons in layer 1 = 33K
- 1.3B

Billions of params = 10-100 GBs, costly network synchronization
What do we actually need to communicate?
Full Updates

- Let matrix parameters be $W$. **Need to send parallel worker updates** $\Delta W$ to other machines…
  - Primal stochastic gradient descent (SGD)
    \[ \min_w \frac{1}{N} \sum_{i=1}^{N} f_i(Wa_i; b_i) + h(W) \]
    \[ \Delta W = \frac{\partial f(Wa_i, b_i)}{\partial W} \]
  - Stochastic dual coordinate ascent (SDCA)
    \[ \min_z \frac{1}{N} \sum_{i=1}^{N} f_i^*(-z_i) + h^*(\frac{1}{N} ZA^T) \]
    \[ \Delta W = (\Delta z_i)a_i \]
Sufficient Factor (SF) Updates
[Xie et al., 2015]

- **Full parameter matrix update** $\Delta W$ can be computed as **outer product of two vectors** $uv^T$ (called sufficient factors)
  - Primal stochastic gradient descent (SGD)
    \[
    \min_W \frac{1}{N} \sum_{i=1}^{N} f_i(Wa_i, b_i) + h(W)
    \]
    \[
    \Delta W = uv^T \quad u = \frac{\partial f(Wa_i, b_i)}{\partial (Wa_i)} \quad v = a_i
    \]
  - Stochastic dual coordinate ascent (SDCA)
    \[
    \min_z \frac{1}{N} \sum_{i=1}^{N} f_i^*(-z_i) + h^*(-\frac{1}{N}ZA^T)
    \]
    \[
    \Delta W = uv^T \quad u = \Delta z_i \quad v = a_i
    \]
- Send the lightweight SF updates $(u, v)$, instead of the expensive full-matrix $\Delta W$ updates!
P2P Topology + SF Updates = Sufficient Factor Broadcasting
SFB Convergence Theorem

[Xie et al., 2015]

**Theorem 1.** Let \( \{W^c_p\}, p = 1, \ldots, P, \) and \( \{W^c\} \) be the local sequences and the auxiliary sequence generated by SFB for problem \( (P) \) (with \( h = 0 \)), respectively. Under Assumption 1 and set the learning rate \( \eta_c^{-1} = \frac{L_c}{2} + 2sL + \sqrt{c} \), then we have

- \( \lim \inf_{c \to \infty} \mathbb{E}\|\nabla F(W^c)\| = 0 \), hence there exists a subsequence of \( \nabla F(W^c) \) that almost surely vanishes;
- \( \lim \max_p \|W^c - W^c_p\| = 0 \), i.e. the maximal disagreement between all local sequences and the auxiliary sequence converges to 0 (almost surely);
- There exists a common subsequence of \( \{W^c_p\} \) and \( \{W^c\} \) that converges almost surely to a stationary point of \( F \), with the rate \( \min_{c \leq C} \mathbb{E}\|\sum_{p=1}^P \nabla F_p(W^c_p)\|_2^2 \leq O \left( \frac{(L+L_F)\sigma^2 Ps \log C}{\sqrt{C}} \right) \).

**Explanation:** Parameter copies \( W_p \) on different workers \( p \) converge to the same optima, i.e. all workers reach the same (correct) answer.

- ✔ Does not need central parameter server or key-value store
- ✔ Works with SSP bridging model (staleness = \( s \))
SF: Convergence Speedup

- Convergence time versus model size, under BSP
- FMS = full matrix updates; SFB = sufficient factor updates
SF: Comm.-Time Reduction

- Computation vs network waiting time
- FMS = full matrix updates; SFB = sufficient factor updates
Hybrid Full Updates + SFB

- Hybrid communications: Parameter Server + Sufficient Factor Broadcasting
  - Parameter Server: Master-Slave topology
  - Sufficient factor broadcasting: P2P topology

- For problems with a mix of large and small matrices,
  - Send small matrices via PS
  - Send large matrices via SFB
Hybrid example: CNN [Zhang et al., 2015]

- AlexNet CNN model
  - Final layers = 4096 * 4096 matrix (17M parameters)
  - Use SF to communicate
    - 1. Decouple into two 4096 vectors: u, v
    - 2. Transmit two vectors
    - 3. Reconstruct the gradient matrix
Hybrid example: CNN [Zhang et al., 2015]

- AlexNet CNN model
  - Convolutional layers = e.g. 11 * 11 matrix (121 parameters)
  - Use Full-matrix updates to communicate
    - 1. Send/receive using Master-Slave PS topology
Summary

1. **How to Distribute?**
   - Structure-Aware Parallelization
   - Work Prioritization

2. **How to Bridge Computation and Communication?**
   - BSP Bridging Model
   - SSP Bridging Model for Data and Model Parallel

3. **How to Communicate?**
   - Managed comms – interleave comms/compute, prioritized comms
   - Parameter Storage: Centralized vs Decentralized
   - Communication Topologies: Master-Slave, P2P, Halton Sequence

4. **What to Communicate?**
   - Full Matrix (FM) updates
   - Sufficient Factor (SF) updates
   - Hybrid FM+SF updates
In Closing: A Distributed Framework for Machine Learning
The Petuum Architecture (50,000 feet view)

ML application library

Data-Parallel API

Bösen Data-Parallel Engine

Parameter Tuning

Model-Parallel API

Strads Model-Parallel Engine

big data storage & transform engine

YARN (resource manager, fault tolerance)

HDFS (distributed storage)

Stand-alone cluster operation
Major Releases
(petuum.org)

- Dec 2013: Petuum 0.1
  - Initial release
  - Apps: LDA, matrix factorization
  - System: Bosen (parameter server)
- March 2014: Petuum 0.2
  - Apps: LDA, matrix factorization, Lasso
  - System: Strads (model-parallel scheduler)
- July, 2014: Petuum 0.9
  - Apps: LDA, matrix factorization, Lasso, Logistic Regression
  - System: large performance improvements
  - Patch releases 0.91 (July 2014), 0.92 (Sept 2014), 0.93 (Dec 2014)
- Jan 2015: Petuum 1.0
  - Many new Apps: MedLDA, NMF, CNN, DML, DNN, DNN speech, Kmeans, MLR, Random forest, Sparse coding
  - System: more performance improvements
- July 2015: Petuum 1.1
  - New Apps: Distributed+GPU CNN, SVM
  - Big Data Ecosystem Support: Java parameter server (JBosen), HDFS, YARN
Petuum Speed Advantage

Topic Detection Speed
On 128 machines

Spark 1x speed

Yahoo 12x speed

Petuum 100x speed
Petuum Size Advantage

100x more scale-up than competitors
Petuum Open Source ML apps

- **ML Library (Petuum v1.1):**
  - Topic Modeling
    - LDA
    - MedLDA (supervised topic models)
  - Deep Learning
    - Fully-connected DNN
    - Convolutional Neural Network (Poseidon)
  - 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
  - HDFS+YARN support
  - Distributed GPU support (for CNN)
  - C++/Java bounded-async key-value store (Bösen)
  - C++ scheduler system (Strads)
CNN: Wait-free backprop speedup

Up to 5x speedup with 8 GPU-equipped machines
Platform vs Tower

- **Platform Approach**
  - Use ideas to build platform that supports many apps
  - Upgrades to platform improve all apps

- **Tower Approach**
  - Build ideas directly into application
  - Best performance; harder to re-use for other apps
Acknowledgements

Garth Gibson
Greg Ganger
Phillip Gibbons
James Cipar