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

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
Massive Data

1B+ USERS
30+ PETABYTES

32 million pages

100+ hours video uploaded every minute

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

Processing power/speed vs. Number of “Machines”

- Perfect
- Pathetic
- Good!
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, \theta)
    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**

**Programming model**

Shotgun with 4 machines flies away!  
Shotgun with 2 machines  
Single machine (shooting algorithm)
How about this …

[Xing et al., 2015]
An ML Program

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

Model  Data  Parameter

Solved by an iterative convergent algorithm

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

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

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

Source: Cisco Global Cloud Index

Source: The Connectivist
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

\[ z_{di} \sim p(z_{di} = k | \text{rest}) \propto (n_{kd}^{-di} + \alpha_k) \cdot \frac{n_{kw}^{-di} + \beta_w}{(n_k^{-di} + \beta V)} \]
Algorithmic Accelerations:

- **Optimization Algorithms**
  - Stochastic gradient descent
  - Coordinate descent
  - Proximal gradient methods --- when $\mathcal{L}$ is not differentiable
    - ISTA, FASTA, Smoothing proximal gradient
  - Proximal average --- complex compound regularizers
  - ADMM --- overlapping constraints
  - …

- **Markov Chain Monte Carlo Algorithms**
  - Aliases samplers (constant time high-dimensional sampler)
  - Auxiliary variable methods (inverse Rao-Blackwellization)
  - Embarrassingly Parallel MCMC (sub-posteriors)
  - Parallel Gibbs Sampling
    - Data parallel
    - Model parallel
Parallelization Strategies

Usually, we worry …

A sequential program

\[
\begin{align*}
\beta_1 \\
\downarrow \\
\beta_2
\end{align*}
\]

A parallel program

\[
\begin{align*}
\beta_1 \\
\downarrow \\
\beta_2 \\
\downarrow \\
\beta_1 \\
\downarrow \\
\beta_2
\end{align*}
\]

Sync

\[\text{for } (t = 1 \text{ to } T) \{ \text{doThings()} \text{ parallelUpdate}(x, \theta) \text{ doOtherThings()} \}\]

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

Unequal performance

Low bandwidth, High delay
ML Computation vs. Classical Computing Programs

ML Program:
- optimization-centric and
- iterative convergent

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

Traditional Program:
- operation-centric and
- deterministic
Traditional Data Processing needs operational correctness

Example: Merge sort

1 6
↓   ↓
1 6
↓
1 3 6 7

7 3
↓   ↓
3 7
↓
4 5

5 4
↓   ↓
4 5 2 8

8 2
↓
2 8

Error persists and is not corrected

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

for (t = 1 to T) {
    doThings()
    \( \bar{\theta}^{t+1} = g(\bar{\theta}^t, \Delta_f \bar{\theta}(\mathcal{D})) \)
    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.

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

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

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

Data Parallel

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

Model Parallel

\[ \tilde{\theta} \equiv [\tilde{\theta}_1^T, \tilde{\theta}_2^T, \ldots, \tilde{\theta}_k^T]^T \]

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

\[ \tilde{\theta}_i \not\perp \tilde{\theta}_j \mid D, \exists(i, j) \]
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 $X\beta$: $\sum_{i=1}^{N} \|y_i - X_i\beta\|_2^2$

- **Regularization $\Omega(\theta)$**
  - $L_1$ 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
Data-Parallel Lasso

Proximal SGD:

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

Coordinate Descent:
Probabilistic Example:

**Topic Models**

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

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

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

\[
\sum_{i=1}^{N} \ln \mathbb{P}_{\text{Dirichlet}}(\delta_i | \alpha) + \sum_{k=1}^{K} \ln \mathbb{P}_{\text{Dirichlet}}(B_k | \beta)
\]

- \( \alpha, \beta \) are “hyperparameters” that control the Dirichlet prior’s strength

- **Algorithm**
  - Collapsed Gibbs Sampling
Data Parallel Gibbs

Global shared model

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

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

Need a System Interface for Parallel ML
– Does ML really Stop at the Ideal View?

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

Lasso via coordinate descent:

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

- A huge number of parameters (e.g.) $M > 100$ million
- How to correctly divide computational workload across workers?
- What is the best order to update parameters?
Model Dependencies

- Concurrent updates of $\beta$ may induce errors

Sequential updates

$\beta_1 \rightarrow \beta_2$

Concurrent updates

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

Sync

Need to check $x_1^Tx_2$ 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 Structure-Aware Parallelization (SAP)

[Lee et al., 2014] [Kim et al., 2016]

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

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

```python
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.
    ...
}
```
A Structure-aware Dynamic Scheduler (Strads) [Lee et al., 2014] [Kim et al, 2016]

Strads System

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

• Priority Scheduling

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

• Block scheduling

[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
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
  - $\rho = \text{Spectral Radius} \left[ \text{BlockDiag} \left[ (X^{(1)})^T X^{(1)}, \ldots, (X^{(t)})^T X^{(t)} \right] \right]$; this block-diagonal matrix quantifies max level of correlation within all SAP blocks $X^{(1)}, X^{(2)}, \ldots, X^{(t)}$

- **SAP converges according to**

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

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 2016]

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
Correctly Measuring Parallel Performance [blinded, to appear]

SAP-LDA progress per iteration

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</td>
<td>58.3 M/s (1x)</td>
</tr>
<tr>
<td>50</td>
<td>114 M/s (1.96x)</td>
</tr>
<tr>
<td>100</td>
<td>204 M/s (3.5x)</td>
</tr>
</tbody>
</table>

- **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
Correctly Measuring Parallel Performance [blinded, to appear]

YahooLDA progress per iteration

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

- 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
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 Bulk Synchronous Parallel Bridging Model

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) …

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
But There Is No Ideal Distributed System!

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

Result: BSP barriers can be slow

Unequal performance

Low bandwidth, High delay

Compute vs Network
LDA 32 machines (256 cores)

Network waiting time
Compute time
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 ...):

\[
\min_{a \in \mathbb{R}^d} \mathcal{L}(a, D), \text{ where } \mathcal{L}(a, D) = f(a, D) + g(a)
\]

\(\text{data } D, \text{ model } a\)

- Algorithm:
  - Update

\[
a(t) := \text{prox}_g \left( a^p(t) - \eta(t) \sum_{(p',t') \in \text{Recv}(t)} \Delta(a^p(t'), D_{p'}) \right)
\]

\(\text{proximal step wrt } g\)

- sub-update

\[
\Delta(a^p(t), D_p) := \nabla f(a^p(t), D_p)
\]

\(\text{gradient step wrt } f\)

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

- Diagram:

  - **Split**
  - **Input Data**
  - **Update local copy of ALL params**
  - **Aggregate**
  - **Update ALL params**
SSP Data-Parallel
Async Speed, BSP Guarantee

- Massive Data Parallelism
- Effective across different algorithms
Theorem: Given L-Lipschitz objective $f_t$ and step size $h_t$,

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

where

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

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:
  \[
  \forall p, \quad a_p(t + 1) = a_p(t) + \gamma_p(t) \cdot F_p(a_p(t))
  \]
  on worker \(p\)
  \[
  = a_p(0) + \sum_{k=0}^t \gamma_p(k) \cdot F_p(a_p(t))
  \]
  workers can skip updates
  (local) \(a_p(t) = (a_1(\tau_1^p(t)), \ldots, a_P(\tau_P^p(t)))\)
  (global) \(a(t) = (a_1(t), \ldots, a_P(t))\).

\[
a_p(t+1) := F_p(a_p(t)) = \text{prox}_{\eta g_p}^p(a_p(t) - \eta \nabla_p f(a_p(t))) - a_p(t)
\]
  gradient step wrt \(f\)
  proximal step wrt \(g\)

- 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

Curves overlap – no compromise to quality
SSP Model Parallel Convergence Theorem

[Zhou et al., 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

Lower is better

Already enjoying SSP speedup

Further 1.8x speedup multiplier over SSP

 SSP
 SSP + Managed Comms

1.8x

training loss

time (seconds)

1e+07
1e+08
1e+09
**LDA:**

Managed Communication Speedup

Already enjoying

SSP speedup

- SSP
- SSP + MC (no prio.)
- SSP + Managed Comms

- 3x additional speed up from comms management
- 25% additional speedup from comms prioritization

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

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

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

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

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 the outer product of two vectors $uv^T$ (called sufficient factors).
- Primal stochastic gradient descent (SGD)

$$\min_{w} \frac{1}{N} \sum_{i=1}^{N} f_i(Wa_i; b_i) + h(W)$$

$$\Delta W = uv^T \quad u = \frac{\partial f(Wa_i, b_i)}{\partial (W)} \quad v = a_i$$

- 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
**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 \( h \equiv 0 \)), respectively. Under Assumption 1 and set the learning rate \( \eta^{-1} = \frac{L_F}{2s} + 2sL + \sqrt{c} \), then we have

- \( \liminf_{c \to \infty} \mathbb{E}\|\nabla F(W^c)\| = 0 \), 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
  \[
  \min_{c \leq C} \mathbb{E}\left\| \sum_{p=1}^P \nabla F_p(W^c_p) \right\|_2^2 \leq O \left( \frac{(L+L_F)\sigma^2 P s \log C}{\sqrt{c}} \right).
  \]

**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
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 updates
   - Sufficient Factor updates
   - Hybrid FM+SF updates (as in a DL model)
In Closing: A Distributed Framework for Machine Learning
ML computation can be handled more effectively and economically on a different system architecture

- **ML Algorithm behavior is different from traditional computing**

- **Flexible and does not need traditional database-style precision**

- **Opportunity for dynamic resource reclamation (CPU, GPU, disk, network)**

- **Intelligently-designed workhorse engines can be shared across many ML algorithms**

- **Existing approaches can’t take advantage of different AI & ML behavior**
  - Traditional platforms specialize at supporting database-style workload, incurring expensive error-recovery and network overheads
  - Traditional platforms do not perform dynamic resource allocation for fast-completing workloads, wasting CPU ops
  - Traditional platforms do not provide sharable workhorse engines, so each vertical application must be developed separately
The Petuum Architecture (50,000 feet view)

- ML application library
  - Data-Parallel API
    - Bösen Data-Parallel Engine
  - Model-Parallel API
    - Strads Model-Parallel Engine
  - Parameter Tuning

- Big data storage & transform engine
  - YARN (resource manager, fault tolerance)
  - HDFS (distributed storage)

- Distributed Container

- 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

![Graph showing topic detection size vs. number of CPUs required for various systems including PetuumLDA, Peacock (LDA), Li & Smola (LDA), Google PLDA (LDA), YahooLDA (LDA), among others, with PetuumLDA having 100x more scale-up than competitors.]
Acknowledgements