A New Platform for Cloud-based Distributed Machine Learning on Big Data

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, Xun Zheng
James Cipar, Henggang Cui,
and, Phil Gibbons, Greg Ganger, Garth Gibson
The First Encounter of Science with Big Data
Machine Learning
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
Challenge #1
– Massive Data Scale

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

Source: The Connectivist

Source: Cisco Global Cloud Index
Challenge #2
– Gigantic Model Size

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

Source: University of Bonn
Challenge #3
– Inadequate ML library

Classic ML 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 Scalability Challenge

![Graph showing processing power/speed vs. number of processors]

- Perfect
- Pathetic
- Good!
Why need new Big ML systems?

MLer’s view

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

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

![Compute vs Network diagram]

LDA 32 machines (256 cores)
- Network waiting time
- Compute time

Parallelize over worker threads
Share global model parameters via RAM
Why need new Big ML systems?

Systems View:

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

Synchronization model

Programming model

Agonistic of ML properties and objectives in system design

Non-uniform convergence
Dynamic structures
Error tolerance
Why need new Big ML systems?

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

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

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

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

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

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

Machine Learning Models/Algorithms

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

---

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

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

**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

**Hardware:** GPU, flash storage, cloud …
for (t = 1 to T) {
    doThings()
    parallelUpdate(x, θ)
    doOtherThings()
}
Naïve MapReduce not best for ML

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

Image source: dzone.com
ML Computation vs. Classical Computing Programs

ML Program: optimization-centric and iterative convergent

Traditional Program: operation-centric and deterministic
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

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

This computation needs to be parallelized!
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

Error persists and is not corrected
ML Algorithms can Self-heal
A Dichotomy of Data and Model in ML Programs

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

New Model = Old Model + Update(Data)

Data Parallel

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

Model Parallel

\[ \hat{\theta} \equiv [\hat{\theta}_1^T, \hat{\theta}_2^T, \ldots, \hat{\theta}_k^T]^T \]
Data and Model Parallelism

**Data Parallelism**

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

**Model Parallelism**

\[ \theta_i \not\perp \theta_j \mid D, \exists (i, j) \]
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?
Spark: Faster MapR on Data-Parallel

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

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

Source: ebaytechblog.com
GraphLab: Model-Parallel via Graphs

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

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

Source: Low et al. (2010)
A New Framework for Large Scale Parallel Machine Learning (Petuum.org)

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

- **Ready-to-run ML programs**
  - Today: Topic Model (LDA), Deep Learning (DNN), Matrix Factorization (Collaborative Filtering), Lasso & Logistic Regression
  - Soon (Jan 1st): Random Forest, K-means, SVM, Deep Learning (CNN), Distance Metric Learning, Multiclass LR, Sparse Coding, Nonnegative MF, Topic Model (MedLDA)
Petuum Overview

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

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

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

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

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

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

```c
schedule() {
    // Select U vars x[j] to be sent
    // to the workers for updating
    ...
    return (x[i_1], ..., x[i_U])
}
```

```c
push(worker = p, vars = (x[i_1], ..., x[i_U])) {
    // Compute partial update z for U vars x[j]
    // at worker p
    ...
    return z
}
```

```c
pull(workers = [p], vars = (x[i_1], ..., x[i_U]),
     updates = [z]) {
    // Use partial updates z from workers p to
    // update U vars x[j]. sync() is automatic.
    ...
}
```
The Science Behind …

principles, design, and theory

- **Key insight**: ML algos have special properties
  - Error-tolerance, dependency structures, uneven convergence
  - How to harness for faster data/model-parallelism?
There Is No Ideal Distributed System!

- **Two distributed challenges:**
  - Networks are slow
  - “Identical” machines rarely perform equally
How to speed up Data-Parallelism?

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

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

- Need straggler tolerance
  - Slow threads must somehow catch up

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

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

Consequence:
- Asynchronous-like speed, BSP-like ML correctness guarantees
- Guaranteed age bound (staleness) on reads
- Contrast: no-age-guarantee Eventual Consistency seen in Cassandra, Memcached

Convergence Theorem

- **Goal:** minimize convex \( f(x) = \frac{1}{T} \sum_{t=1}^{T} f_t(x) \)
  (Example: Stochastic Gradient)
  - \( L \)-Lipschitz, problem diameter bounded by \( F^2 \)
  - Staleness \( s \), using \( P \) threads across all machines
  - Use step size \( \eta_t = \frac{\sigma}{\sqrt{t}} \) with \( \sigma = \frac{F}{L \sqrt{2(s+1)P}} \)

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

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

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

- Put global parameters in PS
  - Examples:
    - Topic Modeling (MCMC)
      - Topic-word table
    - Matrix Factorization (SGD)
      - Factor matrices L, R
    - Lasso Regression (CD)
      - Coefficients $\beta$
    - PS supports many classes of algorithms
      - Above are just a few examples

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

- Massive Data Parallelism
- Effective across different algorithms
Challenges in Model Parallelism

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

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

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

- Concurrent updates of $\beta$ may induce errors

Sequential updates:

$\beta_1$  
$\beta_2$

Concurrent updates:

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

Sync

Induces parallelization error

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

Need to check $x_1^T x_2$ before updating parameters
How to Model-Parallel?

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

Graph Partition

Random Partition

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

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

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

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

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

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


Priority Scheduling

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

Block scheduling

[Kumar, Beutel, Ho and Xing, Fugue: Slow-worker agnostic distributed learning, AISTATS 2014]
SAP versus Naive partitioning

100M features
9 machines

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

\[ p(j) \sim \text{uniform} \]

STRADS Initialization

STRADS
Lasso–RR

Sharp drop
Theoretical Guarantees on SAP

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

$$\mathcal{L} \leq \mathbb{E}_\mathcal{P} \left[ F(\beta^{(t)}) - F(\beta^{(t)} + \Delta \beta^{(t)}) \right].$$ (1)

**Theorem 2** The Block-scheduler updates and the exact gradient descent updates converge to the same set of limit points asymptotically given that the noise terms form a martingale difference sequence. Furthermore, the intra- and inter-subepoch variance is guaranteed to decrease every iteration.
Faster, Better Convergence

- STRADS+SAP achieves better speed and objective
The Landscape of Big ML

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

*GPU cores*
The Landscape of Big ML

More cores, bigger models

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

- **PetuumLDA (v1)**
- **Petuum (MF)**
- **Peacock (LDA)**
- **Petuum (CNN)**
- **COTS* (CNN)**
- **Li & Smola (LDA)**
- **Google DistBelief (CNN)**
- **GraphLab (MF)**
- **Spark (MF)**
- **Microsoft Adam (CNN)**
- **YahooLDA (LDA)**
- **Google PLDA (LDA)**
- **Caffe* (CNN)**

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

Bigger, more accurate models using reasonable # of cores

- PetuumLDA (v2, LightLDA)
- PetuumLDA (v1)
- Petuum (MF)
- Spark (MF)
- Petuum (CNN)
- Microsoft Adam (CNN)
- Google DistBelief (CNN)
- Google PLDA (LDA)
- YahooLDA (LDA)
- Caffe* (CNN)
Computational efficiency per model parameter

Core-seconds per 1m params on log-scale (lower = more efficient)

LDA: PetuumLDA
LDA: YahooLDA
LDA: Li+Smola PS (large data:model ratio)
LDA: Peacock (10 iters only)
LDA: Google plda+ (100 iters only)

DNN: Petuum (CPU cores)
DNN: Hinton (GPU cores)

CNN: Petuum (CPU)
CNN: Caffe (GPU)
CNN: Google Distbelief (CPU)
CNN: COTS (GPU)
CNN: Microsoft Adam (CPU)

MF: Petuum (10 iters only)
MF: GraphLab (to convergence)
MF: Spark (10 iters only)

Core-for-core, Petuum as efficient (and in some cases much better) on Big ML Models, vs competitors
Latest results from NIPS …
PETUUM: An ML-centric Big-Learning Framework

API, Tools, UI, Libraries
Practitioner (ready-to-run toolbox),
ML Researcher (Matlab-style),
Power User (Low-level API)

Programming Models

BIG-ML Architecture

Resource Allocators
Fault Tolerance
Hadoop Ecosystem

- YARN (cluster resource manager)
- HDFS (distributed storage)
- MapReduce
- HBase
- Hive
- ...
Spark enhances the Hadoop ecosystem...

YARN (cluster resource manager)

HDFS (distributed storage)
Petuum to integrate with, further enhance the Hadoop ecosystem

- High-speed, modern ML apps on huge model sizes

Petuum

Spark

MapReduce

HBase

Hive

YARN (cluster resource manager)

HDFS (distributed storage)
Acknowledgements

www.sailing.cs.cmu.edu

Google  IBM