Big ML Software for Modern ML Algorithms

Qirong Ho^ and Eric P. Xing*

^Institute for Infocomm Research, A*STAR
*Carnegie Mellon University
The First Encounter of Science with Big Data
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
Challenge #2 – Gigantic Model Size

Big Data needs Big Models to extract understanding
But ML models with >1 trillion params also won’t fit!
Challenge #3 – Inadequate ML library

Classic ML algorithms used for decades

K-means
Logistic regression
Decision trees
Naive Bayes

And many more...
Challenge #3 – Inadequate ML library

But new tasks have emerged; demand today’s ML algos

Topic models
make sense of documents

Deep learning
make sense of images, audio

Lasso regression
find significant genes, predict stock market
Challenge #3 – Inadequate ML library

But new tasks have emerged; demand today’s ML algos

Latent space network models
- find communities in networks

Tree ensembles
- better than decision trees

Constrained matrix factorization
- collaborative filtering when negative values just don’t make sense

Where are these new algos in today’s Big Data tools?
Challenge #4 – ML algos iterative-convergent

ML algorithms = “engine” to solve ML models

Markov Chain Monte Carlo

Optimization
Hadoop not suited to iterative 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 later in this tutorial...
Why need new Big ML systems?

**ML practitioner’s view**

- Want correctness, fewer iters to converge
Why need new Big ML systems?

ML practitioner’s view

- Want correctness, fewer iters to converge
- ... but assume an ideal system

```plaintext
for (t = 1 to T) {
    doThings()
    parallelUpdate(x, θ)
    doOtherThings()
}
```
Why need new Big ML systems?

ML practitioner’s view

- Want correctness, fewer iters to converge
- … but assume an ideal system

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

- Oversimplify systems issues
  - e.g. machines perform consistently
  - e.g. can sync parameters any time
Why need new Big ML systems?

**Systems view**

- Want more iters executed per second
Why need new Big ML systems?

**Systems view**

- Want more iters executed per second
- … but assume ML algo is a black box
Why need new Big ML systems?

**Systems view**

- Want more iters executed per second
- … but assume ML algo is a black box
- … or assume ML algo “still works” under different execution models

- Slow-but-correct
  - Bulk Sync. Parallel
- Fast-but-unstable
  - Asynchronous Parallel
Why need new Big ML systems?

**Systems view**

- Want more iters executed per second
- … but assume ML algo is a **black box**
- … or assume ML algo “still works” under different execution models

![Slow-but-correct Bulk Sync. Parallel]

- **Fast-but-unstable Asynchronous Parallel**

- Oversimplify ML issues
  - e.g. assume ML algo “works” without proof
  - e.g. ML algo “easy to rewrite” in chosen abstraction: MapR, vertex program, etc.
Why need new Big ML systems?

**ML practitioner’s view**
- Want correctness, **fewer iters to converge**
- … but assume an **ideal system**

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

- **Oversimplify systems issues**
  - e.g. machines perform consistently
  - e.g. can sync parameters any time

**Systems view**
- Want **more iters executed per second**
- … but assume ML algo is a **black box**
- … or assume ML algo “still works” under different execution models

- **Slow-but-correct**
  - Bulk Sync. Parallel

- **Fast-but-unstable**
  - Asynchronous Parallel

- **Oversimplify ML issues**
  - e.g. assume ML algo “works” without proof
  - e.g. ML algo “easy to rewrite” in chosen abstraction: MapR, vertex program, etc.
Why need new Big ML systems?

**ML practitioner’s view**
- Want correctness, **fewer iters to converge**
- ... but assume an ideal system

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

**Systems view**
- Want **more iters executed per second**
- ... but assume ML algo is a **black box**
- “still works” under different execution models

Alone, neither side has full picture ...

New opportunities exist in the middle!

**Oversimplify ML issues**
- e.g. assume ML algo “works” without proof
- e.g. ML algo “easy to rewrite” in chosen abstraction: MapR, vertex program, etc.

**Oversimplify systems issues**
- e.g. machines perform consistently
- e.g. can sync parameters any time
Solution: An Alg/Sys INTERFACE for Big ML

Modern Machine Learning Models/Algorithms

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

Hardware and infrastructure

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

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

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

**Algorithm:** Parallelizable and stochastic MCMC, VI, Opt, Spectral learning ...

**Representation:** Compact, informative features

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

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

**Programming model & Interface:** High: Toolkits, Med: Matlab/R/Python, Low: C/Java
2 parallelization strategies
2 parallelization strategies

Data Parallel

Model Parallel
2 parallelization strategies

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

New Model = Old Model + Update(Data)

Data Parallel

Model Parallel
2 parallelization strategies

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

New Model = Old Model + Update(Data)

Data Parallel

Model Parallel

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

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

New Model = Old Model + Update(Data)

Data Parallel

Model Parallel

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

\[ \tilde{\theta} \equiv [\tilde{\theta}_1^T, \tilde{\theta}_2^T, \ldots, \tilde{\theta}_k^T]^T \]
Modern ML Parallelism: Topic Models

Source: D. Blei (2012)
Modern ML Parallelism: Topic Models

$$\hat{\theta}^{t+1} = \hat{\theta}^t + \Delta_f \hat{\theta}(D)$$
Modern ML Parallelism: Topic Models

\[ \theta^{t+1} = \theta^t + \Delta_f \theta(D) \]
Modern ML Parallelism: Topic Models

\[ \tilde{\theta}^{t+1} = \tilde{\theta}^t + \Delta_f \tilde{\theta}(D) \]
Modern ML Parallelism: Topic Models

Model (Topics)
- gene 0.04
- dna 0.02
- genetic 0.01
- ...
- life 0.02
- evolve 0.01
- organism 0.01
- ...

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

Data (Docs)
- brain 0.04
- neuron 0.02
- nerve 0.01
- ...
- data 0.02
- number 0.02
- computer 0.01
- ...

Seeking Life's Bare (Genetic) Necessities
- Image caption: "Seeking Life's Bare (Genetic) Necessities."
- Text snippet: "..."
Modern ML Parallelism: Topic Models

Model (Topics)
- gene 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
- ...

Update (MCMC algo)

$$\theta^{t+1} = \theta^t + \Delta f \bar{\theta}(D)$$

Data (Docs)

BIG DATA (billions of docs)
Modern ML Parallelism: Topic Models

Data-parallel strategy for topic models

\[ D \equiv \{D_1, D_2, \ldots, D_n\} \]
Modern ML Parallelism:
Topic Models

Data-parallel strategy for topic models

$\mathcal{D} \equiv \{ \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n \}$

Worker machines with **local** data
Modern ML Parallelism: Topic Models

Data-parallel strategy for topic models

Global shared model

Worker machines with local data
Modern ML Parallelism: Lasso Regression

Feature Matrix
(N samples by M features)
Input

Parameter Vector
(M features)
Output

Response Matrix
(N samples)
Input

Lasso outputs sparse parameter vectors (few non-zeros)
=> Easily find most important features
Modern ML Parallelism: Lasso Regression

\[ \hat{\theta}^{t+1} = \hat{\theta}^t + \Delta_f \hat{\theta}(\mathcal{D}) \]
Modern ML Parallelism: Lasso Regression

Model (Parameter Vector)

\[
\theta^{t+1} = \theta^t + \Delta_f \theta(D)
\]
Modern ML Parallelism: Lasso Regression

Model (Parameter Vector)

Data (Feature + Response Matrices)

\[
\theta^{t+1} = \theta^t + \Delta_f \theta(D)
\]
Modern ML Parallelism: Lasso Regression

Model (Parameter Vector)

Data (Feature + Response Matrices)

Update (CD algo)

\[ \hat{\theta}^{t+1} = \hat{\theta}^t + \Delta_f \hat{\theta}(D) \]
Modern ML Parallelism: Lasso Regression

Model (Parameter Vector)

Update (CD algo)

BIG MODEL (100 billions of params)

Data (Feature + Response Matrices)

\[ \theta^{t+1} = \theta^t + \Delta_f \theta(D) \]
Modern ML Parallelism: Lasso Regression

Model-parallel strategy for Lasso

\[ \Theta \equiv [\mathbf{\theta}_1^T, \mathbf{\theta}_2^T, \ldots, \mathbf{\theta}_k^T]^T \]
Modern ML Parallelism: Lasso Regression

Model-parallel strategy for Lasso

\[ \Delta \theta_1(D), \Delta \theta_2(D), \Delta \theta_3(D), \Delta \theta_k(D) \]

\[ \hat{\theta} = [\hat{\theta}_1^T, \hat{\theta}_2^T, \ldots, \hat{\theta}_k^T]^T \]

Worker machines with local model
Modern ML Parallelism: Lasso Regression

Model-parallel strategy for Lasso

Not as easy as this picture suggests - will see why later

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

Worker machines with local model
A General Picture of ML Iterative Algos

Iterative Algorithm

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

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

*F(*) Aggregate + Transform

\[ \Delta \quad \text{Intermediate Updates} \]

*Model Parameters at iteration (t-1)*
Data Parallelism

\[ \Delta_1 = \Delta(A^{(t-1)}, D_1) \]
\[ \Delta_2 = \Delta(A^{(t-1)}, D_2) \]
\[ \Delta_3 = \Delta(A^{(t-1)}, D_3) \]

Additive Updates

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

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

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

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

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

\[ S_1 \in S \]
\[ S_2 \in S \]
\[ S_3 \in S \]

model parameters not updated in this iteration
Modern ML Parallelism: Deep Neural Networks

Source: University of Bonn
Modern ML Parallelism:
Deep Neural Networks

\[ \tilde{\theta}^{t+1} = \tilde{\theta}^t + \Delta_f \tilde{\theta}(D) \]
Modern ML Parallelism: Deep Neural Networks

Model (edge weights)

\[ \theta^{t+1} = \theta^t + \Delta_f \theta(D) \]
Modern ML Parallelism: Deep Neural Networks

\[
\hat{\theta}^{t+1} = \hat{\theta}^t + \Delta_f \hat{\theta}(D)
\]
Modern ML Parallelism: Deep Neural Networks

\[ \hat{\theta}^{t+1} = \hat{\theta}^t + \Delta_f \hat{\theta}(\mathcal{D}) \]
Modern ML Parallelism: Deep Neural Networks

Data and Model can both be big! Millions of images, Billions of weights
What to do?

Data (images)

Model (edge weights)

Update (backpropagation)

\[ \theta^{t+1} = \theta^t + \Delta_f \theta(D) \]
Modern ML Parallelism: Deep Neural Networks

Data-and-Model-parallel strategy for DNN

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

\[ \bar{\theta} \equiv [\bar{\theta}_1^T, \bar{\theta}_2^T, \ldots, \bar{\theta}_k^T]^T \]
Modern ML Parallelism: Deep Neural Networks

Data-and-Model-parallel strategy for DNN

\(\Delta \theta_1(D_1)\)
\(\Delta \theta_2(D_1)\)
\(\Delta \theta_1(D_2)\)
\(\Delta \theta_2(D_2)\)

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

\(\bar{\theta} \equiv [\bar{\theta}_1^T, \bar{\theta}_2^T, \ldots, \bar{\theta}_k^T]^T\)
Data-and-Model-parallel strategy for DNN

“All-pairs” of data and model chunks

\[ \mathcal{D} \equiv \{ \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n \} \]

\[ \bar{\theta} \equiv [\bar{\theta}_1^T, \bar{\theta}_2^T, \ldots, \bar{\theta}_k^T]^T \]
Modern ML Parallelism: Deep Neural Networks

Data-and-Model-parallel strategy for DNN

“All-pairs” of data and model chunks

$\bar{\theta} \equiv \begin{bmatrix} \bar{\theta}_1^T, \bar{\theta}_2^T, \ldots, \bar{\theta}_k^T \end{bmatrix}^T$

Parameter Synchronization Channel
Is data/model-parallelism that easy?

- Not always - certain 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 - will see why later!
  - NOT equivalent to serial execution of ML algo

- What about software to write data/model-parallel ML easily, quickly?
Modern Systems for Big ML

- Just now: basic ideas of data-, model-parallelism in ML
- What systems allow ML programs to be written, executed this way?
Modern Systems for Big ML

- Just now: basic ideas of data-, model-parallelism in ML
- What systems allow ML programs to be written, executed this way?

[GraphLab
Spark
PETUUM]
Spark Overview

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

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

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

Source: Zaharia et al. (2012)
Spark Overview

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

Source: Zaharia et al. (2012)
Spark Demo in Linux VM

Logistic Regression (Spark Shell)

# Start Spark Shell
cd ~/spark-1.0.2/
bin/spark-shell

// Scala code starts here
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils

// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_linear_regression_data.txt")
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
Spark Demo in Linux VM

Logistic Regression (Spark Shell)

```
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// Clear the default threshold.
model.clearThreshold()

// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
  val score = model.predict(point.features)
  (score, point.label)
}
// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()
println("Area under ROC = " + auROC)

// More info @ https://spark.apache.org/docs/latest/mllib-linear-methods.html
```
GraphLab Overview

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

- Comes with ready-to-run “toolkits”
  - ML-centric toolkits: clustering, collaborative filtering, topic modeling, graphical models
GraphLab Overview

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

Source: Gonzalez (2012)
GraphLab Overview

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

Source: Gonzalez (2012)
GraphLab Overview

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

Source: Gonzalez (2012)
GraphLab Overview

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

Source: Gonzalez (2012)
GraphLab Overview

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

Source: Gonzalez (2012)
GraphLab Demo in Linux VM

Topic Modeling (Linux shell)

```
cd ~/graphlab-master/release/toolkits/topic_modeling

# Run Topic Model on sample dataset, continuous output to screen
./cgs_lda --corpus ./daily_kos/tokens --dictionary ./daily_kos/dictionary.txt --ncpus=4

# Run Topic Model on sample dataset, save output to disk
./cgs_lda --corpus ./daily_kos/tokens --dictionary ./daily_kos/dictionary.txt --ncpus=4 \
  --word_dir word_counts --doc_dir doc_counts --burnin=60

# More info @ http://docs.graphlab.org/topic_modeling.html
```
Petuum Overview

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

- Ready-to-run ML programs
  - Now: Topic Model, DNN, Lasso & Logistic Regression, MF
  - Soon: Tree ensembles, Metric Learning, Network Models, CNN, more
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 **data-parallelism**: model parameters become global
  - Special type of Distributed Shared Memory (DSM)

```
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

- Parameter Server benefits:
  - ML-tailored consistency model: Stale Synchronous Parallel (SSP)
  - Asynchronous-like speed, BSP-like ML correctness guarantees
Petuum Overview

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

```plaintext
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

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

![Diagram showing the flow of parameters, variables, and input data with dynamic re-grouping and dispatching blocks to workers, followed by dynamically revising.]

![Graph showing a sharp drop in Objective over time due to SAP.]

Sharp drop due to SAP
Petuum Demo in Linux VM

Deep Neural Network (Linux shell)

```
cd ~/petuum-release_0.93/apps/dnn

# Generate N=10,000 simulated dataset (args: #samples, #features, #classes, #partitions)
scripts/gen_data.sh 10000 440 1993 3 datasets

# Run 6-layer, 2M-param DNN on simulated dataset (args: #threads_per_machine, staleness)
scripts/run_dnn.sh 4 5 machinefiles/localserver datasets/para_imnet.txt
   datasets/data_ptt_file.txt weights.txt biases.txt

# More info @ https://github.com/petuum/public/wiki/ML-App:-Deep-Neural-Network
```
Science of BigML: Principles, design, theory

- Just saw Spark, GraphLab, Petuum in action

- Each has distinct technical innovations
  - How suited are they to Big ML problems?
  - How do they enable Data, Model-Parallel execution?

- **Key insight**: ML algos have special properties
  - Error-tolerance, dependency structures, uneven convergence
  - How to harness for faster data/model-parallelism?
Refresher: data, model-parallel

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

New Model = Old Model + Update(Data)

Data Parallel

Model Parallel

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

\[ \tilde{\theta} \equiv [\tilde{\theta}_1^T, \tilde{\theta}_2^T, \ldots, \tilde{\theta}_k^T]^T \]
ML algos are iterative-convergent

- “Hill-climbing”
  - Repeat update function until no change
  - True for sequential, as well as data/model-parallel ML algos

- Why are ML algos I-C?
  - Vast majority of ML algos are optimization or MCMC-based (and both are I-C procedures)
Contrast: Non-iterative-convergent

Example: Merge sort

1 6
→ 1 6
→ 1 3 6 7
→ 1 3 4 5 6 7 2 8

7 3
→ 3 7
→ 4 5 2 8

5 4
→ 4 5
→ 4 5 2 8

8 2
→ 2 8
→ 1 3 4 5 6 7 2 8

Sorting error: 2 after 5

Error persists and is not corrected
Why not Hadoop?

- 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!
Properties of I-C ML algos

● (1) “Self-healing” and error-tolerant
  ○ Model parameters a bit wrong => won’t affect final outcome

Topic Models, Lasso Regression, DNNs, all essentially do this:
Properties of I-C ML algs

- (2) Block-structured dependencies
  - Model parameters NOT independent, form blocks
  - Inside a block: should update sequentially
  - Between blocks: safe enough to model-parallelize

Dependency Analysis

Blocks in Lasso Regression problem
Properties of I-C ML algos

- (3) Non-uniform convergence
  - Some model parameters converge much faster!

Pagerank is a famous example.

Also applies to many ML algos, especially Lasso Regression and DNN.
ML Properties vs BigML Platforms

- Data/Model-parallel ML algos are:
  - Iterative-convergent
    - (1) Self-healing/error-tolerant
    - (2) Have block-structured dependencies
    - (3) Exhibit non-uniform convergence

- How do Spark, GraphLab, Petuum fit the above?
Spark: I-C done faster than Hadoop

- Hadoop’s problem: can’t execute many MapR iterations quickly
  - Must write to HDFS after every reduce()

Image source: dzone.com
Spark: I-C done faster than Hadoop

- 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

Source: ebaytechblog.com
Spark: I-C done faster than Hadoop

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

Zaharia et al. (2012)
GraphLab: Model-parallel graphs

- Graph-structured problems really model-parallel
  - Common reason: graph data not IID; data-parallel style poor fit
  - In ML: sparse MatrixFact, some graphical models, pagerank

- How to correctly parallelize graph ML algos?
GraphLab: Model-parallel graphs

- GraphLab **Graph consistency models**
  - Guide search for “ideal” model-parallel execution order
  - ML algo correct if input graph has all dependencies
  - Similar to “block structures”: Graph = “hard” deps, Blocks = “soft”

Source: Low et al. (2010)
GraphLab: Model-parallel graphs

- 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. (2012)
Petuum: ML props = 1st-class citizen

- Idea: ML properties expose new speedup opportunities

- Can data/model-parallel ML run faster if we
  - Allow error in model state? (error-tolerance)
  - Dynamically compute model dependencies? (block structure)
  - Prioritize parts of the model? (non-uniform convergence)
Petuum: ML props = 1st-class citizen

- Error tolerance via Stale Sync Parallel (SSP) Parameter Server (PS)
  - ML Insight 1: old, cached params = small deviation to current params
  - ML Insight 2: deviation strictly limited => ML algo still correct
Petuum: ML props = 1st-class citizen

- Error tolerance via Stale Sync Parallel Parameter Server (PS)
  - System Insight 1: ML algos bottleneck on network comms
  - System Insight 2: More caching => less comms => faster execution
Petuum: ML props = 1st-class citizen

- Harness Block dependency structure via **Scheduler**
  - Model-parallel execution = update many params at same time
  - ML Insight 1: Correctness affected by inter-param dependency
  - ML Insight 2: Even w/o explicit graph, can still compute deps
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

Blocks in Lasso Regression problem

Diagram:
- Worker 1
- Worker 2
- Worker 3
- Worker 4
- Round 1
- Round 2
- Round 3
- Round 4

- Prioritize Params/Vars for update
- Generate Blocks
- Check Variable Dependencies
- All Parameters and Variables

Blocks of variables
Petuum: ML props = 1st-class citizen

- Exploit Uneven Convergence via **Prioritizer**
  - ML Insight 1: “Steepest descent” - progress correlated with last iter
  - ML Insight 2: Complex model deps => params converge at diff rates
Petuum: ML props = 1st-class citizen

- Exploit Uneven Convergence via **Prioritizer**
  - System Insight 1: Prioritize small # of vars => fewer deps to check
  - System Insight 2: Great synergy with **Scheduler**
Open research topics

- Early days for data-, model-parallelism
  - New properties, principles still undiscovered
  - Potential to accelerate ML beyond naive strategies

- Deep analysis of BigML systems limited to few ML algos
  - Need efforts at deeper, foundational level

- Major obstacle: lack common formalism for data/model parallelism
  - Model of ML execution under error due to imperfect system?
  - Model not just “theoretical” ML costs, but also system costs?
No Ideal Distributed System!

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

Unequal performance

Low bandwidth, High delay

![Graph showing compute vs network time](image-url)
Recall 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
\[ \Delta = \sum_{p=1}^{3} \Delta_p \]

\[ A^{(t)} = F(A^{(t-1)}, \Delta) \]
High-Performance Consistency Models for Fast Data-Parallelism

- Recall **Stale Synchronous Parallel (SSP)**
  - Asynchronous-like speed, BSP-like ML correctness guarantees
  - Guaranteed age bound (staleness) on reads
  - C.f.: no-age-guarantee **Eventual Consistency** seen in Cassandra, Memcached

![Graph showing thread updates and staleness threshold]

- **Thread 1** will always see these updates.
- **Thread 1** may not see these updates (limited error).
The BSP-Async dichotomy

- **BSP**
  - Barrier after every iteration; wait for stragglers
  - Since all comms occur at barrier, the barrier itself can be slow

- **Async**
  - No barriers, but risk unlimited straggler duration

- **Want best of BSP (slow, correct) and Async (fast, no guarantees)**
SSP is best-of-both-worlds

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

- **Straggler tolerance**
  - Slow threads can catch up by reducing network comms

```
Thread 1 catches up by reducing network comms

Thread 1 catches up by reducing network comms
```

```
Force threads to sync up

Thread 1
Thread 2
Thread 3
Thread 4
```

```
Time
```
Why Does SSP Converge?

- When a thread reads a parameter, # of “missing updates” is bounded
- Partial, but bounded, loss of serializability
- Hence numeric error in parameter also bounded
SSP Convergence Theorem

- **Goal:** minimize convex  \( f(x) = \frac{1}{T} \sum_{t=1}^{T} f_t(x) \)
  
  (Example: Stochastic Gradient)
  
  - \( L \)-Lipschitz, \( T \) is num iters, problem diameter \( F^2 \)
  
  - 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

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

- Note the RHS interrelation between \((L, F)\) and \((s, P)\)
  
  - An interaction between theory and systems parameters
Eager SSP (ESSP)

- **Better SSP protocol**
  - Use spare bandwidth to push fresh params sooner

- Figure shows difference in stale reads between SSP and ESSP

- ESSP has fewer stale reads; lower variance
ESSP has faster convergence

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

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

  \[
  \leq \exp \left\{ \frac{-T\tau^2}{2\bar{\epsilon}_T \epsilon_v + \frac{2}{3} \sigma L^2 (2s + 1) P \tau} \right\}
  \]

  where

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

  $L$ is a lipschitz constant, and $\epsilon_m$ and $\epsilon_v$ are the mean and variance of the observed staleness.

- **Intuition:** Under ESSP, distance between current param and optimal value decreases exponentially with more iters => guarantees faster convergence than normal SSP.
ESSP has steadier convergence

- Theorem: the variance in the ESSP estimate is

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

where

\[
\text{cov}(\mathbf{v}_1, \mathbf{v}_2) := \mathbb{E}[\mathbf{v}_1^T \mathbf{v}_2] - \mathbb{E}[\mathbf{v}_1^T] \mathbb{E}[\mathbf{v}_2]
\]

\[
\mathcal{O}_{\epsilon_t}^* \text{ represents 5th order or higher terms}
\]

- Intuition: under ESSP, parameter variance decreases near the optimum
- Lower variance => less oscillation in estimate => more confidence in estimate quality and stopping criterion
(E)SSP: Async Speed + BSP Guarantees

- Massive **Data** Parallelism
- Effective across different algorithms

![Graphs showing performance improvements for LDA, Lasso, and Matrix Fact algorithms using (E)SSP](image)
(E)SSP Scales With # Machines

Double # machines:
→ 78% speedup
→ converge in 56% time

(E)SSP: linear scaling with # machines
Model Parallelism

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

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

$\Delta = \{\Delta_p\}$

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

Scheduling Function

$S = S(A^{(t-1)}, D)$

$S_1 \in S$

$S_2 \in S$

$S_3 \in S$

$A^{(t-1)}$

model parameters not updated in this iteration
Challenges in Model Parallelism
Lasso as case study

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

Huge # of parameters, e.g. \( J = 100M \)
Update elements of \( \beta \) in parallel
Model Dependencies in Lasso

- Concurrent updates of $\beta$ may induce errors

Sequential updates

\[
\beta_1 \\
\beta_2
\]

Concurrent updates

\[
\beta_1 \\
\beta_2
\]

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
The model-parallelism dichotomy

- **Ideal:** compute dependency graph, then partition the graph
  - Expensive and not practical; $O(J^2)$ computation just to find all dependencies

- **Naive:** randomly partition
  - Very fast, but risks parallelization error and algo instability/divergence!

- Is there a balanced middle ground?
Structure-Aware Parallelization (SAP) as a middle ground

1. Prioritization
2. Dependency-Checking
3. Load Balancing; Dispatching
SAP for Lasso

- **Prioritizer:**
  \[ \mathcal{U} = \{ \beta_j \} \sim \left( \delta \beta_j^{(t-1)} \right)^2 + \eta \]
  Select a few vars which are changing quickly

- **Dependency checker:**
  \[ |x_j x_k| < \rho \text{ for all } j \neq k \in \mathcal{U} \]
  Very fast - only need to check (few) prioritized vars
  Vars which violate above must be updated sequentially

- **Dispatcher:**
  Construct \( \{ \beta_j \} \) sets according to sequential constraints
  Load-balance \( \{ \beta_j \} \) sets, and dispatch to workers
SAP versus Naive partitioning

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

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

SAP prioritizer

100M features
9 machines

Objective

Sharp drop

Seconds
**Theoretical Guarantee on SAP**

**Guaranteed optimality on Lasso:**

Theorem 1 Suppose $P = \{v_i\}_{i=1}^L$ 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 $i \neq j \in P$, 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 $P$, where $\mathcal{L}$ is defined as

$$\mathcal{L} \leq E_P \left[ F(\beta^{(t)}) - F(\beta^{(t)} + \Delta \beta^{(t)}) \right] .$$

(1)
SAP: Faster, Better Convergence

- SAP achieves better speed and objective
SAP: Scales With # Machines

Increase # of machines; time to reach fixed objective decreases

- STRADS LDA
  - 2.5M vocab, 5K topics
  - Log-Likelihood vs. Seconds
  - 16 machines: Red
  - 32 machines: Dark blue
  - 64 machines: Yellow
  - 128 machines: Black

- Graph shows that as the number of machines increases, the time to reach a fixed objective decreases.
SAP: Bigger Models Now Manageable

- Massive **Model** Parallelism
- Effective across different models

**Lasso**
9 machines

**MF**
9 machines

**LDA**
64 machines
Another model-parallel idea:
Block Scheduling (Gemulla ’11)

Partition data & model into $d \times d$ blocks

This example: $d=3$ strata

In strata: Process blocks with different colors in parallel
Between strata: Process strata sequentially
Fugue: Slow-Worker Agnosticism to solve the straggler problem

- **In real distributed systems:**
  - Slow workers force fast workers to wait

- **Idea: fast workers keep working**
  - Keep updating same block
  - Until slowest worker completes block

- **Strong theoretical guarantees**
  - Faster convergence
  - Lower variance (stable convergence)
Acknowledgements

Jin Kyu Kim  Seunghak Lee  Jinliang Wei  Wei Dai  Pengtao Xie  Xun Zheng  Abhimanu Kumar

Garth Gibson  Greg Ganger  Phillip Gibbons  James Cipar

www.sailing.cs.cmu.edu