An Evaluation of Systems for Scalable Linear Algebra

Anthony Thomas

UC San Diego

July 2, 2018

Introduction

Our Contributions

1. Empirical evaluation of systems focusing on linear algebra (LA) based machine learning
2. Articulate a new set of LA workloads stress testing different data access and communication patterns
3. Extensive empirical comparison of several popular LA systems on real and synthetic data
4. Analysis and discussion of system strengths and weaknesses

Under submission to VLDB 2019
`https://adalabucsd.github.io/slab.html`

Background

- What is linear algebra?
  - Formal mathematical language for describing transformations to matrices
  - Characterizes systems of *linear* equations (in a vector space)
  - Example: $\boldsymbol{Ax} = \boldsymbol{b} \Rightarrow \boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$
- Why should we care?
  - Most common statistics algorithms can be expressed as transformations to matrices
  - Elegant language of abstraction for programming statistical algorithms
  - Algorithms can be expressed in "near math" syntax
  - Loosely analogous to SQL and relational algebra

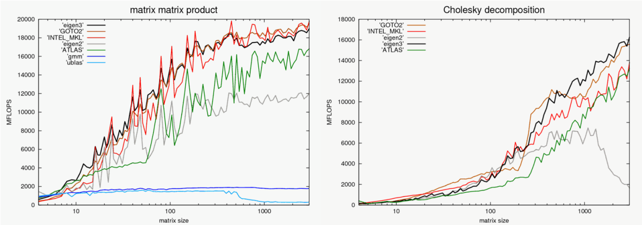
Numerical Linear Algebra (Classical)

LAPACK (C/FORTRAN) 1979/1992	R Language (C) 1993	Eigen (C++) 2009

- Mostly low level libraries and wrappers
- Some work towards scalability (ScaLAPACK)

Yet Another Linear Algebra Benchmark?

- Linear Algebra has been extensively studied, but...
  - Focus mostly on single-node in-memory setting
  - Target low level libraries (BLAS, Eigen, etc...)
  - Goal is to optimize primitive linear algebra operations



Source: `http://eigen.tuxfamily.org/index.php?title=Benchmark`

What's new in Systems for Scalable Linear Algebra?

- Scaling beyond main memory:
  - RDBMS based systems: Apache MADlib, SimSQL, RIOT
  - Map-reduce/Spark: Spark ML/MLlib, Apache SystemML, Mahout Samsara
  - Something new: TensorFlow
- Moving towards declarative programming style:
  - Less painful (not painless) implementation of distributed programs
  - Decoupling physical implementation from program design
  - Hollistic *inter-operator* program optimization

What's new in Systems for Scalable Linear Algebra?

- Scaling beyond main memory:
  - RDBMS based systems: Apache MADlib, SimSQL, RIOT
  - Map-reduce/Spark: Spark ML/MLlib, Apache SystemML, Mahout Samsara
  - Something new: TensorFlow
- Moving towards declarative programming style:
  - Less painful (not painless) implementation of distributed programs
  - Decoupling physical implementation from program design
  - Hollistic *inter-operator* program optimization

Example: Apache SystemML

- Aims to bring SQL style "declarative programming" to machine learning
- Compiles programs written in a custom R-like language (DML) into batch jobs run on Spark.
- Sophisticated program optimization:
  - Physical operator selection (e.g. GMM implementation)
  - Optimization based on LA semantics
  - Automatic choice of dense/sparse matrices, local/distributed computation

Experimental Evaluation

- Performance evaluation targeted at the "typical data science user"
- Focus on **bulk** LA operations - not mini-batch SGD and neural networks
- Two scale factors controlling data complexity:
  1. Number of rows
  2. Data sparsity (% cells which are 0)
- And two scale factors controlling the computational environment:
  1. Number of CPU cores
  2. Number of cluster nodes (implcitly scales RAM)

Experimental Evaluation

- Performance evaluation targeted at the "typical data science user"
- Focus on **bulk** LA operations - not mini-batch SGD and neural networks
- Two scale factors controlling data complexity:
  1. Number of rows
  2. Data sparsity (% cells which are 0)
- And two scale factors controlling the computational environment:
  1. Number of CPU cores
  2. Number of cluster nodes (implcitly scales RAM)

Experimental Evaluation

- Performance evaluation targeted at the "typical data science user"
- Focus on **bulk** LA operations - not mini-batch SGD and neural networks
- Two scale factors controlling data complexity:
  1. Number of rows
  2. Data sparsity (% cells which are 0)
- And two scale factors controlling the computational environment:
  1. Number of CPU cores
  2. Number of cluster nodes (implcitly scales RAM)

Task Categories

Primitive Matrix Operators

- Aggregation Operators: Frobenius Norm, Matrix Vector Multiplication
- Binary Block Operators: Matrix Addition
- Multiplication: General matrix-matrix multiplication, transpose-self multiplication

Pipelines and Decompositions

- Multiplication chains:

$$\underset{N\times 1}{\boldsymbol{p}} = \underset{N\times 1}{\boldsymbol{u}} \cdot \underset{1\times N}{\boldsymbol{v}} \cdot \underset{N\times 1}{\boldsymbol{w}}$$

- Singular Value Decomposition:

$$\mathrm{SVD}(\underset{N\times K}{\boldsymbol{M}}) \rightarrow \underset{N\times K}{\boldsymbol{U}} \cdot \underset{K\times K}{\boldsymbol{\Sigma}} \cdot \underset{K\times K}{\boldsymbol{V}}^T$$

Task Categories - ML Algorithms

OLS Regression solved via normal equations:

Input: $\boldsymbol{X}$ - data, $\boldsymbol{y}$ - outcomes
Result: $\beta = \texttt{solve}(\boldsymbol{X}^T\boldsymbol{X}, \boldsymbol{X}^T\boldsymbol{y})$

Logistic Regression solved via gradient descent:

Input: $\boldsymbol{X}$ - data, $\boldsymbol{y}$ - outcomes
$\beta = \mathrm{rand}(K, 1)$
while *not converged* **do**
$\quad \beta = \beta + \boldsymbol{X}^T(y - \frac{1}{1+\exp(-\boldsymbol{X}\beta)})$
end
Result: β

Task Categories - ML Algorithms

OLS Regression solved via normal equations:

Input: $\boldsymbol{X}$ - data, $\boldsymbol{y}$ - outcomes
Result: $\beta = \texttt{solve}(\boldsymbol{X}^T\boldsymbol{X}, \boldsymbol{X}^T\boldsymbol{y})$

Logistic Regression solved via gradient descent:

Input: $\boldsymbol{X}$ - data, $\boldsymbol{y}$ - outcomes
$\beta = \texttt{rand}(K, 1)$
while *not converged* **do**
  $\beta = \beta + \boldsymbol{X}^T(y - \frac{1}{1+\exp(-\boldsymbol{X}\beta)})$
end
Result: β

Task Categories - ML Algorithms

Non-Negative Matrix Factorization solved via multiplicative updates:

Input: $\boldsymbol{X}$ - data matrix, r - rank
$\boldsymbol{W} = \texttt{rand}(N, r)$
$\boldsymbol{H} = \texttt{rand}(r, K)$
while *not converged* **do**
$\quad \boldsymbol{W} = \boldsymbol{W} \cdot \frac{\boldsymbol{X}\boldsymbol{H}^T}{\boldsymbol{W}\boldsymbol{H}\boldsymbol{H}^T}$
$\quad \boldsymbol{H} = \boldsymbol{H} \cdot \frac{\boldsymbol{W}^T\boldsymbol{X}}{\boldsymbol{W}^T\boldsymbol{W}\boldsymbol{H}}$
end
Result: $(\boldsymbol{H}, \boldsymbol{H})$

Heteroscedasticity Robust Standard Errors solved by White's Method

Input: $\boldsymbol{X}$ - data, ϵ - OLS Residuals
$\mathcal{V} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\texttt{diag}(\epsilon^2)\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}$
Result: $\mathcal{V}$

Task Categories - ML Algorithms

Non-Negative Matrix Factorization solved via multiplicative updates:

Input: $\boldsymbol{X}$ - data matrix, r - rank
$\boldsymbol{W} = \texttt{rand}(N, r)$
$\boldsymbol{H} = \texttt{rand}(r, K)$
while *not converged* **do**
$\quad \boldsymbol{W} = \boldsymbol{W} \cdot \frac{\boldsymbol{X}\boldsymbol{H}^T}{\boldsymbol{W}\boldsymbol{H}\boldsymbol{H}^T}$
$\quad \boldsymbol{H} = \boldsymbol{H} \cdot \frac{\boldsymbol{W}^T\boldsymbol{X}}{\boldsymbol{W}^T\boldsymbol{W}\boldsymbol{H}}$
end
Result: $(\boldsymbol{H}, \boldsymbol{H})$

Heteroscedasticity Robust Standard Errors solved by White's Method

Input: $\boldsymbol{X}$ - data, ϵ - OLS Residuals
$\mathcal{V} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\texttt{diag}(\epsilon^2)\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}$
Result: $\mathcal{V}$

Technical Details

- Systems Compared (Distributed Context)
  - Spark MLLib - Spark based
  - Apache SystemML - Spark based
  - SciDB - Custom Array DBMS
  - Apache MADLib - RDBMS (Greenplum/Postgres) based
  - "Programming with Big Data in R" (pbdR) DMAT - MPI/ScaLAPACK based
- Measurements performed on CloudLab "Clemson" site
  - 200 GB RAM, 24 CPU, 800GB per node
  - Most experiments (and intermediates) fit in distributed RAM
- Each test is run 5 times
  - First measurement is discarded
  - Median, min and max a reported
- Data loading time is **not included**
- Disk spills are allowed

Technical Details

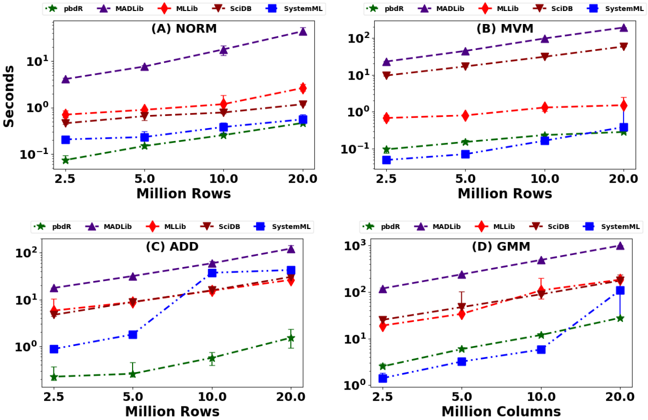
- Systems Compared (Distributed Context)
  - Spark MLLib - Spark based
  - Apache SystemML - Spark based
  - SciDB - Custom Array DBMS
  - Apache MADLib - RDBMS (Greenplum/Postgres) based
  - "Programming with Big Data in R" (pbdR) DMAT - MPI/ScaLAPACK based
- Measurements performed on CloudLab "Clemson" site
  - 200 GB RAM, 24 CPU, 800GB per node
  - Most experiments (and intermediates) fit in distributed RAM
- Each test is run 5 times
  - First measurement is discarded
  - Median, min and max a reported
- Data loading time is **not included**
- Disk spills are allowed

Technical Details

- Systems Compared (Distributed Context)
  - Spark MLLib - Spark based
  - Apache SystemML - Spark based
  - SciDB - Custom Array DBMS
  - Apache MADLib - RDBMS (Greenplum/Postgres) based
  - "Programming with Big Data in R" (pbdR) DMAT - MPI/ScaLAPACK based
- Measurements performed on CloudLab "Clemson" site
  - 200 GB RAM, 24 CPU, 800GB per node
  - Most experiments (and intermediates) fit in distributed RAM
- Each test is run 5 times
  - First measurement is discarded
  - Median, min and max a reported
- Data loading time is **not included**
- Disk spills are allowed

Results!

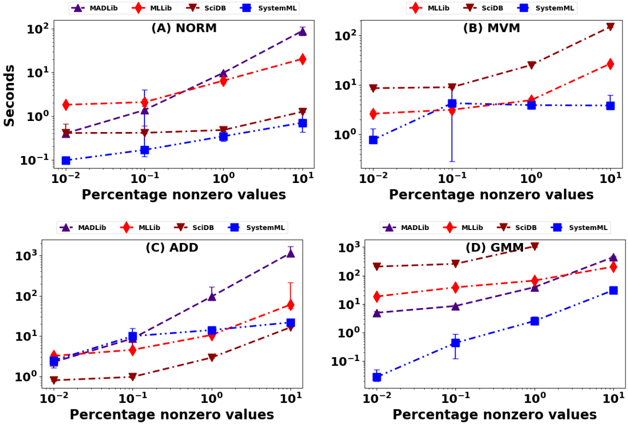
Figure 1: Distributed Dense Matrix Ops - Vary Rows



Cluster size: 8 nodes, Matrix fixed axis: 100, CPUs: 24

What's going on with SystemML?

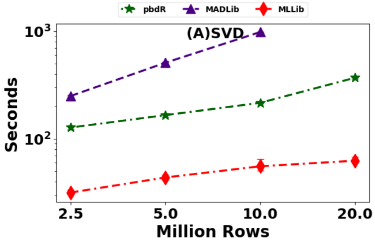
Figure 2: Distributed Sparse Matrix Ops - Vary Sparsity



Cluster size: 8 nodes, CPUs: 24, Logical Matrix Size: 100*GB*

What happened to MADlib for MVM?

Figure 3: Distributed Pipelines and Decompositions



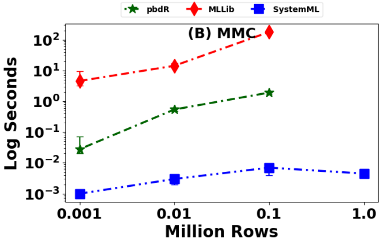
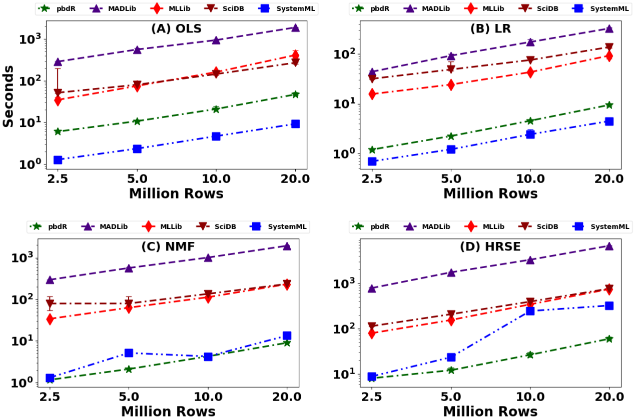
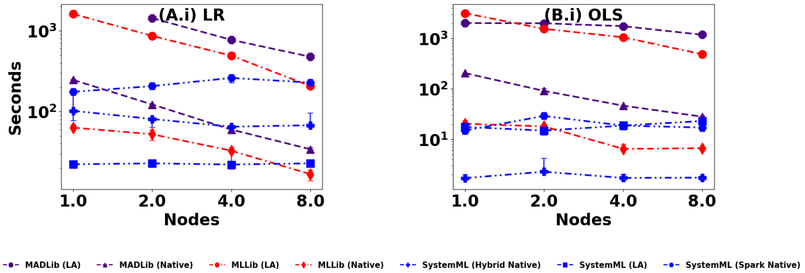


Figure 4: Distributed Dense LA Algorithms - Vary Rows



Implemented by us...

Figure 5: Distributed Dense Algorithms - Criteo Adclick Data



Implemented by us and them...

Comparing LA Abstractions - Example I

Apache SystemML

```
logit = function(matrix[double] X,
                 matrix[double] y,
                 Integer iterations)
    return (matrix[double] w) {

    N = nrow(X)
    w = matrix(0, rows=ncol(X), cols=1)
    iteration = 0
    stepSize = 10

    while (iteration < iterations) {
        xb = X %*% w
        delta = 1/(1+exp(-xb)) - y
        stepSize = stepSize / 2
        w = w - ((stepSize * t(X) %*% delta)/N)

        iteration = iteration+1
    }
}
```

Spark MLLib

```
def logit(X: IndexedRowMatrix, y: IndexedRowMatrix,
          max_iter: Int = 3, sc: SparkContext) : Matrix = {
    var iteration = 0
    var step_size = 0.001
    val N = X.numRows.toInt
    val K = X.numCols.toInt
    var w = Matrices.rand(K, 1, new Random())
    val XT = X.toBlockMatrix(1024,X.numCols.toInt).transpose
    XT.persist(MEMORY_AND_DISK_SER)

    while (iteration < max_iter) {
        println(s"Iteration => ${iteration}")
        val xb = X.multiply( w )
        val gg = new IndexedRowMatrix(
            xb.rows.map(row => new IndexedRow(
                row.index, from_breeze(
                    bNum.sigmoid(as_breeze(row.vector))))
        ))
        val eps = elem_subtract(gg, y).toBlockMatrix(XT.colsPerBlock,1)
        val XTe = XT.multiply( eps, 500 ).toLocalMatrix
        val w_update = (step_size/N.toDouble)*as_breeze( XTe )
        w = from_breeze( as_breeze( w ) :- w_update )
        step_size /= 2.0
        iteration += 1
    }

    return w
}
```

Comparing LA Abstractions - Example II

pbdR

```r
reg <- function(X, y) {
    b <- solve(t(X) %*% X, t(X) %*% y)
    return(b)
}
```

SciDB

```
store(gemm(X, X, Z, transa:true), XTX)
store(gemm(project(
    apply(cross_join(
        transpose(gesvd(XTX, 'VT')) as V,
            project(apply(
                    gesvd(XTX, 'S'), sigma_inv,
                    POW(sigma,-1)), sigma_inv)
                AS SINV, V.i, SINV.i),
    vsinv, v*sigma_inv), vsinv),
            transpose(gesvd(XTX, 'U')), Z), XTX_INV)
gemm(XTX_INV, gemm(X, y, Z, transa:true), Z)
```

Some Commentary

Challenges Remain...

- Physical data independence is often poor - need to decide *a priori* on dense vs sparse, distributed vs. local, which data type to use etc...
- Tuning remains a "significant challenge":
  1. Often requires substantial systems knowledge (GC tuning, caching and buffer pools)
  2. Poor tuning can kill performance
  3. Often labor intensive - especially for RDBMS type systems
  4. Too many tunable parameters - leads to "tuning fatigue"
  5. Tuning parameters may be workload specific
- More nodes does not always lead to better performance!

Key Takeaways

- Transparently switching between distributed and local execution improves performance and improves usability
- Automatically detecting LA optimizations (diagonal matrix multiply, multiplication chain order) improves performance and improves usability
- Intermediate results should not be needlessly materialized and computations should be pipelined whenever possible
- Strong physical data independence and LA based abstractions are key to an enjoyable programming experience