Scientific Machine Learning with Alchemist (An Apache Spark <=> MPI Interface) and Beyond

Michael W. Mahoney

(RISELab, ICSI, and Department of Statistics, UC Berkeley)

Alchemist

April 2018

Overview

Thoughts on Machine Learning, Scientific Machine Learning, etc.

Spark and Spark performance

Alchemist: an Apache Spark <=> MPI Interface

Communication-avoiding Machine Learning

Current and Future Directions
- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

Overview

Thoughts on Machine Learning, Scientific Machine Learning, etc.

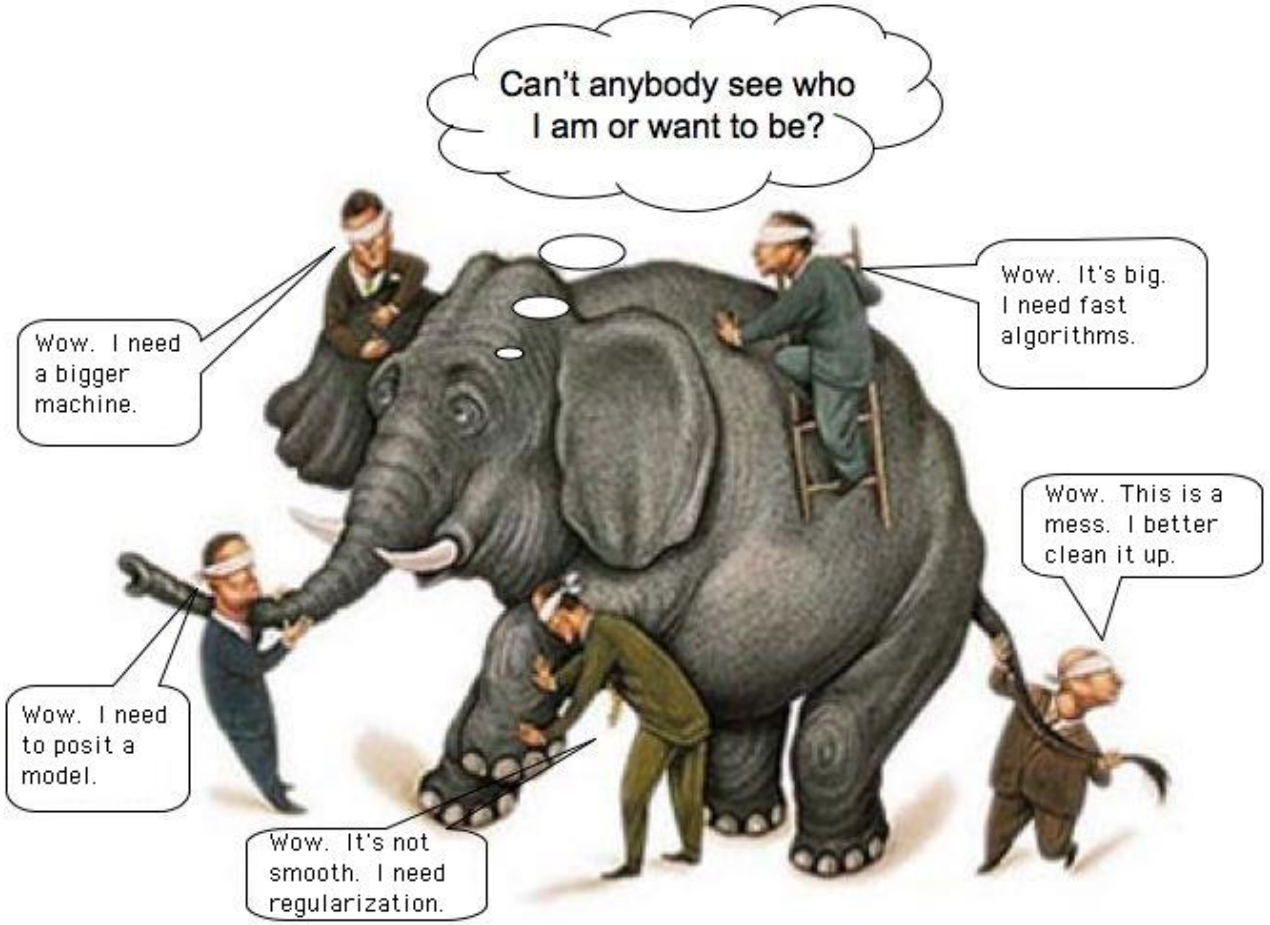
Spark and Spark performance

Alchemist: an Apache Spark <=> MPI Interface

Communication-avoiding Machine Learning

Current and Future Directions
- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

How do we view BIG data?

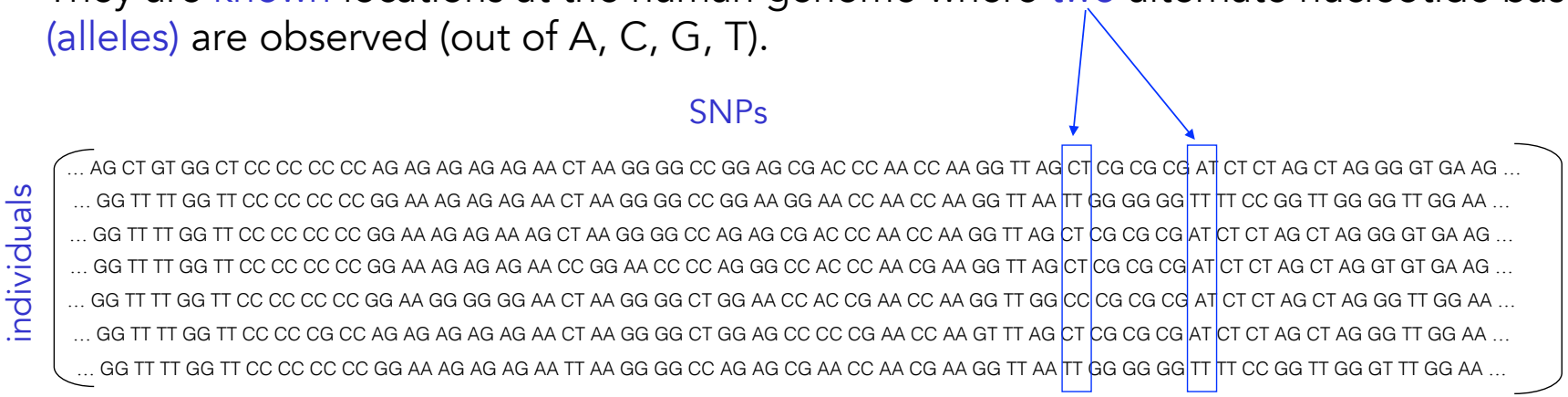


Scientific data and choosing good columns as features

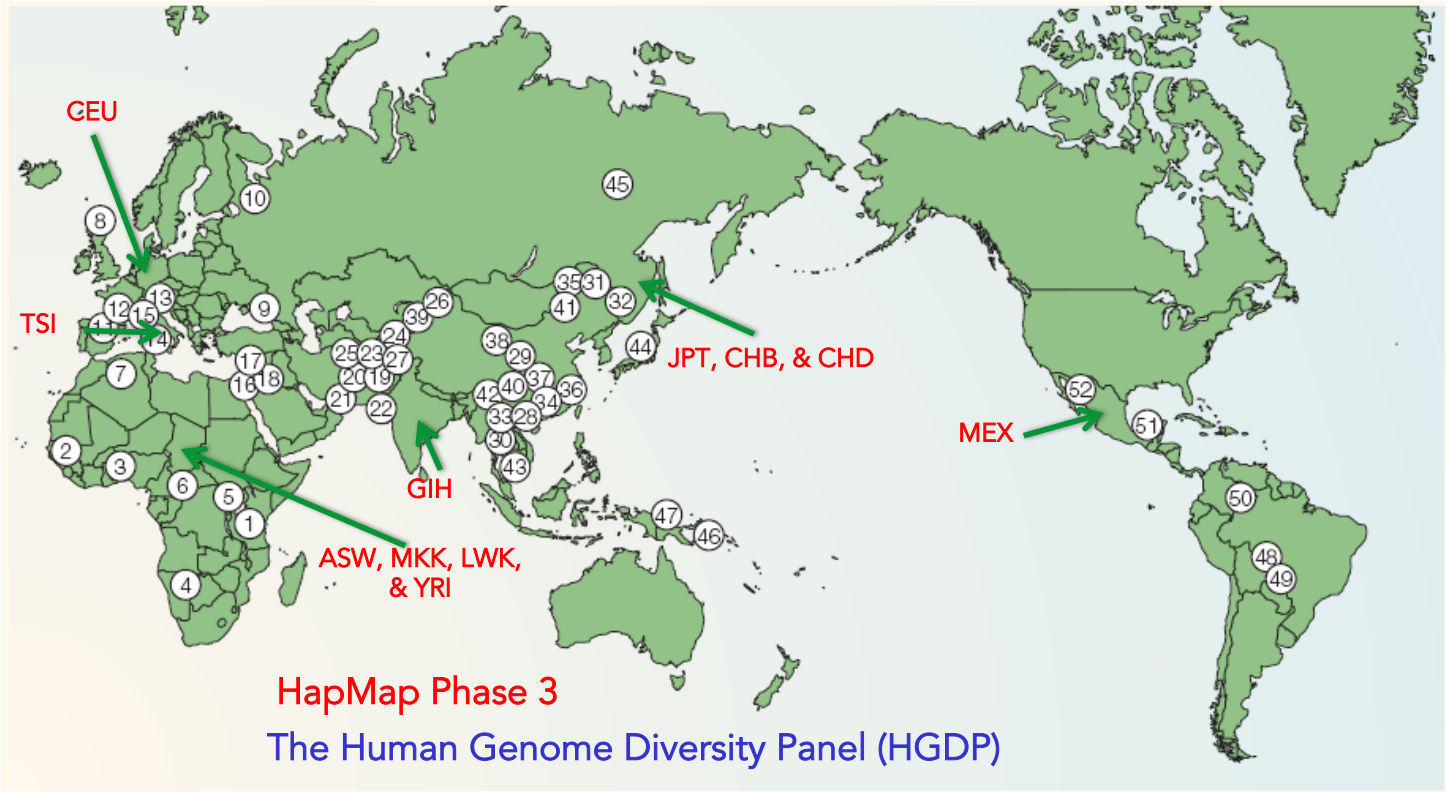
E.g., application in: Human Genetics

Single Nucleotide Polymorphisms: the most common type of genetic variation in the genome across different individuals.

They are known locations at the human genome where two alternate nucleotide bases (alleles) are observed (out of A, C, G, T).



Matrices including thousands of individuals and hundreds of thousands (large for some people, small for other people) if SNPs are available.



Africans
1 Bantu
2 Mandenka
3 Yoruba
4 San
5 Mbuti pygmy
6 Biaka
7 Mozabite

Europeans
8 Orcadian
9 Adygei
10 Russian
11 Basque
12 French
13 North Italian
14 Sardinian
15 Tuscan

Western Asians
16 Bedouin
17 Druze
18 Palestinian

Central and Southern Asians
19 Balochi
20 Brahui
21 Makrani
22 Sindhi
23 Pathan
24 Burusho
25 Hazara
26 Uygur
27 Kalash

Eastern Asians
28 Han (S. China)
29 Han (N. China)
30 Dai
31 Daur
32 Hezhen
33 Lahu
34 Miao
35 Oroqen
36 She
37 Tujia
38 Tu
39 Xibo
40 Yi
41 Mongola
42 Naxi
43 Cambodian
44 Japanese
45 Yakut

Oceanians
46 Melanesian
47 Papuan

Native Americans
48 Karitiana
49 Surui
50 Colombian
51 Maya
52 Pima

Cavalli-Sforza (2005) *Nat Genet Rev*

Rosenberg et al. (2002) *Science*

Li et al. (2008) *Science*

The International HapMap Consortium (2003, 2005, 2007) *Nature*

HGDP data

- 1,033 samples
- 7 geographic regions
- 52 populations

HapMap Phase 3 data

- 1,207 samples
- 11 populations

Apply SVD/PCA on the (joint) HGDP and HapMap Phase 3 data.

Matrix dimensions:

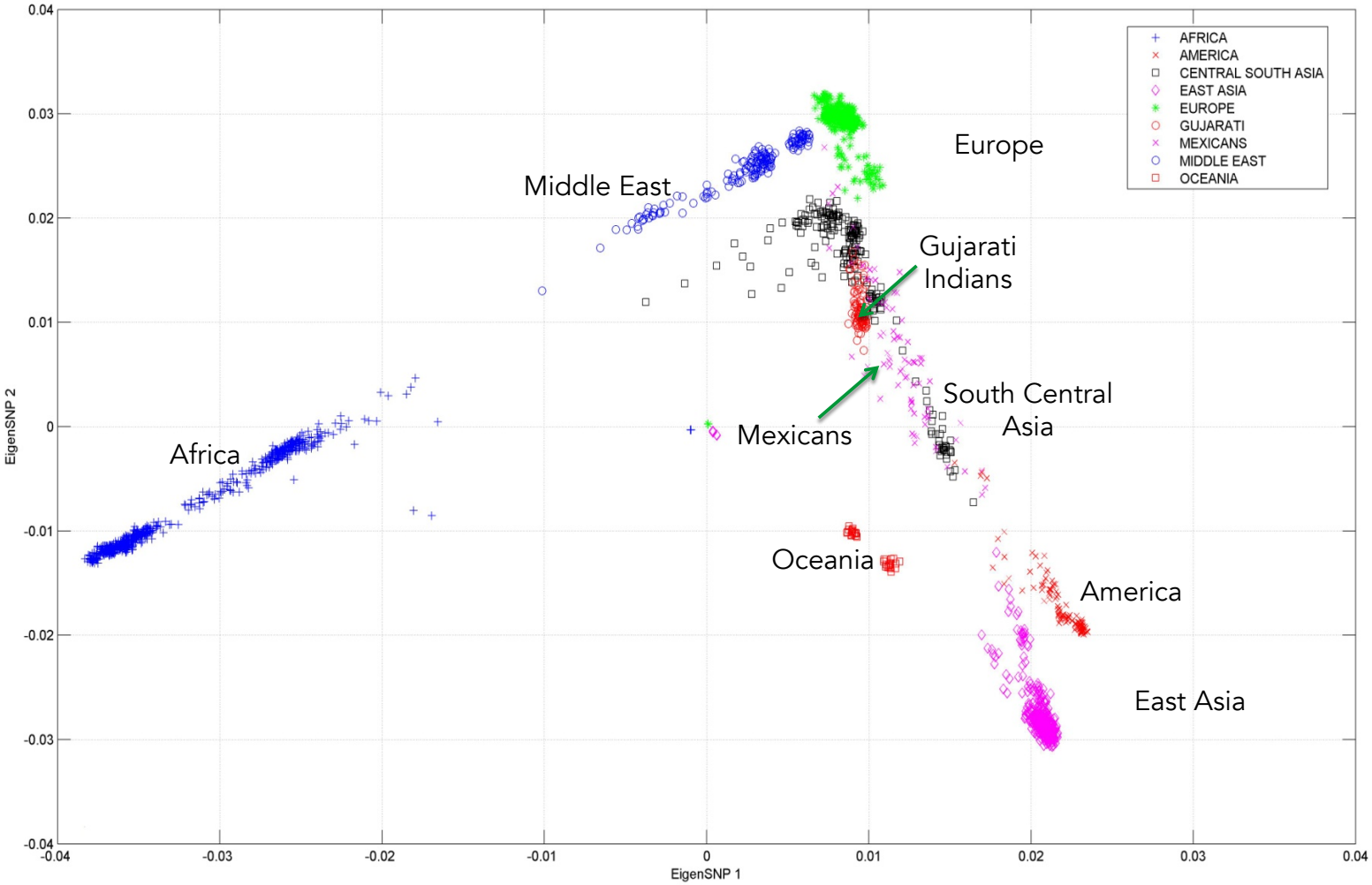
2,240 subjects (rows)

447,143 SNPs (columns)

Dense matrix:

over one billion entries

Paschou, et al (2010) J Med Genet



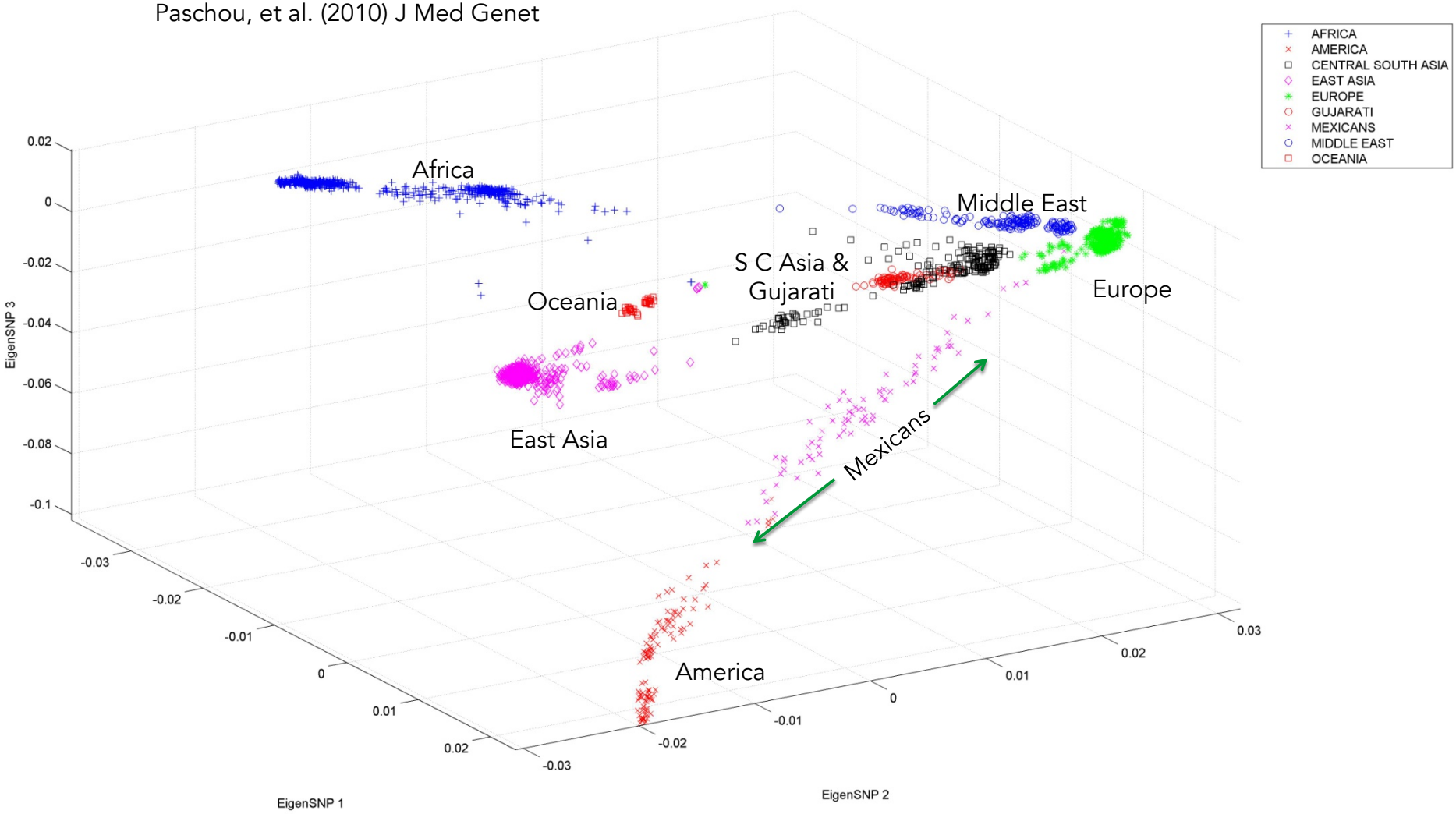
• Top two Principal Components (PCs or eigenSNPs)

(Lin and Altman (2005) *Am J Hum Genet*)

• The figure renders visual support to the "out-of-Africa" hypothesis.

• Mexican population seems out of place: we move to the top three PCs.

Paschou, et al. (2010) J Med Genet



- Not altogether satisfactory: the principal components are linear combinations of all SNPs, and – of course – can not be assayed!
- Can we find actual SNPs that capture the information in the singular vectors?
- Relatedly, can we compute them and/or the truncated SVD "efficiently."

Two related issues with eigen-analysis

Computing large SVDs: computational time

- In commodity hardware (e.g., a 4GB RAM, dual-core laptop), using MatLab 7.0 (R14), the computation of the SVD of the dense 2,240-by-447,143 matrix A takes ca 20 minutes.
- Computing this SVD is not a one-liner, since we can not load the whole matrix in RAM (runs out-of-memory in MatLab).
- Instead, compute the SVD of AA^T.
- In a similar experiment, compute **1,200 SVDs** on matrices of dimensions (approx.) 1,200-by-450,000 (roughly, a full leave-one-out cross-validation experiment) (DLP2010)

Selecting *actual columns* that "capture the structure" of the top PCs

- Combinatorial optimization problem; hard even for small matrices.
- Often called the Column Subset Selection Problem (CSSP).
- Not clear that such "good" columns even exist.
- Avoid "reification" problem of "interpreting" singular vectors!
- (Solvable in "random projection time" with CX/CUR decompositions! (PNAS, MD09))

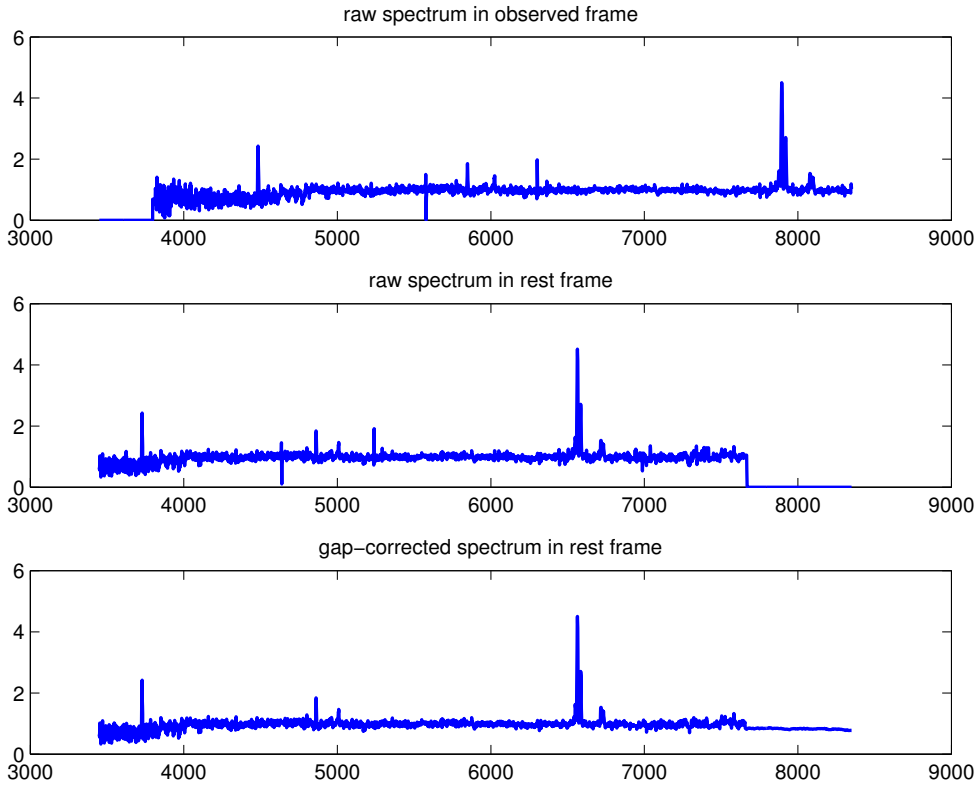
Use case[2]: Galactic spectra from SDSS

$x_i \in \mathbb{R}^{3841}$, $N \approx 500k$

photon fluxes in ≈ 10 Å wavelength bins

preprocessing corrects for redshift, gappy regions

normalized by median flux at certain wavelengths



Global embedding: effect of k

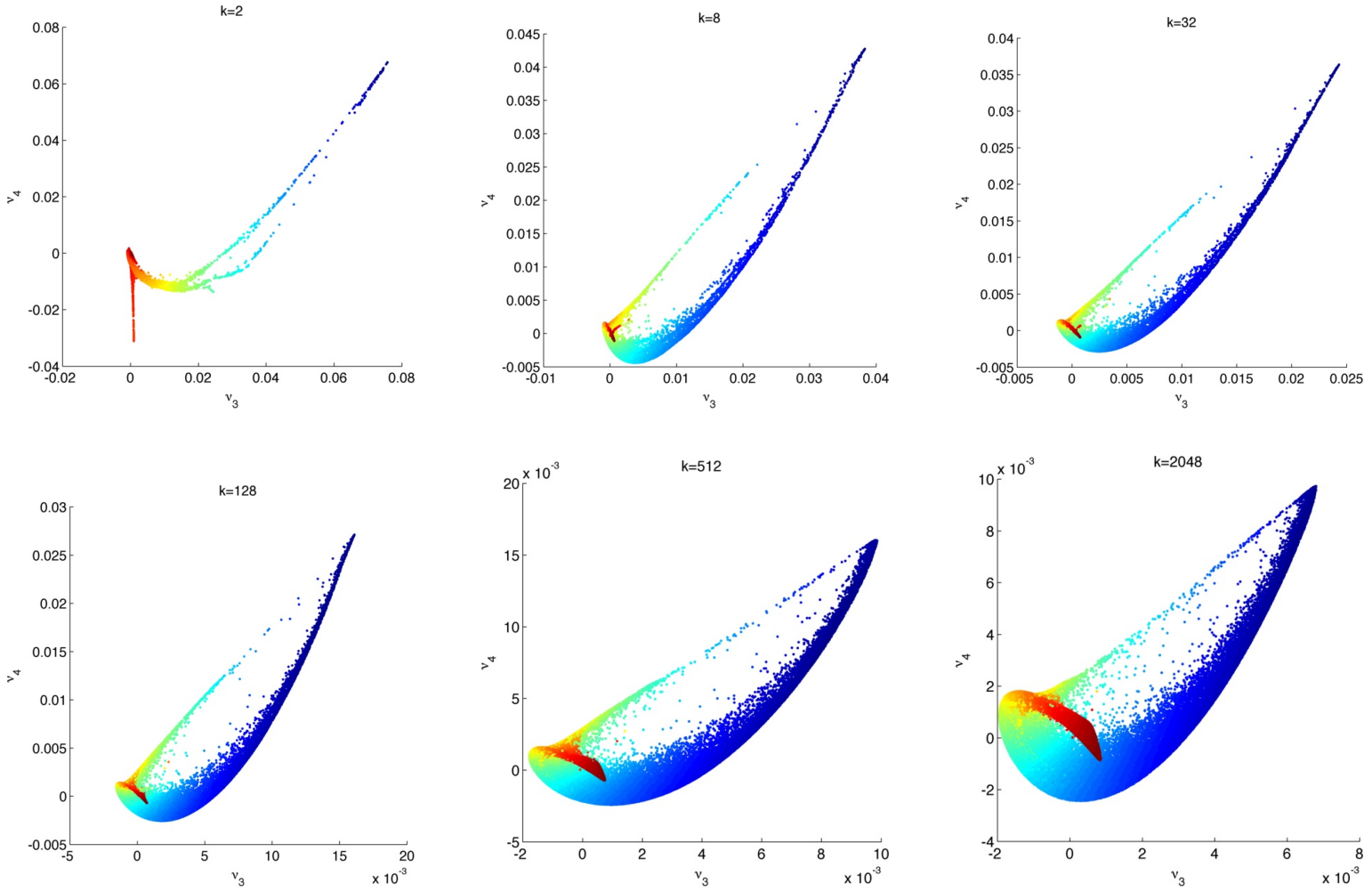
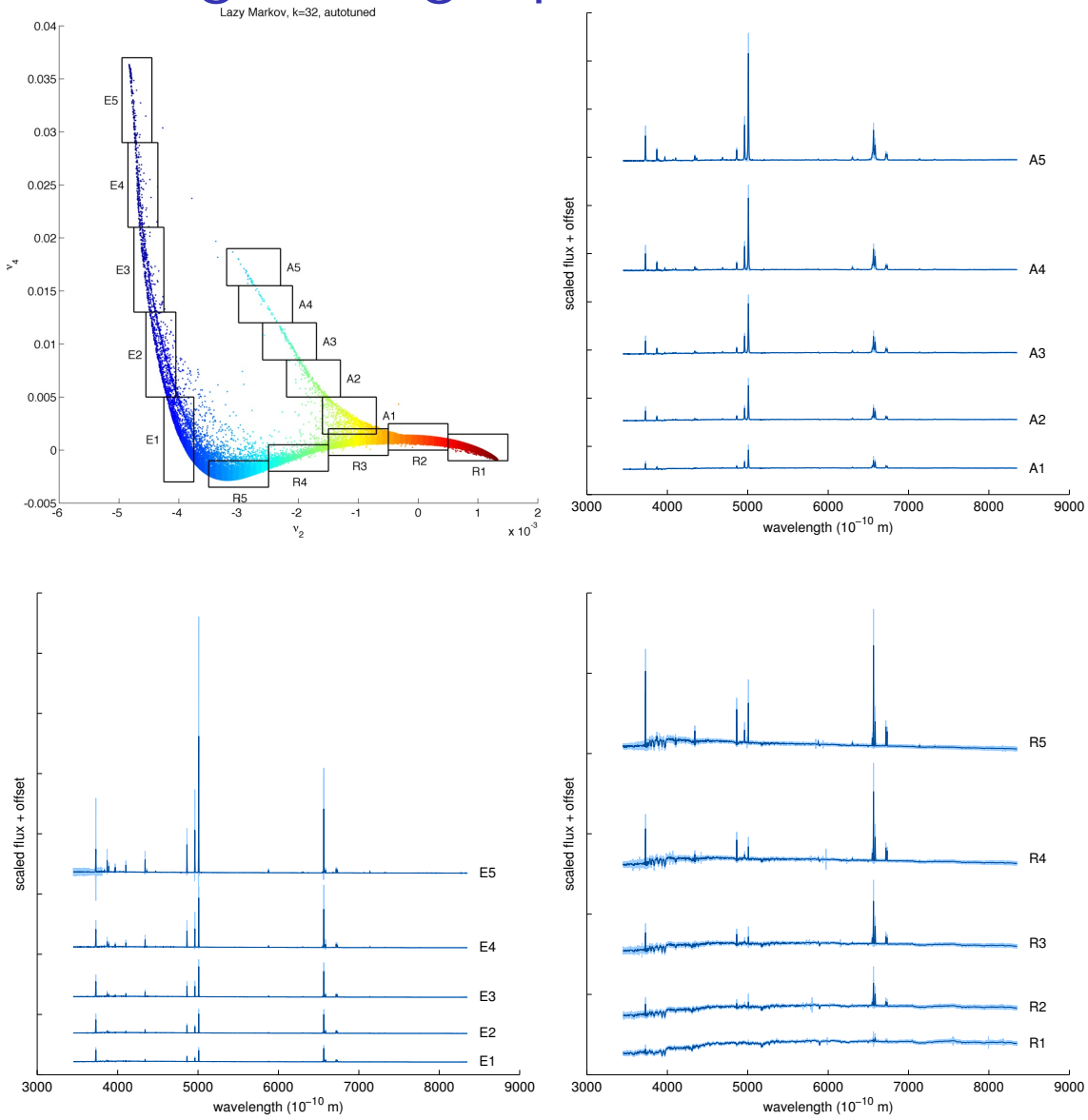


Figure: Eigenvectors 3 and 4 of Lazy Markov operator, $k = 2 : 2048$

Global embedding: average spectra



Local embedding: scale parameter and effect of seed

For an appropriate choice of c and $\gamma = \gamma(\kappa) < \lambda_2$, one can show

$$\begin{aligned} w_2 &= c(L - \gamma D)^+ Ds \\ &= c(L_G - \gamma L_{k_n})^+ Ds \end{aligned}$$

(In practice, binary search to find "correct" γ.)

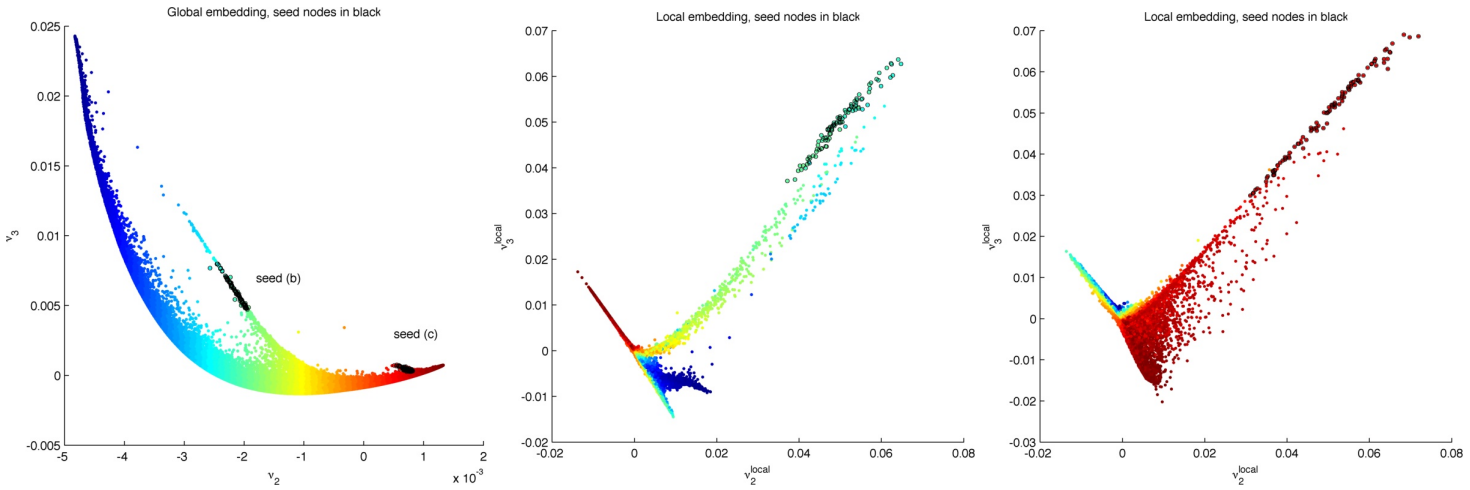


Figure: (left) Global embedding with seeds in black. (middle, right) Local embeddings using specified seeds.

Overview

Thoughts on Machine Learning, Scientific Machine Learning, etc.

Spark and Spark performance

Alchemist: an Apache Spark <=> MPI Interface

Communication-avoiding Machine Learning

Current and Future Directions
- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

Where do you run your linear algebra?

Single machine

- Think about RAM, call LAPACK, etc.
- Someone else thought about numerical issues, memory hierarchies, etc.
- This is the 99%

Supercomputer

- High end, compute-intensive.
- Big emphasis on HPC (High Performance Computing)
- C+MPI, etc.

Distributed data center

- High end, data-intensive
- BIG emphasis on HPC (High Productivity Computing)
- Databases, MapReduce/Hadoop, Spark, etc.

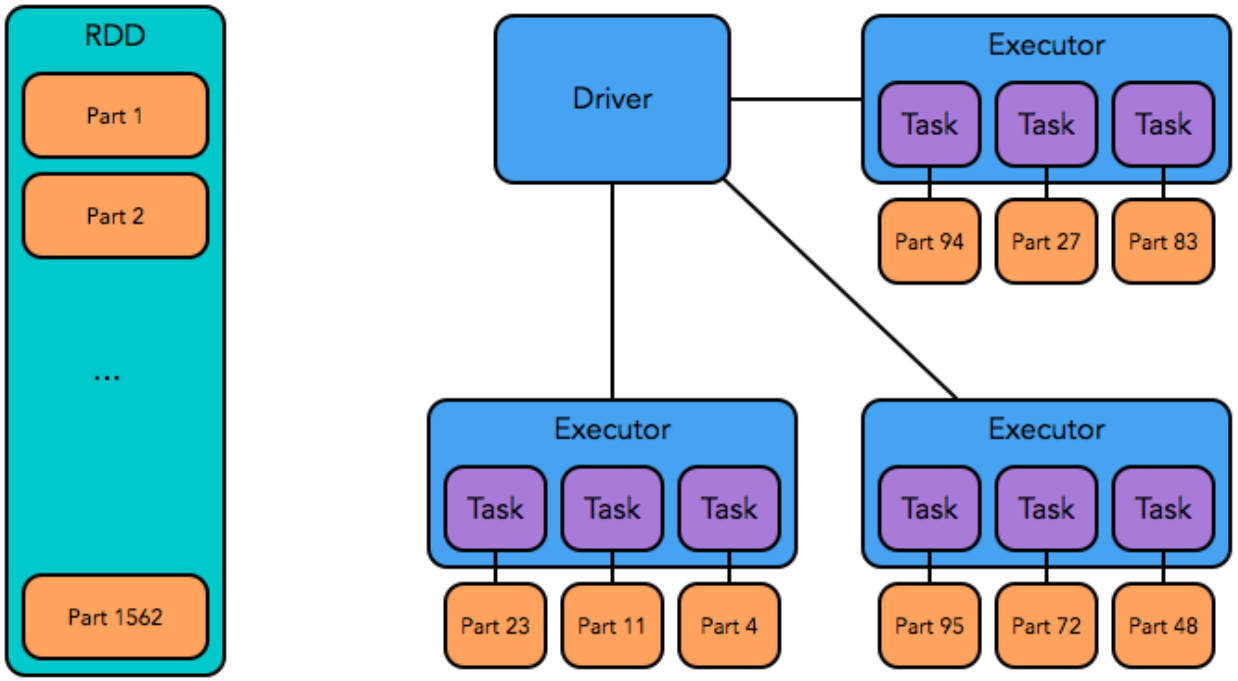
Apache Spark

- Cluster computing system
- Interoperable with Apache Hadoop, much faster
- Improved efficiency:
  - In-memory computing primitives
  - Computation graphs
- Rich and easy-to-use API
- Allows for iterative algorithms (important for ML and linear algebra)
- Fault tolerant

MPI

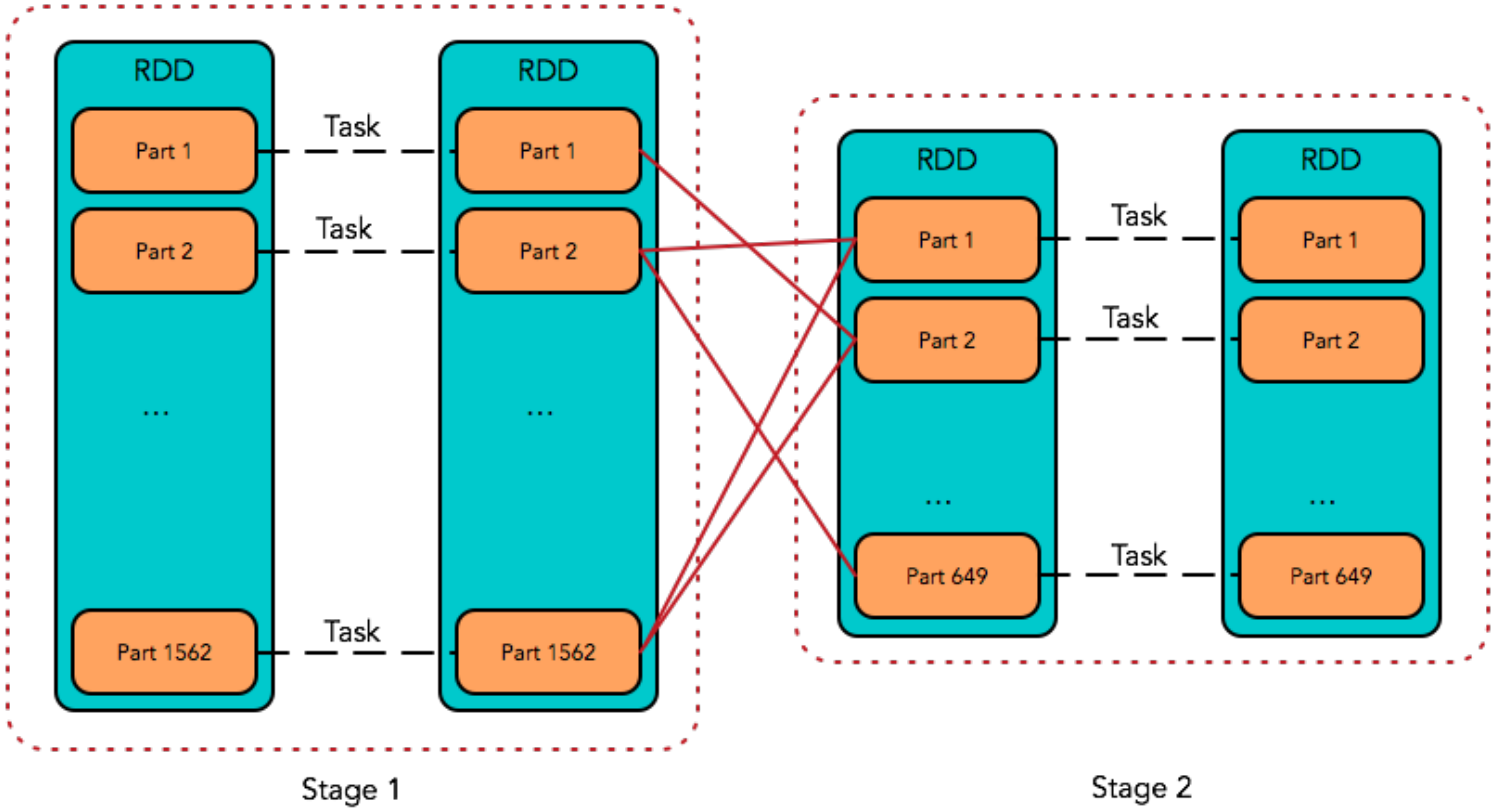
- MPI = Message Passing Interface
- A *specification* for the developers and users of message passing libraries
- *Message-Passing Parallel Programming Model*: cooperative operations between processes, data moved from address space of one process to that of another
- Popular *implementations*: MPICH, Open MPI

Spark Architecture



- Data parallel programming model
- Resilient distributed datasets (RDDs) (think: distributed array type)
- RDDs can optionally be cached in memory b/w iterations
- Driver forms DAG, schedules tasks on executors

Spark Communication



- Computation operate on one RDD to produce another RDD
- Each overall job (DAG) broken into stages
- Stages broken into parallel, independent tasks
- Communication happens only between stages

Why do linear algebra in Spark?

Pros:

- *Widely used*
- *Easier to use for non-experts*
- An entire ecosystem that can be used before and after the NLA computations
- Spark can take advantage of available single-machine linear algebra codes (e.g. through netlib-java)
- Automatic fault-tolerance
- Transparent support for out of memory calculations

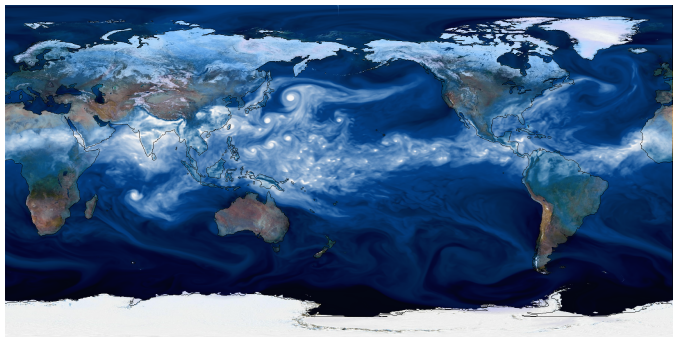
Cons:

- Classical MPI-based linear algebra algorithms are faster and more efficient
- No way, currently, to leverage legacy parallel linear algebra codes
- JVM matrix size restrictions, and RDD rigidity

Our Goals

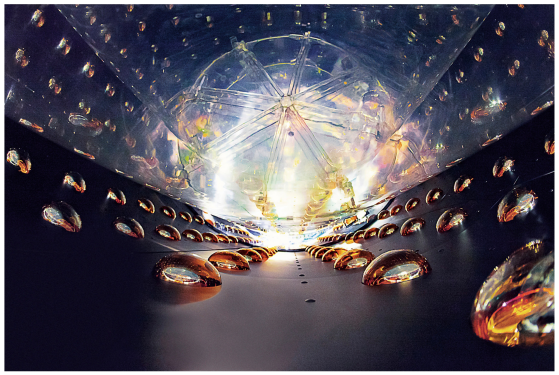
- ***Provide implementations*** of low-rank factorizations (PCA, NMF, and randomized CX) in Spark
- Apply low-rank matrix factorization methods ***to TB-scale scientific datasets*** in Spark
- Understand Spark performance on ***commodity clusters vs HPC platforms***
- ***Quantify the scalability gaps*** between highly-tuned C/MPI and current Spark-based implementations
- ***Provide a general-purpose interface*** for matrix-based algorithms between Spark and traditional MPI codes

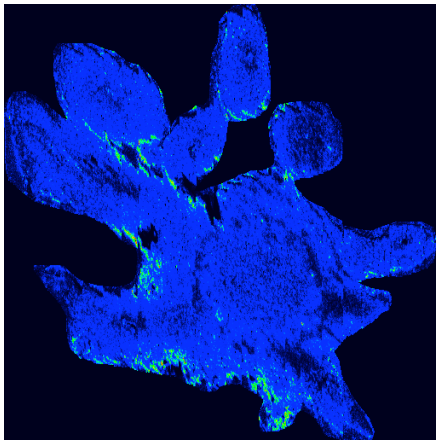
Three Science Drivers



Climate Science:
extract trends in variations of oceanic and atmospheric variables (**PCA**)

Nuclear Physics:
learn useful patterns for classification of subatomic particles (**NMF**)





Mass Spectrometry:
location of chemically important ions (**CX**)

Datasets

Science Area	Format/Files	Dimensions	Size
MSI	Parquet/2880	$8,258,911 \times 131,048$	1.1TB
Daya Bay	HDF5/1	$1,099,413,914 \times 192$	1.6TB
Ocean	HDF5/1	$6,349,676 \times 46,715$	2.2TB
Atmosphere	HDF5/1	$26,542,080 \times 81,600$	16TB

MSI — a sparse matrix from measurements of drift times and mass charge ratios at each pixel of a sample of *Peltatum*; used for CX decomposition

Daya Bay — neutrino sensor array measurements; used for NMF

Ocean and Atmosphere — climate variables (ocean temperature, atmospheric humidity) measured on a 3D grid at 3 or 6 hour intervals over about 30 years; used for PCA

Our first (of by now several) results

Matrix Factorizations at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies

Alex Gittens [*] Aditya Devarakonda[†] Evan Racah[‡] Michael Ringenburg[§]
Lisa Gerhardt[‡] Jey Kottaalam[†] Jialin Liu[‡] Kristyn Maschhoff[§] Shane Canon[‡]
Jatin Chhugani[¶] Pramod Sharma[§] Jiyan Yang[‖] James Demmel[**] Jim Harrell[§]
Venkat Krishnamurthy[§] Michael W. Mahoney[*] Prabhat [‡]

July 1, 2016

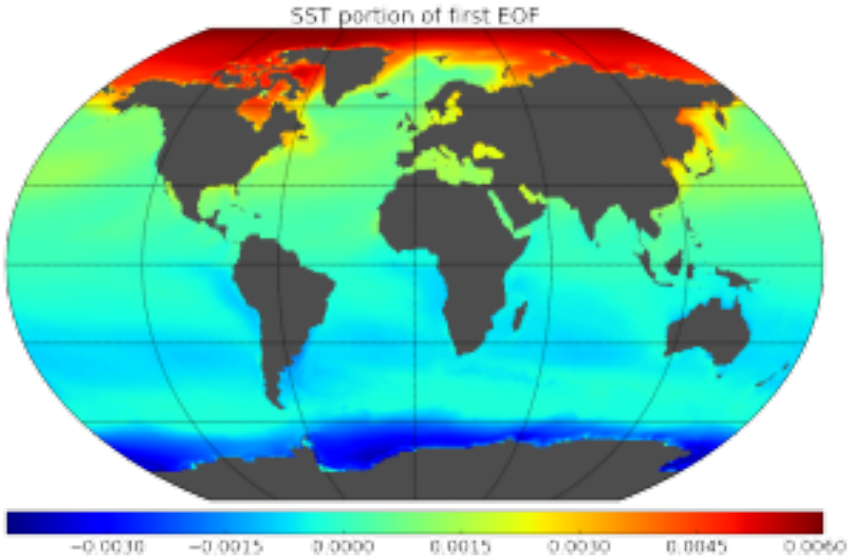
arXiv:1607.01335v1 [cs.DC] 5 Jul 2016

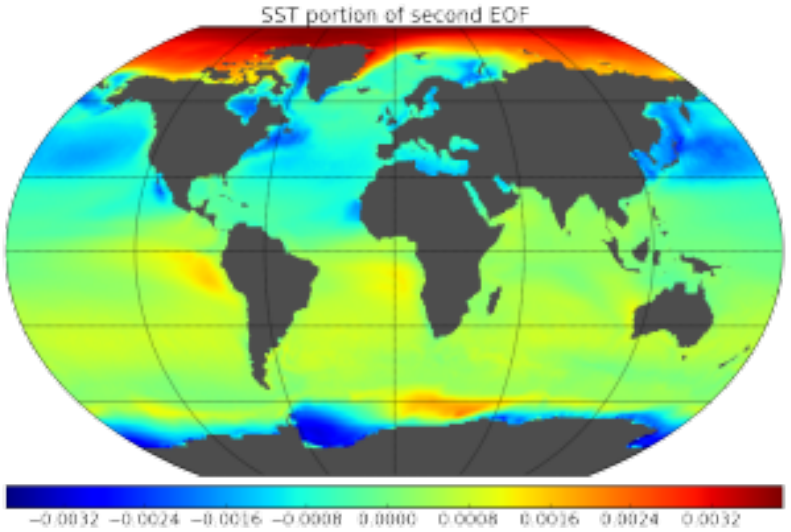
[*]ICSI and Department of Statistics, UC Berkeley
[†]EECS, UC Berkeley
[‡]NERSC, Lawrence Berkeley National Laboratory
[§]Cray, Inc.
[¶]Hiperform Consulting LLC
[‖]ICME, Stanford University
[**]EECS and Math, UC Berkeley

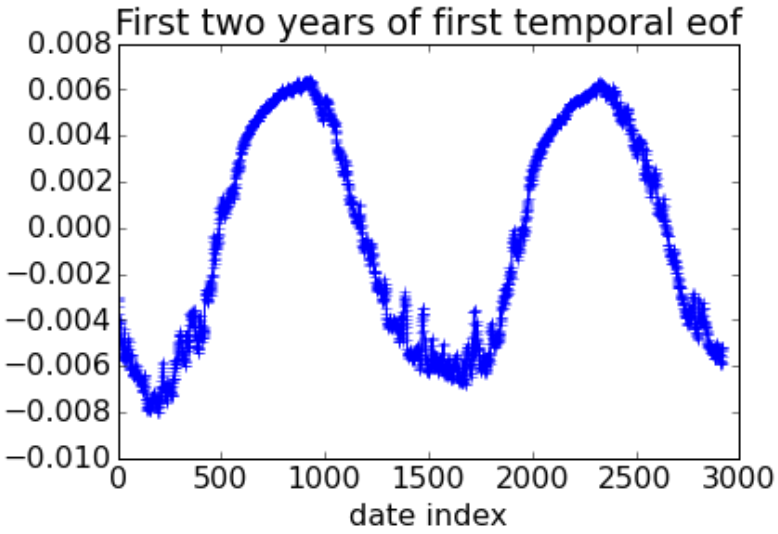
BIG thanks in particular to:
- Mike Ringenberg
- Kristi Maschoff
- Pramod Sharma
- Jim Harrell
- Venkat Krishnamurthy

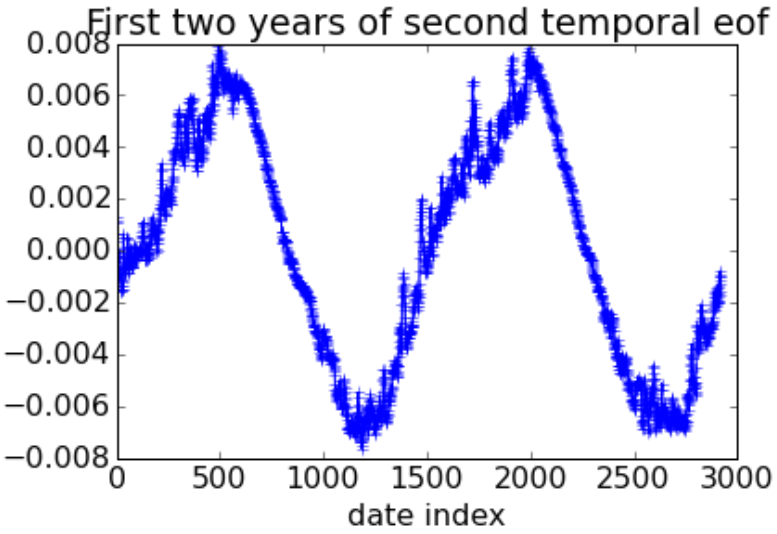
(and Cray for funding!)

CFSR Ocean Temperature Dataset (II)









Running times for NMF and PCA

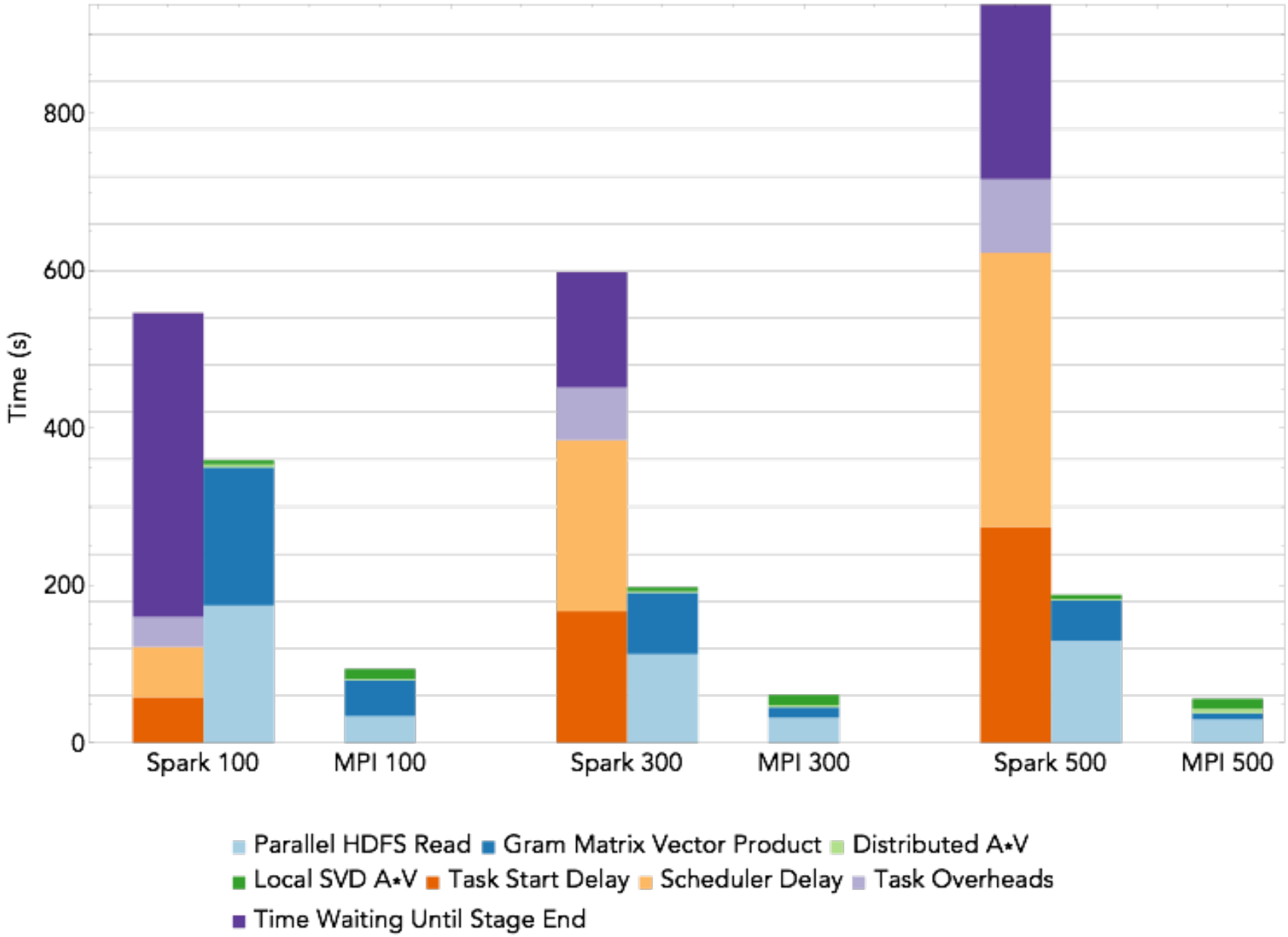
Cori's specs:

- 1630 compute nodes,
- 128 GB/node,
- 32 2.3GHz Haswell cores/node

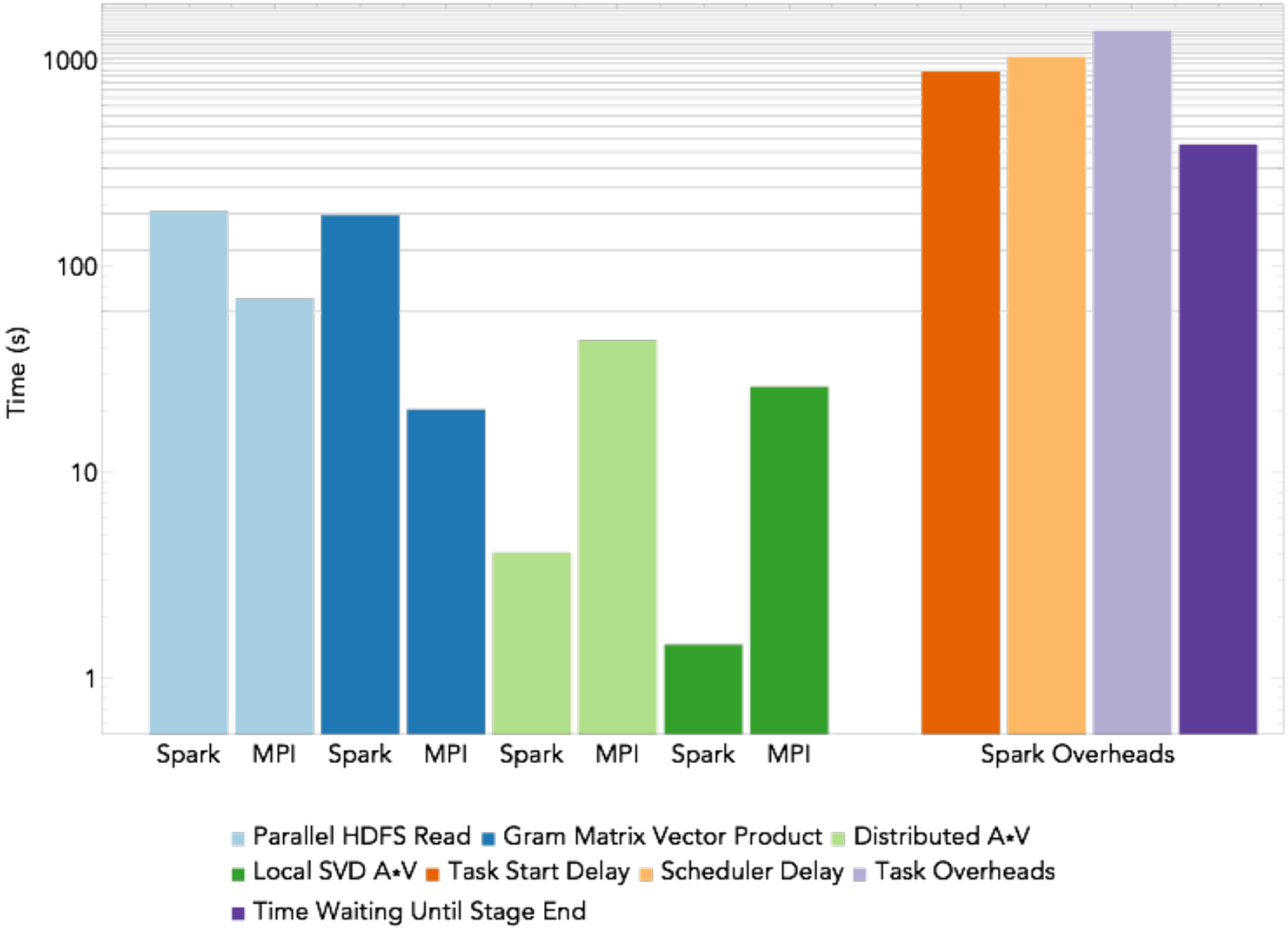
	Nodes / cores	MPI Time	Spark Time	Gap
NMF	50 / 1,600	1 min 6 s	4 min 38 s	4.2x
	100 / 3,200	45 s	3 min 27 s	4.6x
	300 / 9,600	30 s	70 s	2.3x
PCA (2.2TB)	100 / 3,200	1 min 34 s	15 min 34 s	9.9x
	300 / 9,600	1 min	13 min 47 s	13.8x
	500 / 16,000	56 s	19 min 20 s	20.7x
PCA (16TB)	MPI: 1,600 / 51,200 Spark: 1,522 / 48,704	2 min 40 s	69 min 35 s	26x

- Anti-scaling!
- And it worsens both with concurrency and data size.

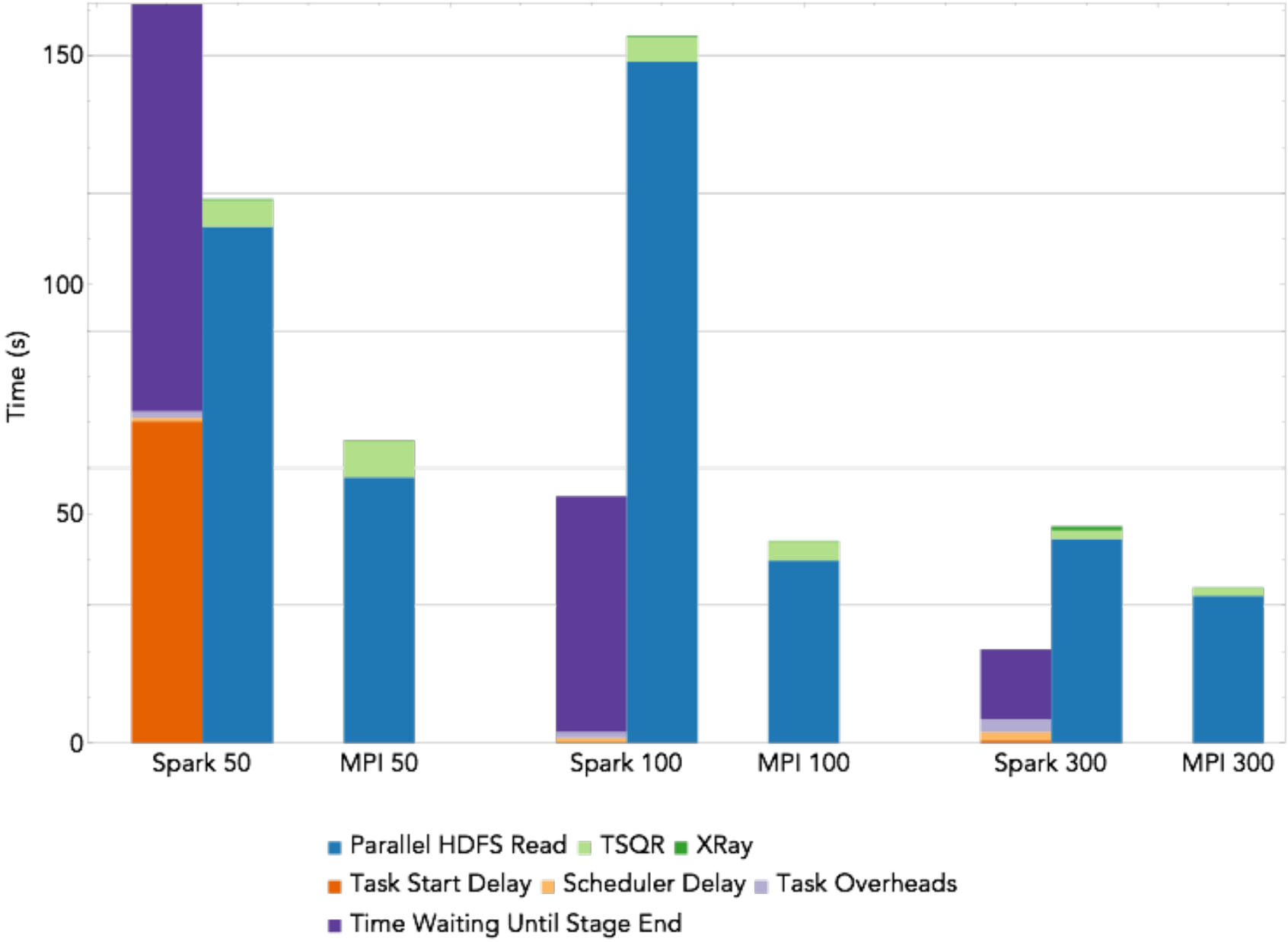
PCA Run Times: rank 20 PCA of 2.2TB Climate



Rank 20 PCA of 16 TB Climate using 48K+ cores



NMF Run Times: rank 10 NMF of 1.6TB Daya Bay



MPI vs Spark: Lessons Learned

- With favorable data (tall and skinny) and well-adapted algorithms, ***Spark LA is 4x-26x slower than MPI when IO is included***
- ***Spark overheads are orders of magnitude higher than the computations*** in PCA (time till stage end, scheduler delay, task start delay, executor deserialize time), ***and it anti-scales***
- The large gaps mean it is worthwhile to ***investigate efficiently interfacing MPI-based codes with Spark***

Overview

Thoughts on Machine Learning, scientific Machine Learning, etc.

Spark and Spark performance

Alchemist: an Apache Spark <=> MPI Interface

Communication-avoiding Machine Learning

Current and Future Directions
- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

The Next Step: Alchemist

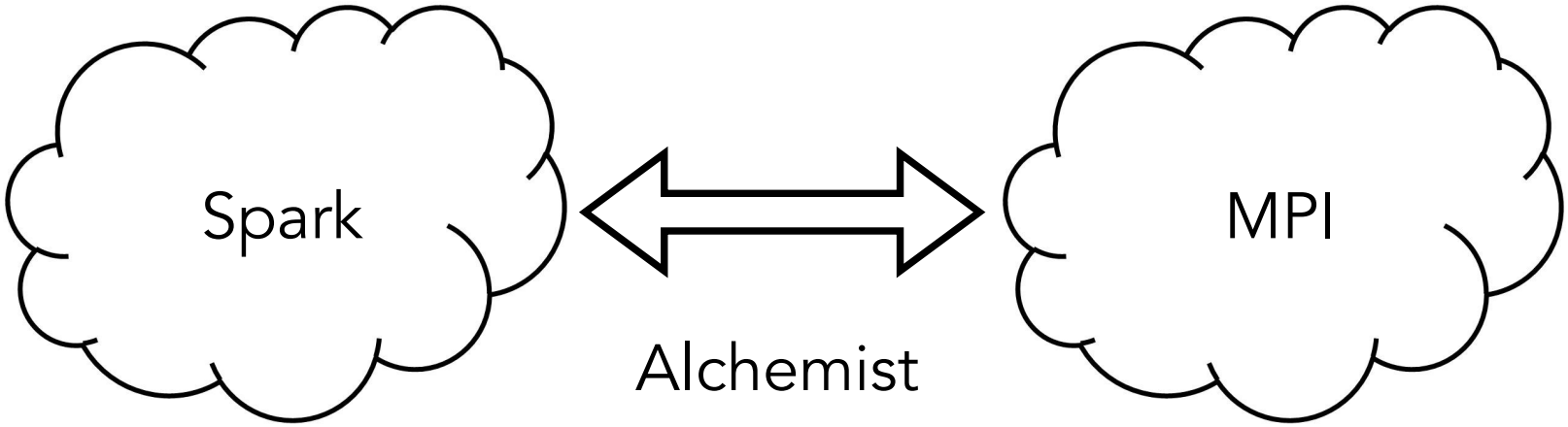
- Since Spark is 4+x slower than MPI, propose sending the matrices to MPI codes, then receiving the results
- For efficiency, want as little overhead as possible (File I/O, RAM, network usage, computational efficiency)

Alternative approaches:

1. Write to HDFS: ***slow file I/O, manual data layout***
2. Other MPI–Spark bridges: ***assume sparse data sets, use ram disk or write to file***
3. Apache Ignite (and Alluxio, etc.): ***requires using C/C++ interfaces, manual data layout, extra copy in memory, TCP/IP***

Alchemist:
Uses *in-memory transfer*, transparently *provides data relayout*, explicitly *handles dense data sets*

Using Alchemist



Spark:

1) Sends the metadata for input and output matrices to Alchemist

2) Sends the matrix to Alchemist using sockets

3) Sends commands to the Alchemist driver

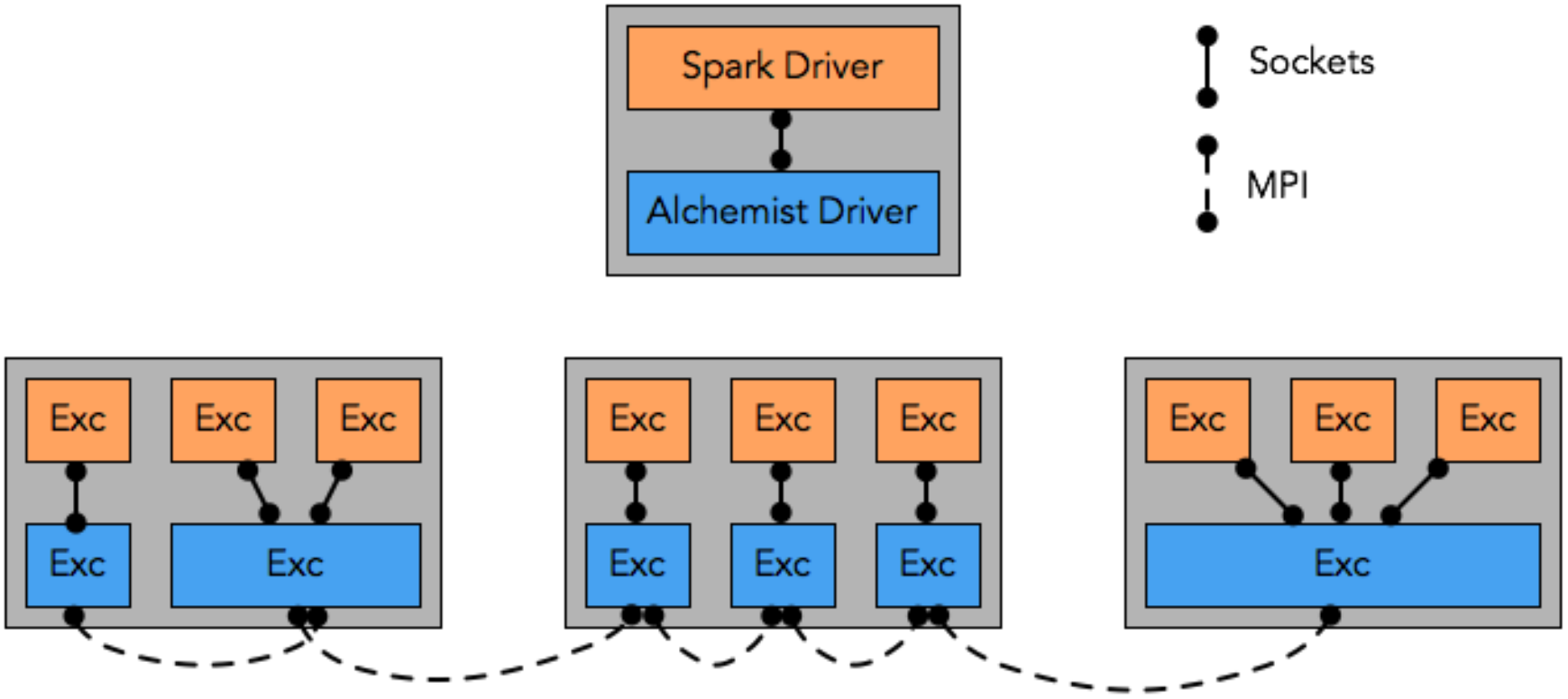
Alchemist:

1) Repartitions the matrix for MPI using Elemental

2) Driver coordinates workers in executing the MPI codes

3) Returns outputs to Spark

Current Alchemist Architecture



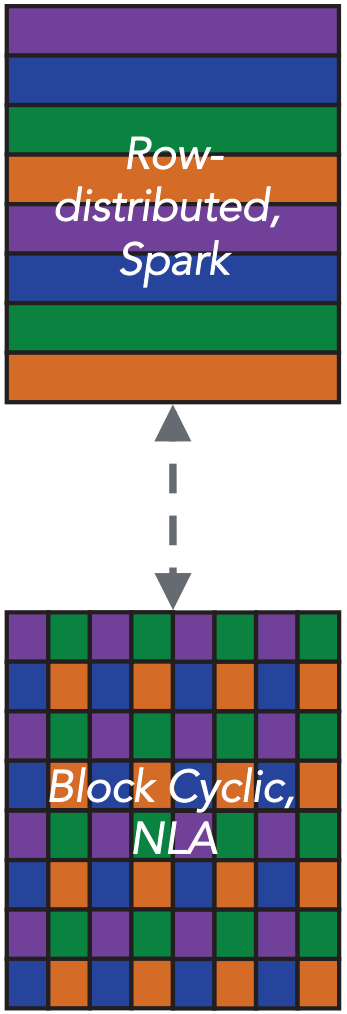
- Exploit locality to reduce communication
  - Allow for hybrid OpenMP/MPI

Main Challenges

Minimizing communication time between Spark workers and Alchemist workers

while also

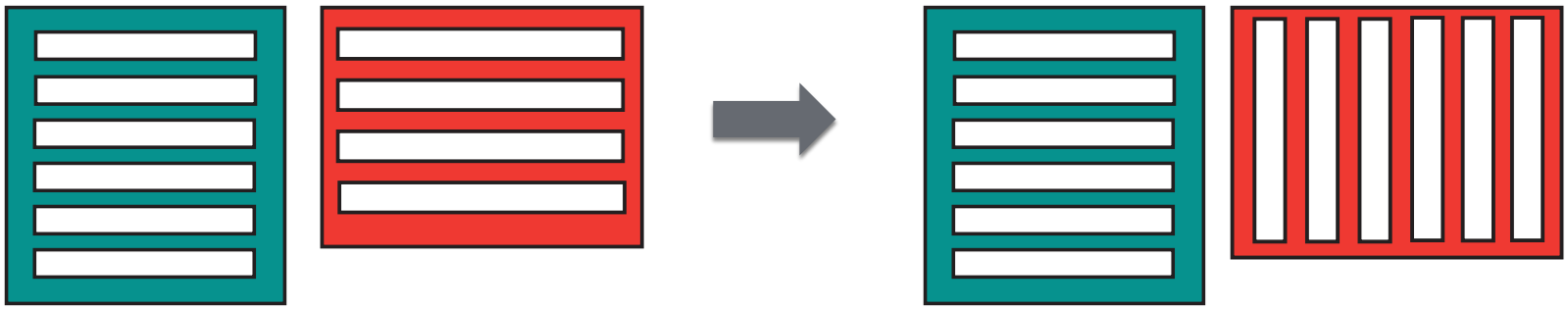
Switching between the matrix distribution schemes imposed by Spark and NLA codes, as needed



Currently Implemented Operations

Operations Implemented	Library/Memory Cost
Matrix **Send**	- / 1X
Matrix **Retrieve**	- / 1X
Matrix **Transpose**	Elemental / 2X
Matrix **Multiply**	Elemental / 2X
KMeans	- / 1X
SVD	Elemental / 2X
Truncated SVD	ARPACK / 2X
LSQR **linear solver**	LibSkylark / 1X
Regularized **CG linear solver**	LibSkylark / 1X
Kernel Solver (regression, classification, regularization)	LibSkylark / 1X
HDF5 Reader	- / 2X

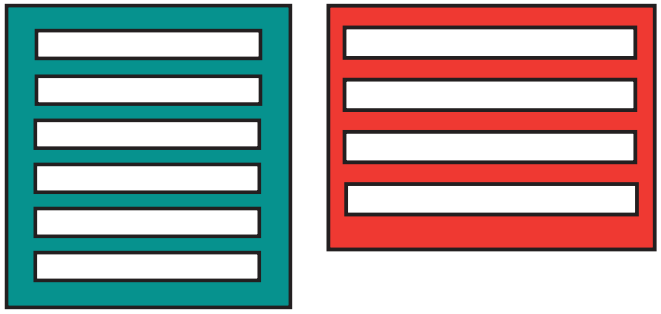
Example: Matrix Multiplication



Requires expensive shuffles in Spark:

- Matrices/RDDs are row partitioned
- One must be converted to be column-partitioned
- This requires an all-to-all shuffle that often fails ***even for matrices that could fit in memory on one executor***

Simplest Example: Matrix Multiplication

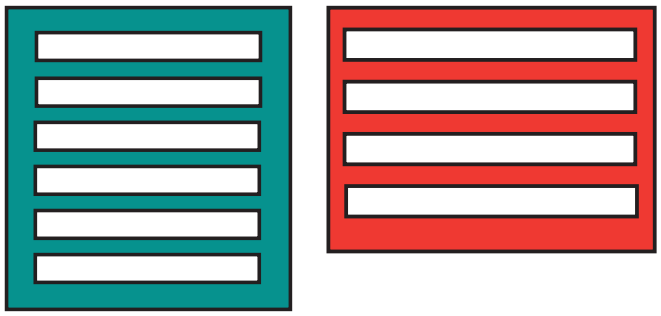


A: 100K-by-10K (8 GB)
B: 10K-by-70K (5.6 GB)
C=AB: 100K-by-70K (56 GB)

Setup:
-128 GB RAM and 32 cores per node
-2 Spark and 2 Alchemist nodes (need 2 nodes due to 2x overhead)

	Send	*Compute*	*Receive*
Alchemist	35.18s	207.21s	56.25s
Spark	-	Fail after 465 s	-

Larger Example: Matrix Multiplication



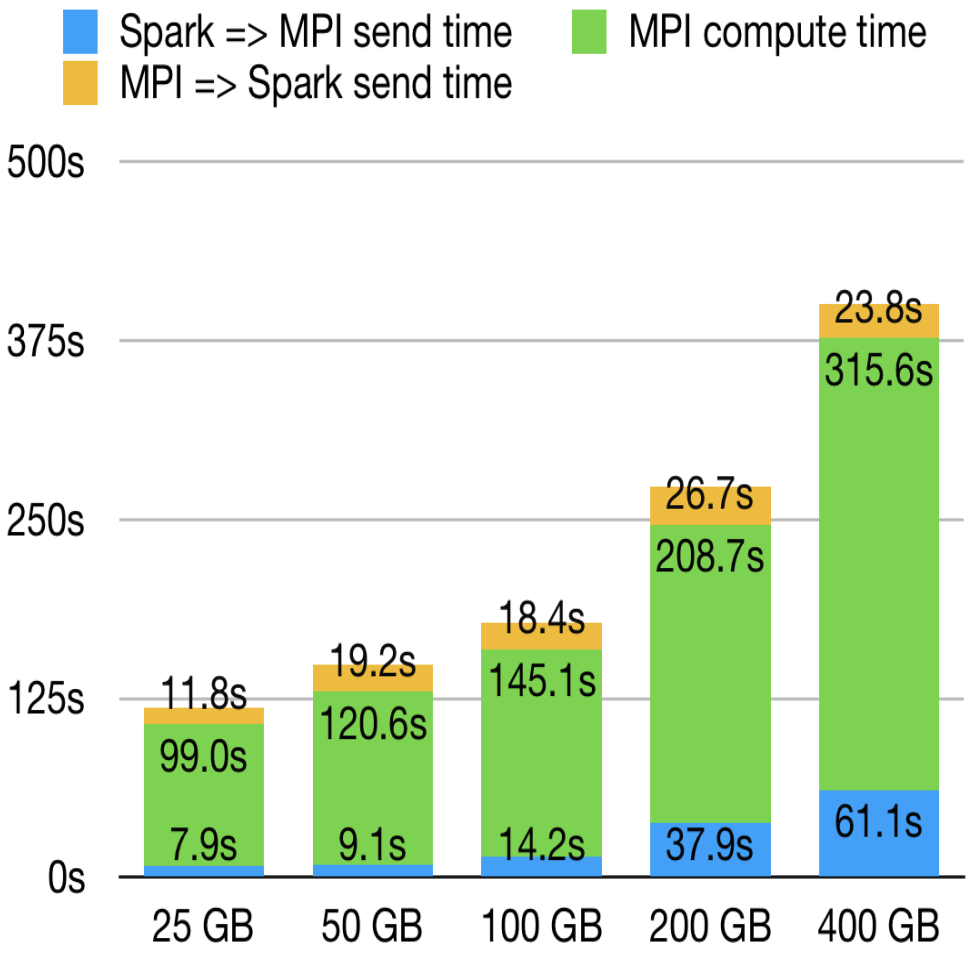
A: 300K-by-10K (24 GB)
B: 10K-by-60K (4.8 GB)
C=AB: 300K-by-60K (144 GB)

Setup:
-128 GB RAM and 32 cores per node
-10 Spark and 10 Alchemist nodes

	Send	*Compute*	*Receive*
Alchemist	83.8s	98s	36.65s
Spark	-	Fail after 30 min	-

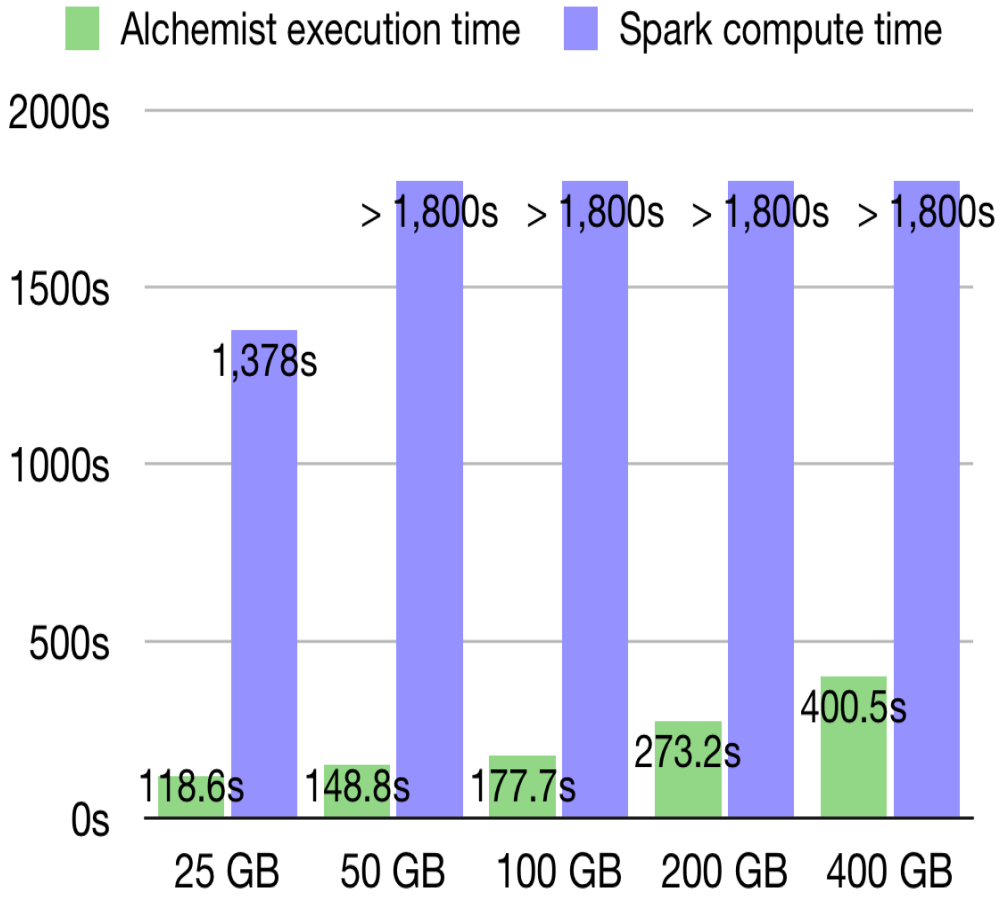
Scaling of Truncated SVD

- Use Alchemist and MLlib to get rank 20 truncated SVD
- Setup:
  - Each node of Cori has 128GB RAM and 32 cores
  - Spark: 22 nodes; Alchemist: 8 nodes
  - A: m-by-10K, where m = 5M, 2.5M, 1.25M, 625K, 312.5K
  - Ran jobs for at most 30 minutes (1800 s)



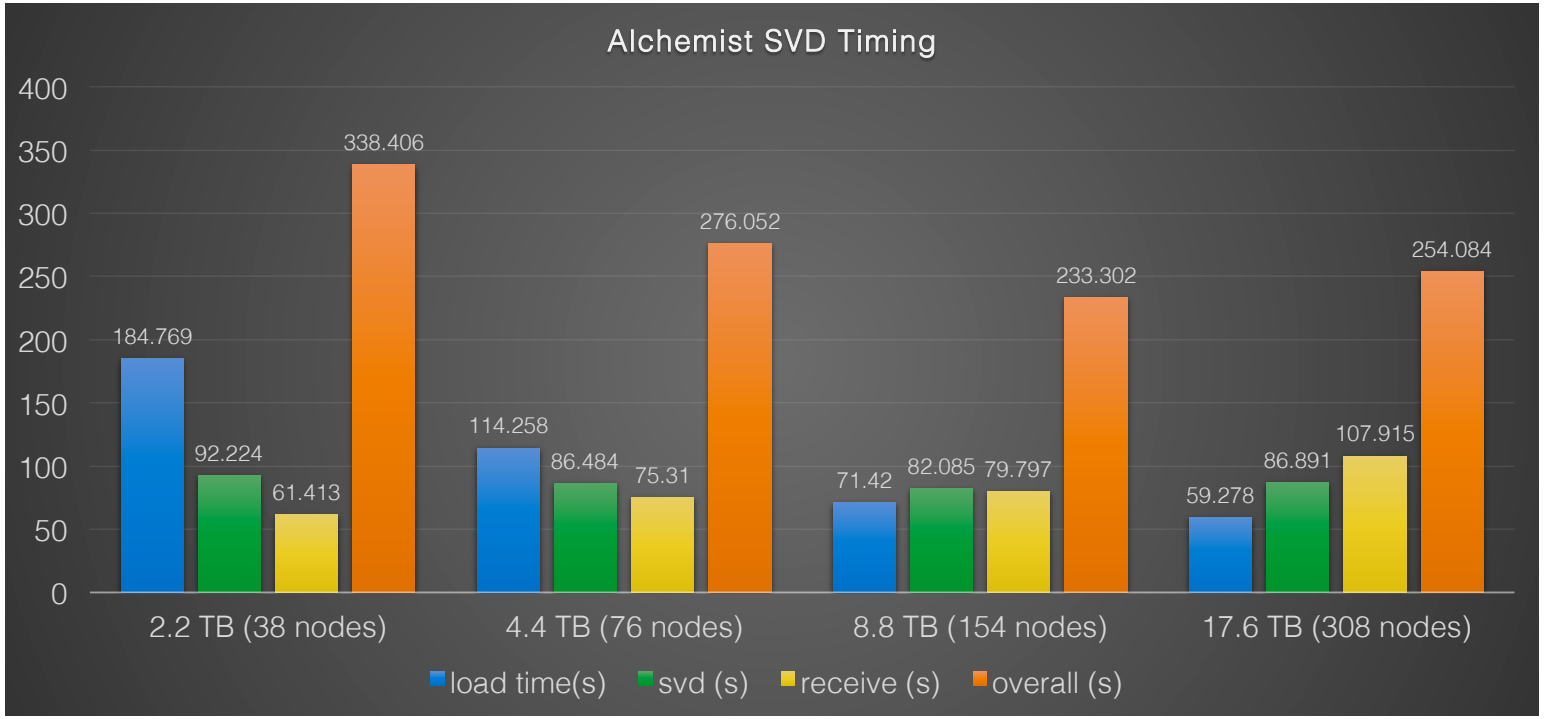
Scaling of Truncated SVD

- Use Alchemist and MLlib to get rank 20 truncated SVD
- Setup:
  - Each node of Cori has 128GB RAM and 32 cores
  - Spark: 22 nodes; Alchemist: 8 nodes
  - A: m-by-10K, where m = 5M, 2.5M, 1.25M, 625K, 312.5K
  - Ran jobs for at most 30 minutes (1800 s)



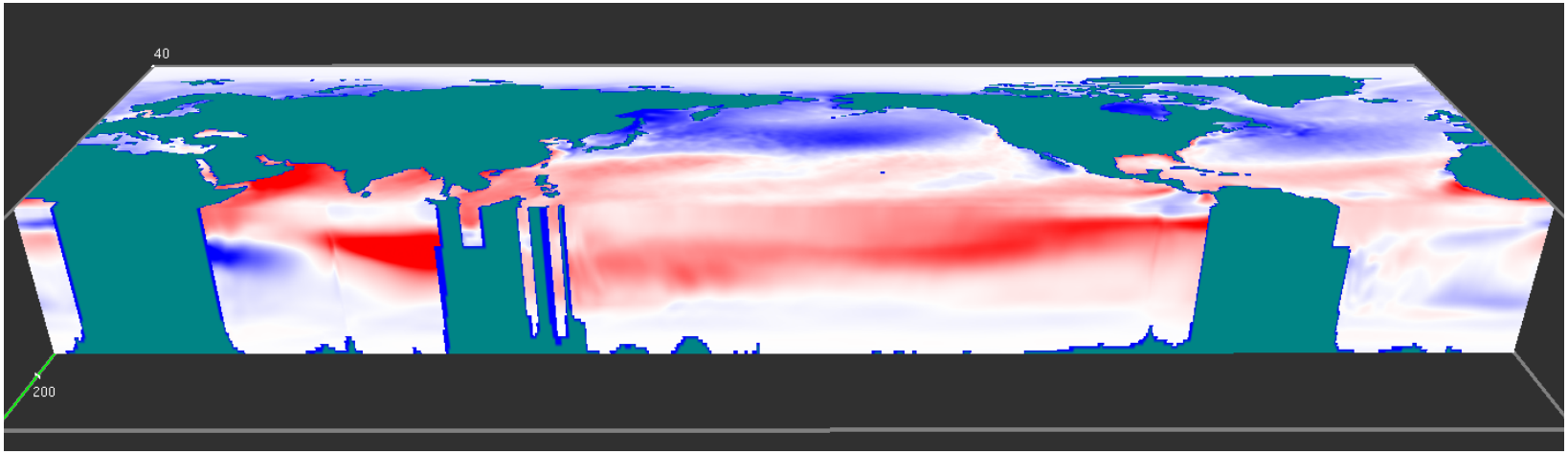
Scaling of Truncated SVD

- Each node of Cori has 128GB RAM and 32 cores
- Replicated the 2.2 TB Climate data set (row-wise)
- Use Alchemist to get rank 20 truncated SVD
- *Prior work shows Spark requires 934s on 100 nodes for just 2.2 TB*



A Climate Science Application

- Problem: extract the top principal components of a 3D ocean temperature data set collected over 30 years at 3 hour increments on a 360-by-720-by-40 lat-long-depth grid
- Yields a **2.2TB matrix, 6M-by-46K and dense**



Visualization of the 5th principal component

Alchemist: Next Steps

- Eliminate 2X overhead of GEMM (ongoing)
- Allow dynamic loading (ongoing)
- More thorough logging, error-handling, and profiling (ongoing)

- Integration with container management frameworks (Kubernetes)
- Exploit locality to minimize communication costs
- Support sparse matrices
- Provide interface for ScaLAPACK codes
- Increased ML functionality via LibSkylark, MCMC clustering codes, Cyclops tensor framework

Overview

Thoughts on Machine Learning, scientific Machine Learning, etc.

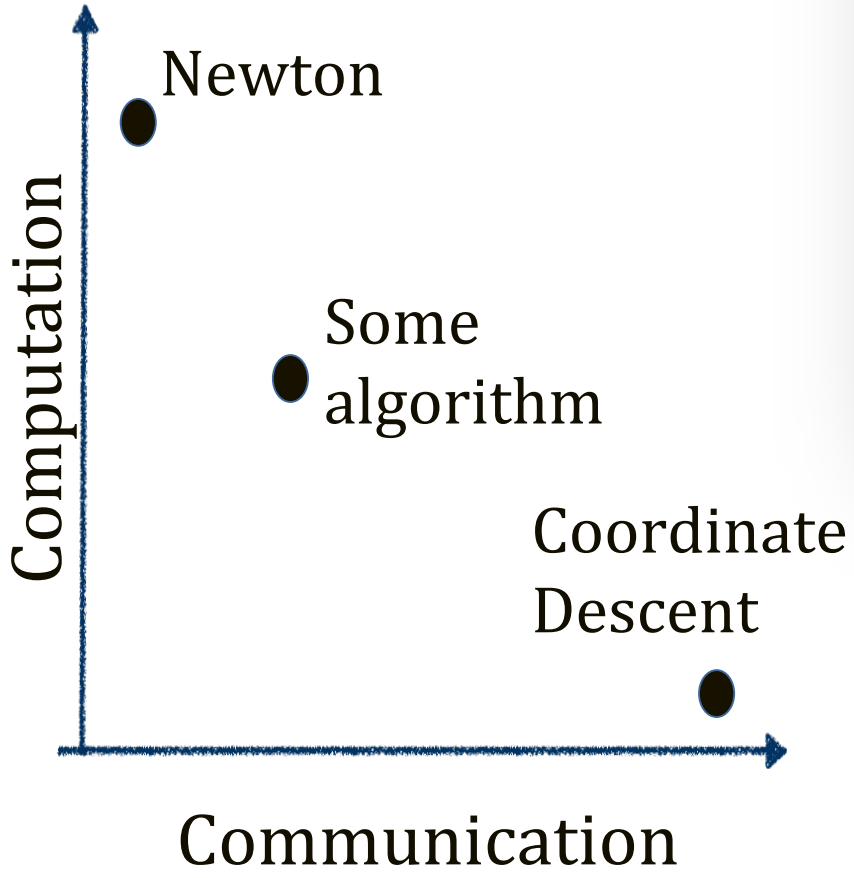
Spark and Spark performance

Alchemist: an Apache Spark <=> MPI Interface

Communication-avoiding Machine Learning

Current and Future Directions
- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

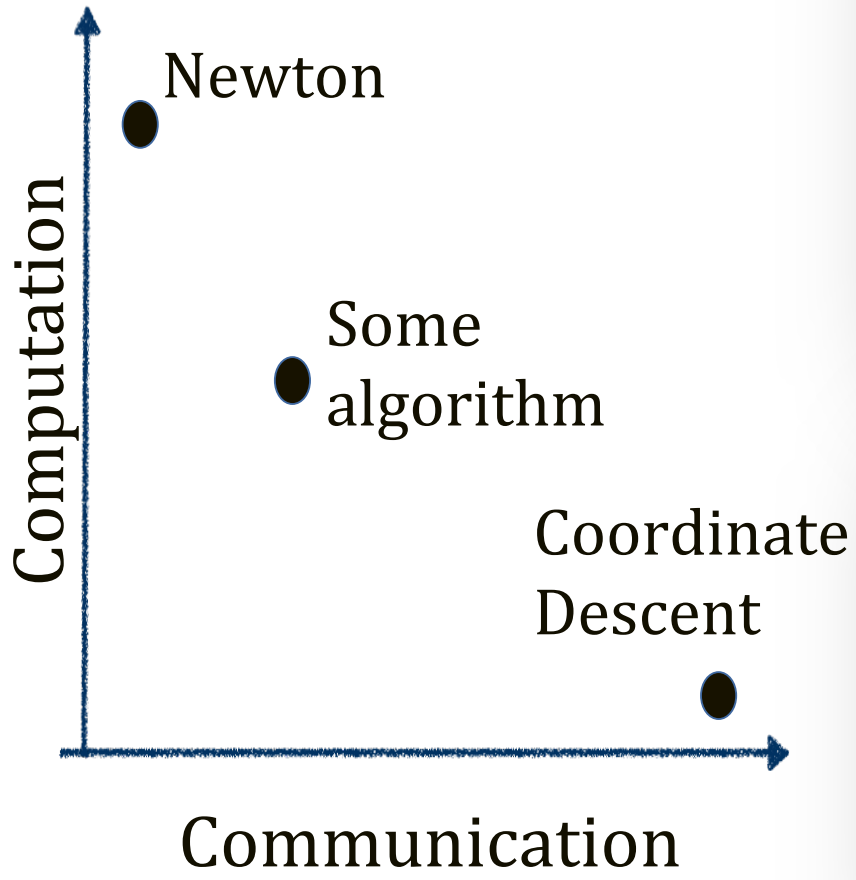
- Trade-offs and existing approaches



Current approach:

choose an algorithm based on computation and communication trade-off

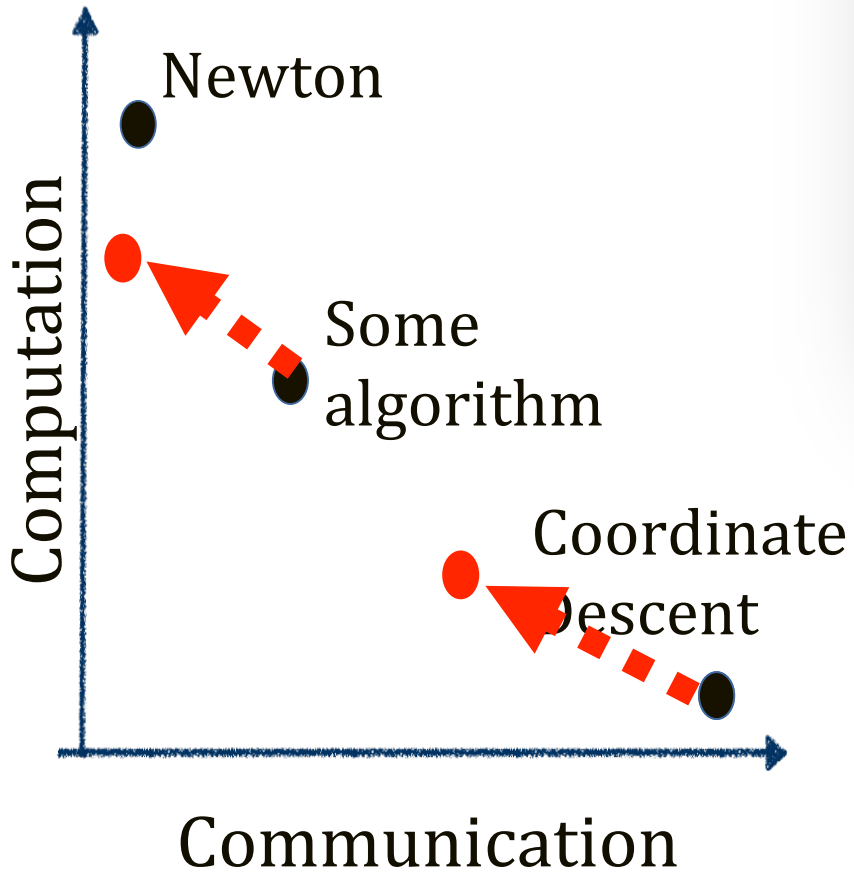
- Trade-offs and existing approaches



What happens if there is no algorithm with the required trade-off?

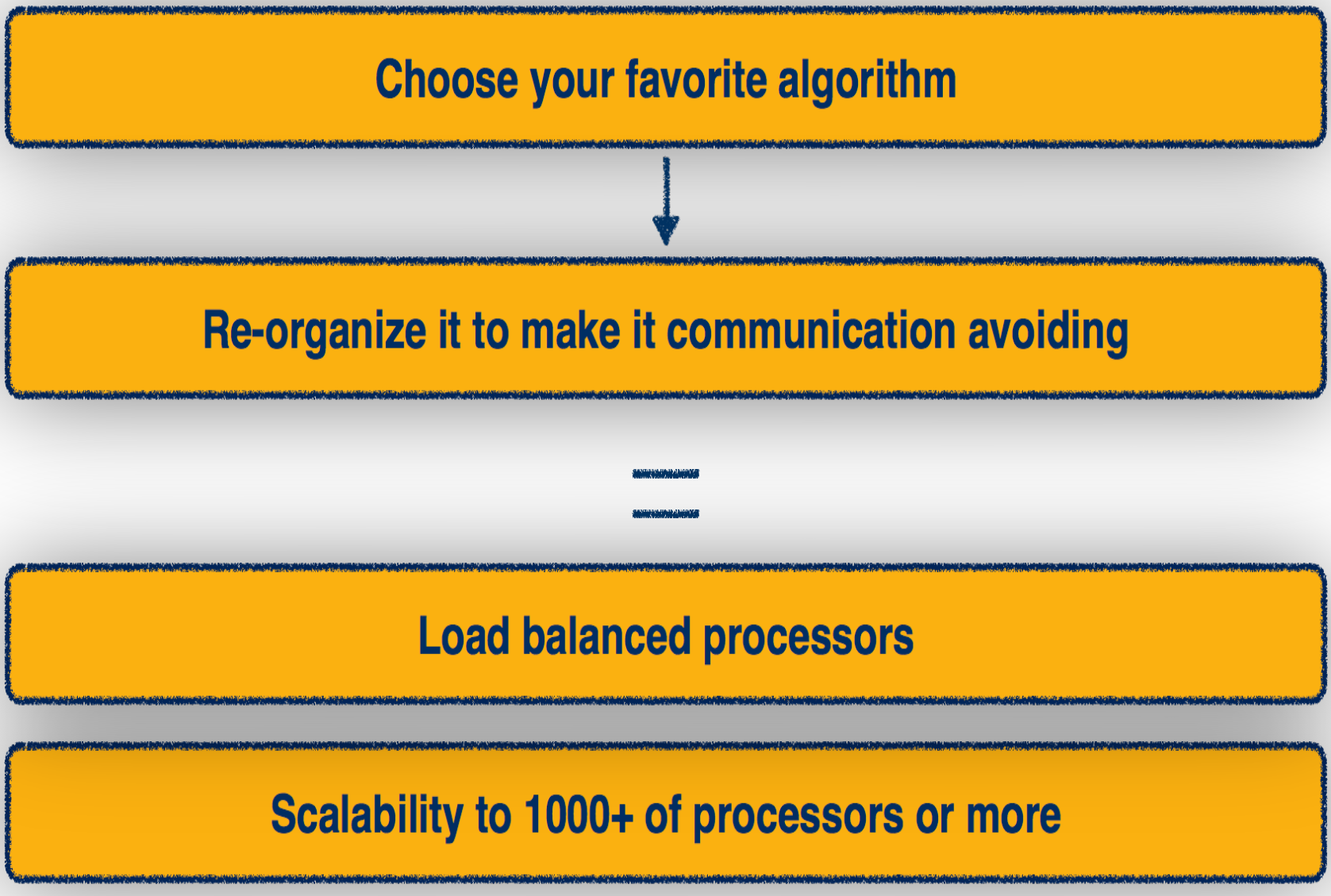
We need to wait until a mathematician comes up with a solution

- Our approach

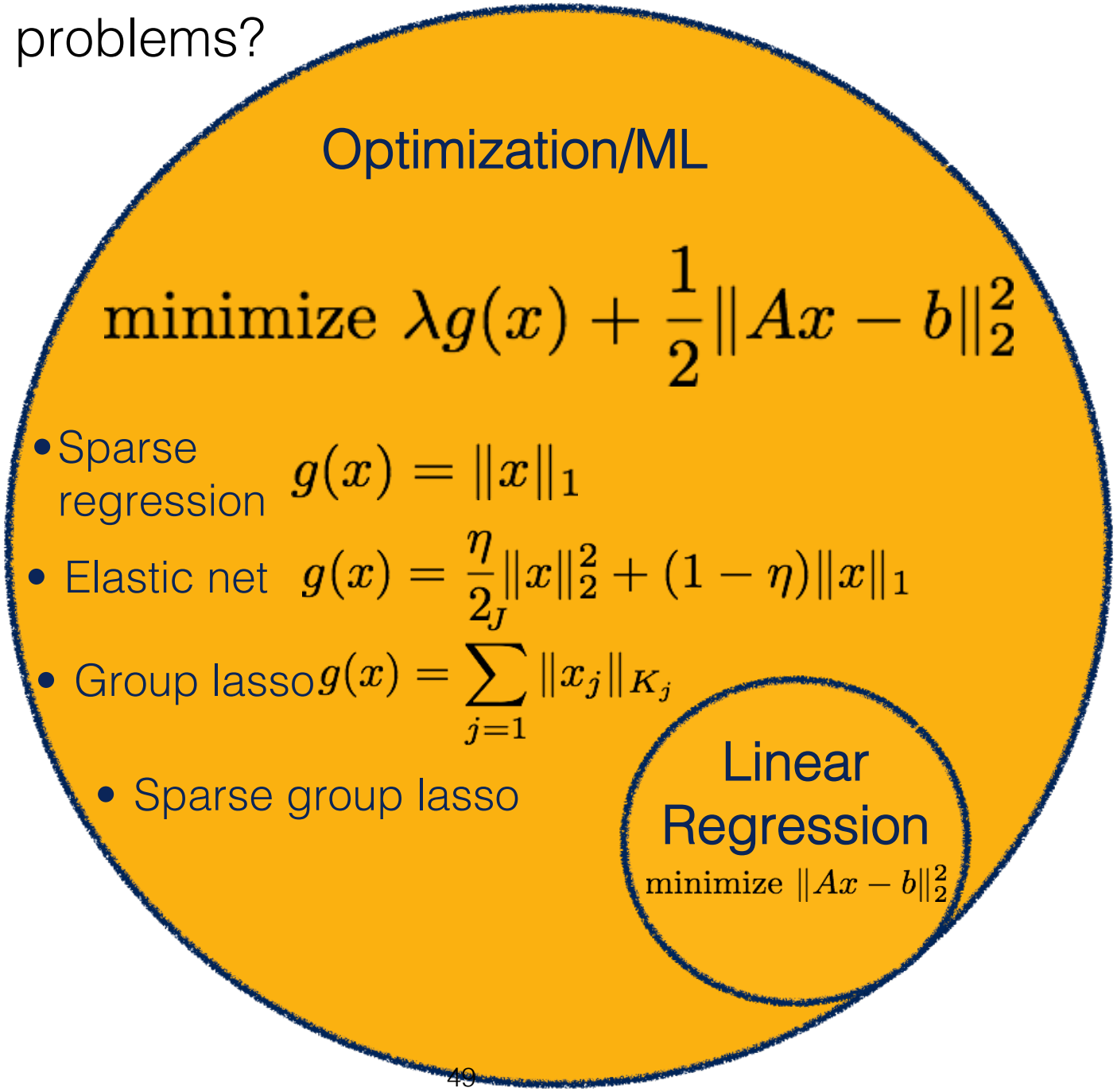


Take existing algorithms and make them communication avoiding

- Outline of the approach and results



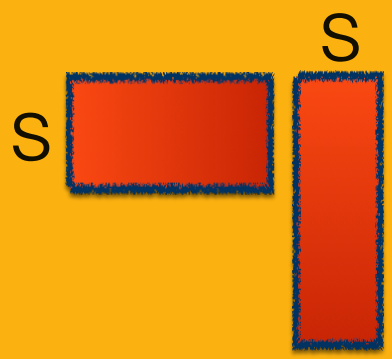
- For what problems?



- How to avoid communication: basic idea

Coordinate descent is a recurrence: unroll “s” iterations

Define the anticipated computations to perform “s” iterations. In this case that is matrix multiplication.



Compute matrix multiplication in parallel **by communicating only once**

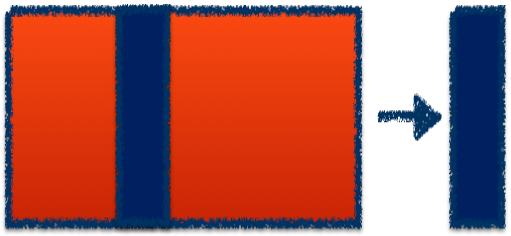
Compute “s” iterations redundantly on each processor

- An example: coordinate descent

Pseudo-code

1 communication per iteration

- Sample a column of data



- Compute partial derivative

- Update solution

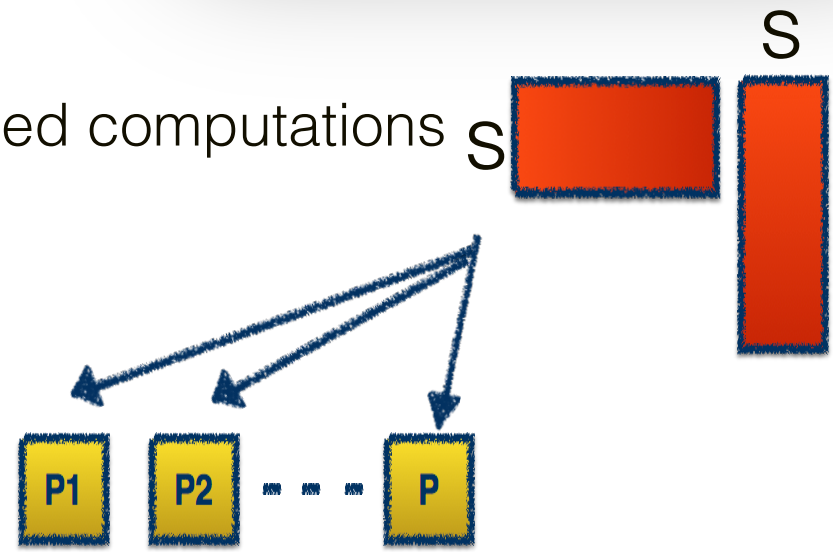
- Repeat

- An example: communication avoiding coordinate descent

Pseudo-code

1 communication round per s iterations

- Compute in parallel anticipated computations for the next “s” iterations
- Redundantly store the result in all processors



- Each processor independently computes the next “s” iterations
- Repeat

- More details about the results

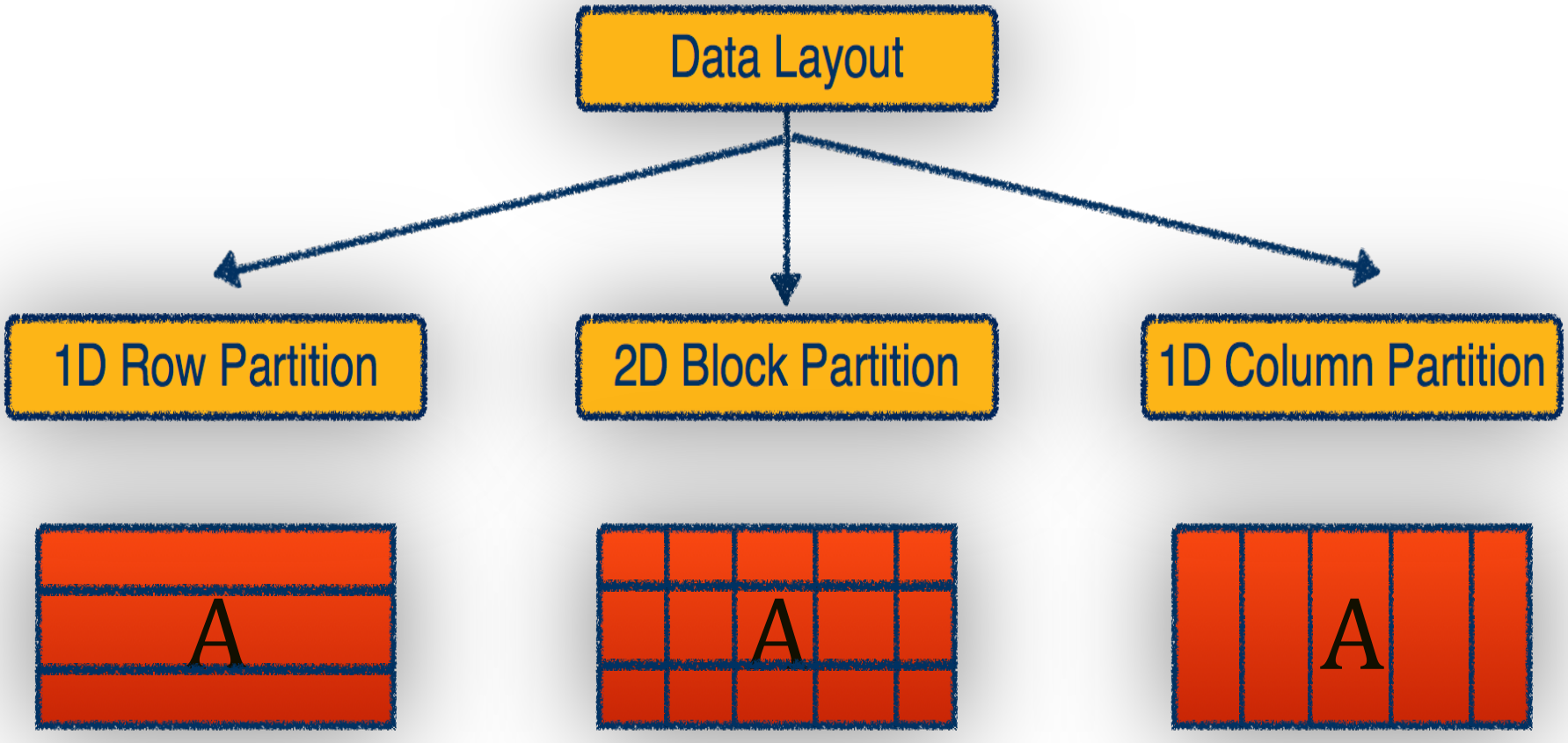
Decrease communication by a factor of s

No free lunch: increase message size and flops by a factor of s

Flops are distributed across processors

Logarithmic dependence of communication cost **on number of processors**

- Scalable results for all data layouts



* Best performance depends on dataset and algorithm

- Datasets

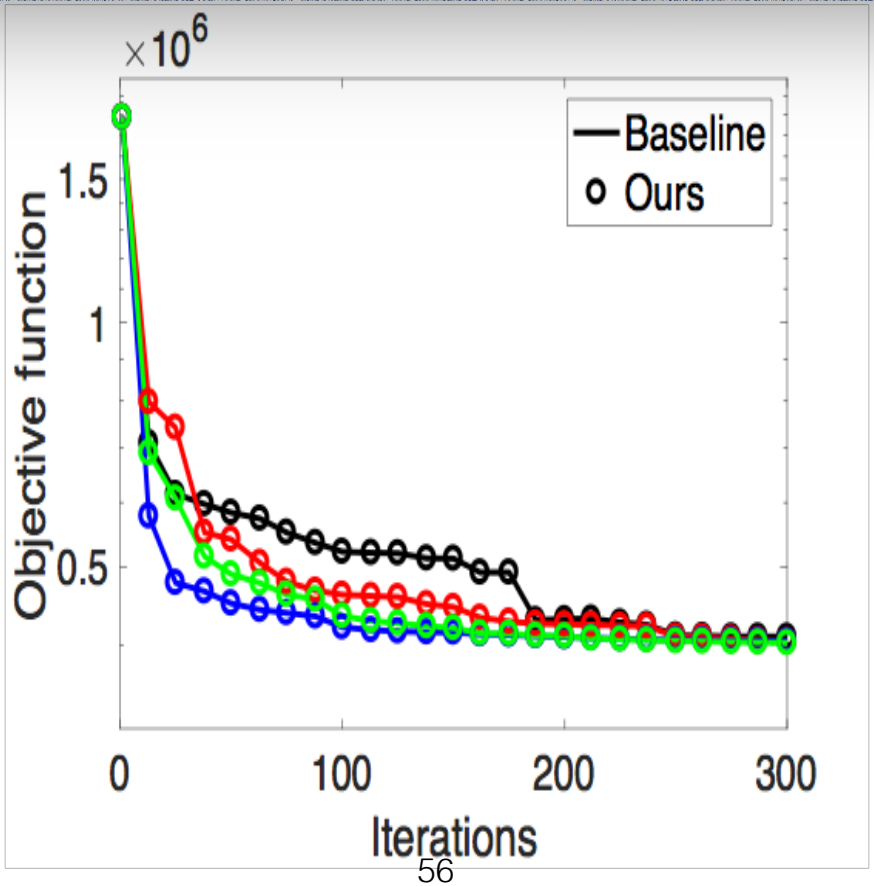
Summary of (LIBSVM) datasets

Name	#Features	#Data points	Density of non-zeros
url	3,231,961	2,396,130	0.0036%
epsilon	2,000	400,000	100%
news20	62,021	15,935	0.13%
covtype	54	581,012	22%

C++ using the Message Passing Interface (MPI). Intel MKL library for sparse and dense BLAS routines. All methods were tested on a Cray XC30.

- Convergence of re-organized algorithms

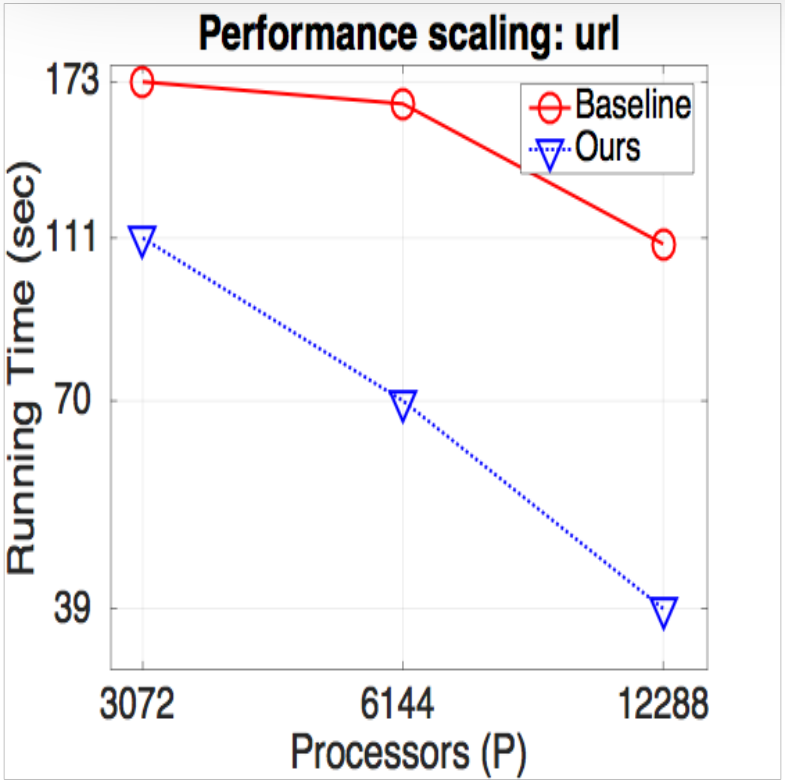
Convergence rate remains the same in exact arithmetic
Empirically stable convergence: no divergence between methods

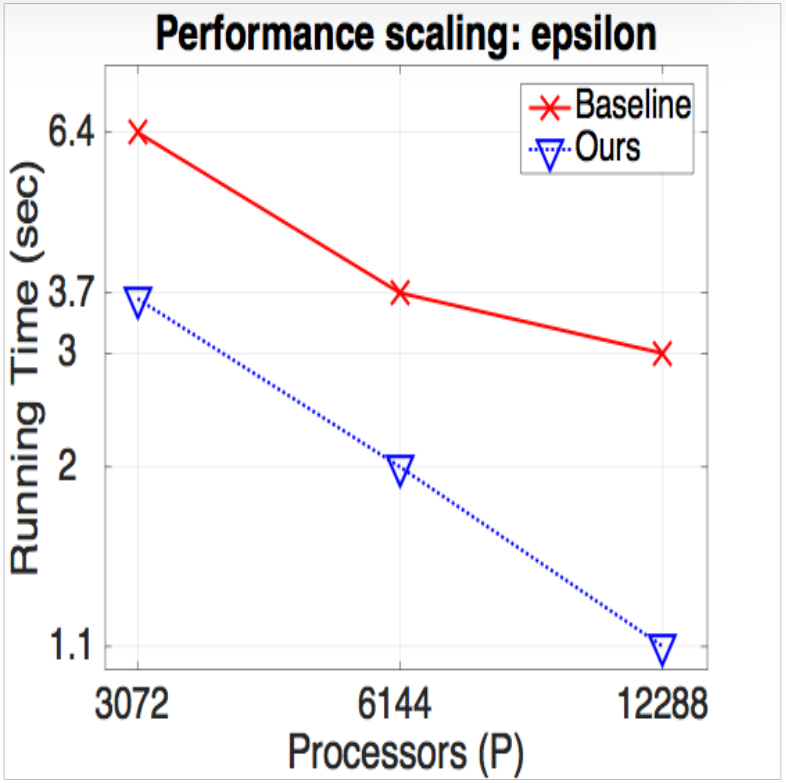


- Scalability performance

The more processors the better

The gap between CA and non-CA increases w.r.t. #processors

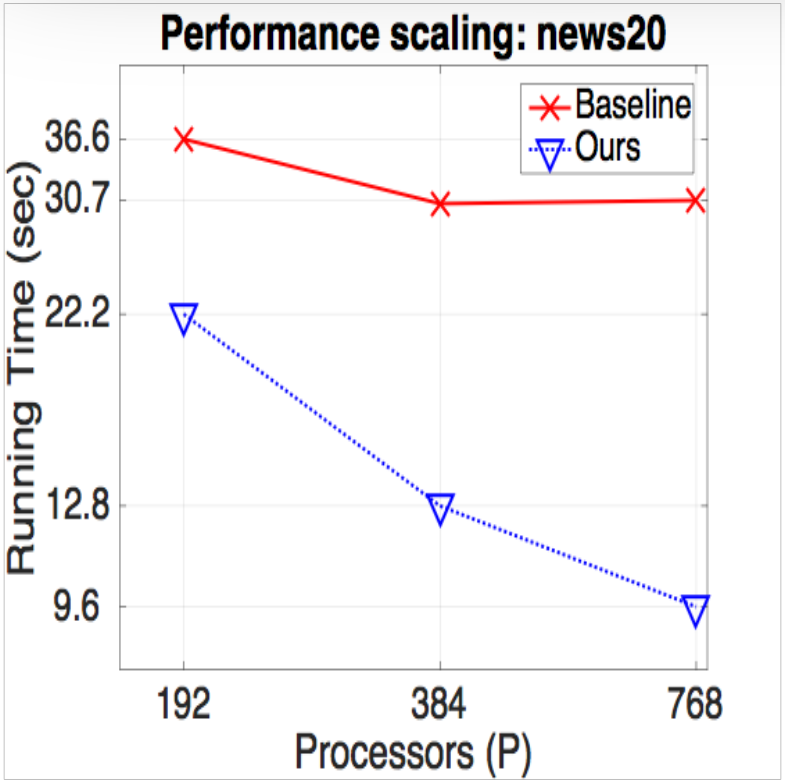


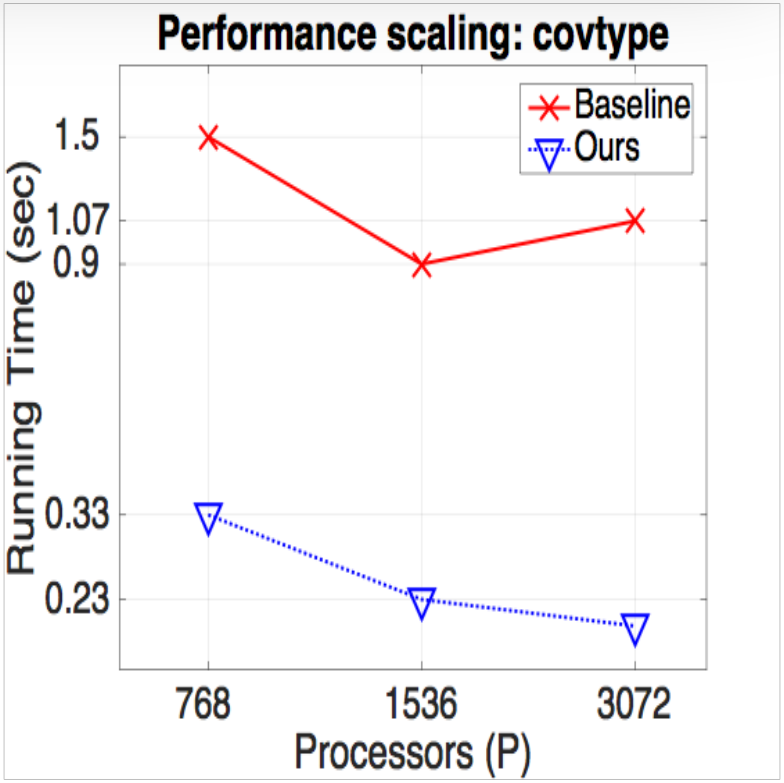


- Scalability performance

The more processors the better

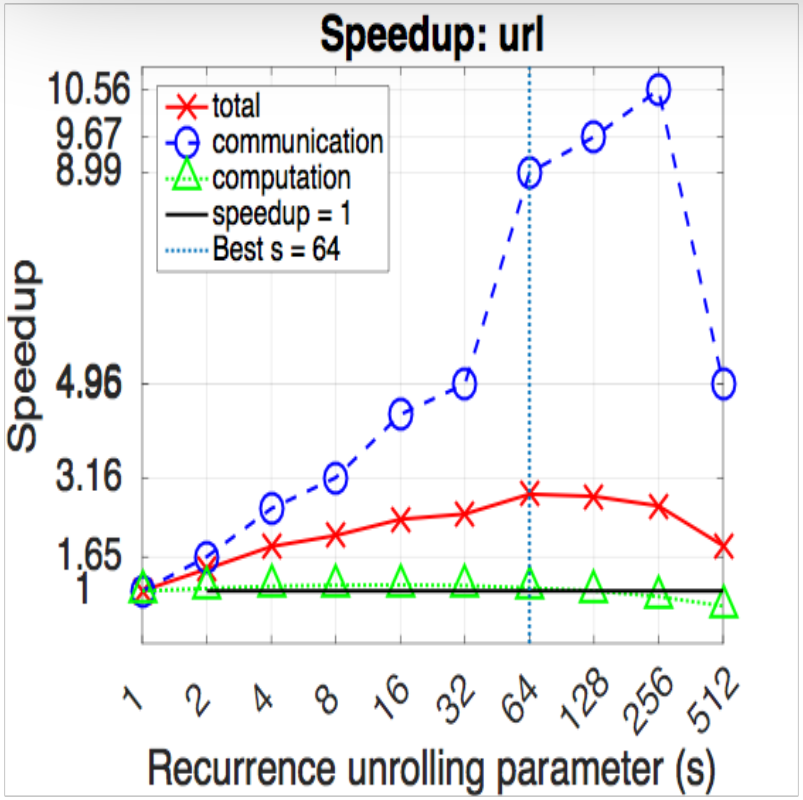
The gap between CA and non-CA increases w.r.t. #processors

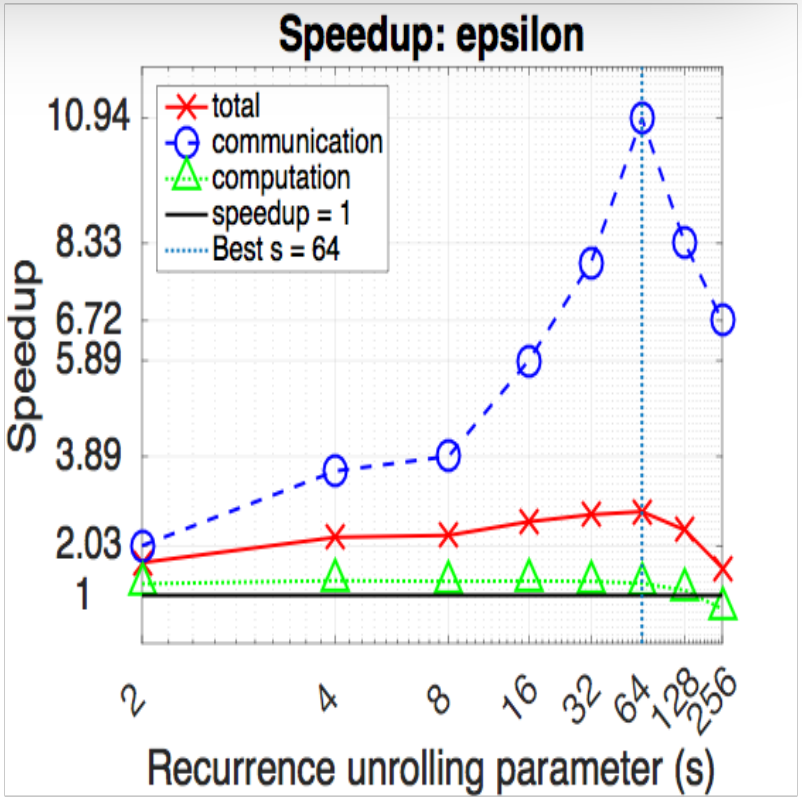




- Speed up breakdown

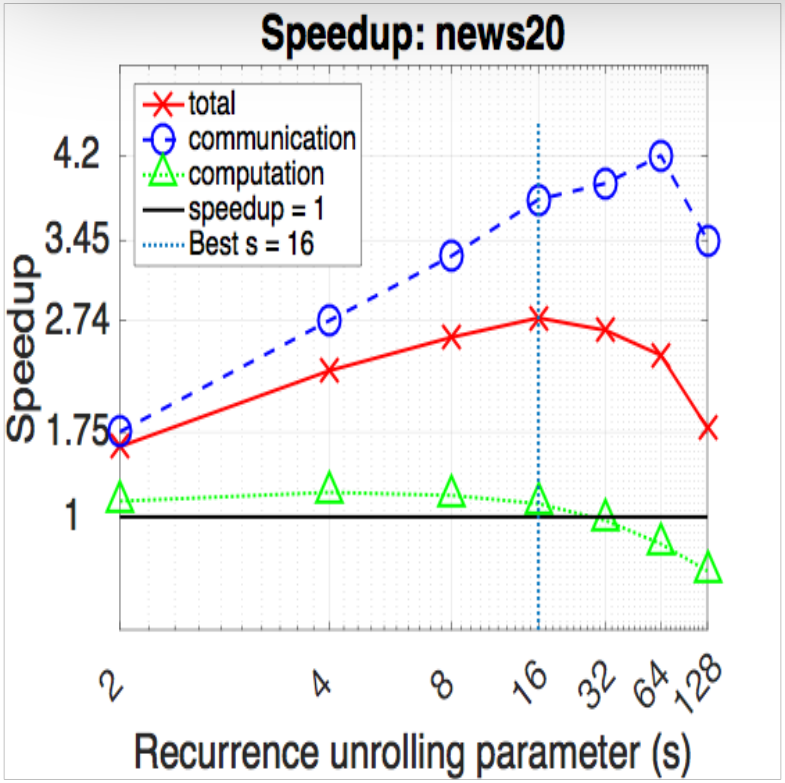
Large communication speedup until bandwidth takes a hit
Computation is maintained due to local cache-efficient (BLAS-3) computations

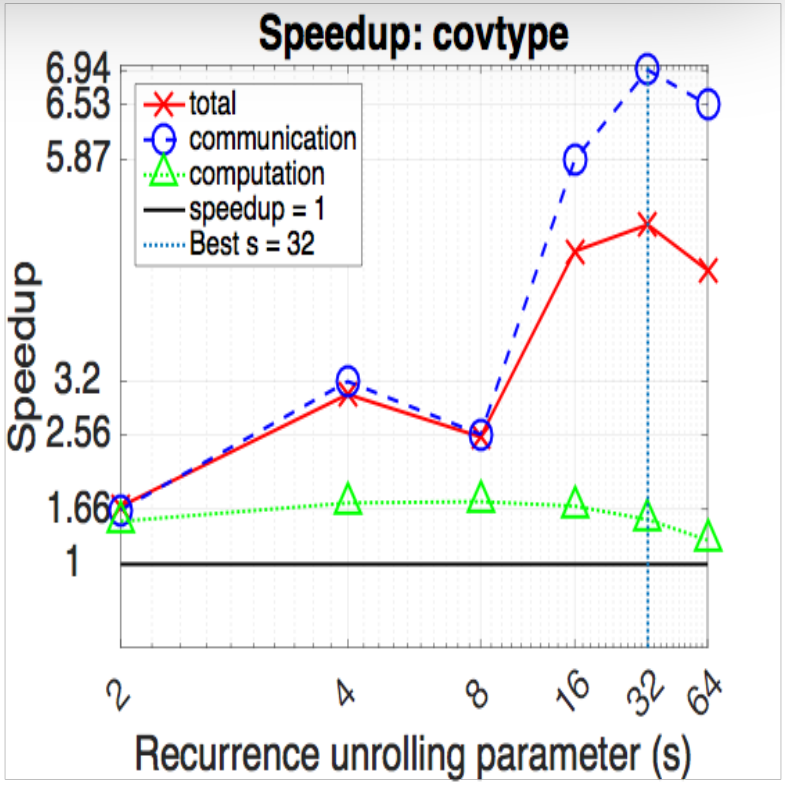




- Speed up breakdown

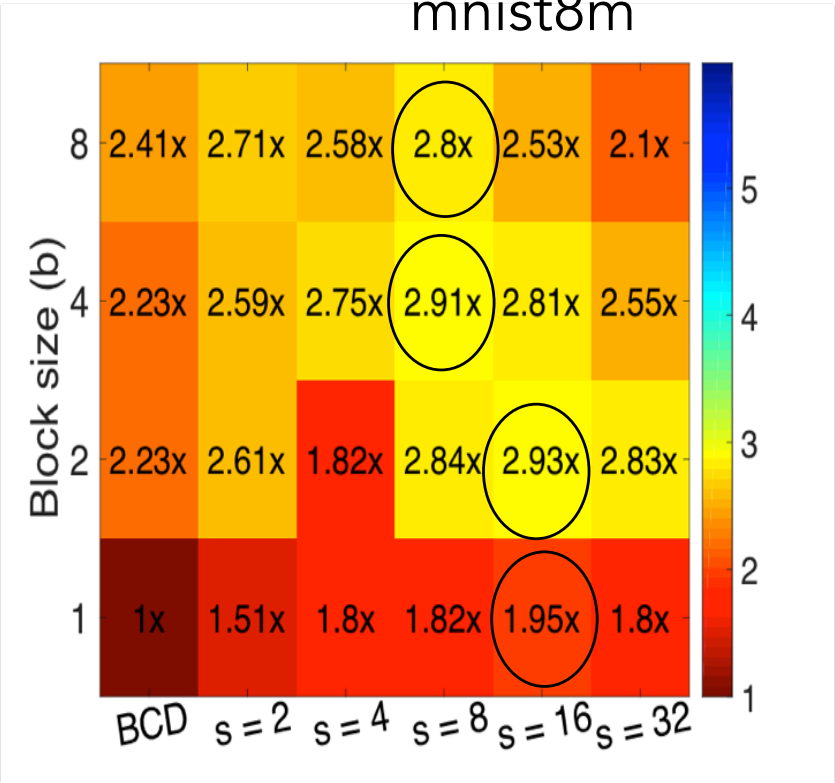
Large communication speedup until bandwidth takes a hit
Computation is maintained due to local cache-efficient (BLAS-3) computations



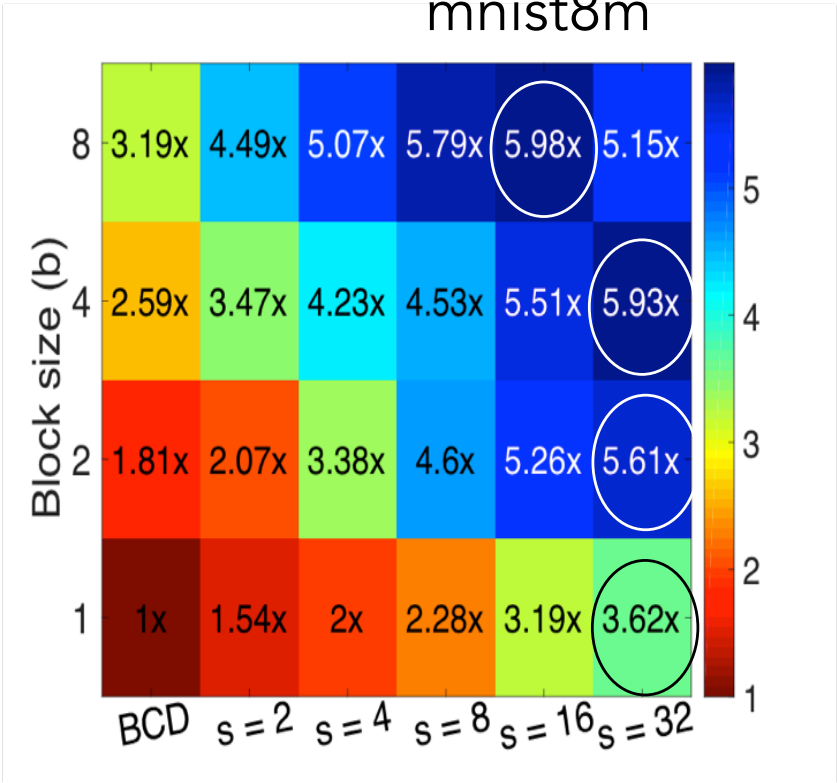


Circles = best speedups

Speedups (Ridge Regression)



64 nodes of Edison



1K nodes of Edison

Joint with A. Devarakonda, K. Fountoulakis, and J. Demmel

- Other examples

  - Block coordinate descent
  - Accelerated block coordinate descent
  - Gradient descent
  - Any proximal method

- Summary

Generalize from linear algebra to optimization/ML

Provably avoid communication

Scalability to 10,000+ processors

Applies to many algorithms

Overview

Thoughts on Machine Learning, scientific Machine Learning, etc.

Spark and Spark performance

Alchemist: an Apache Spark <=> MPI Interface

Communication-avoiding Machine Learning

Current and Future Directions

- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

A jupyter/ipython + MPI interface

"Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages."

Most of the functionality is in place.

- Once the Alchemist rewrite is complete, *should* be straightforward.
- For simple workflows, e.g., load a large dataset from a file, do some computations, and then return the results.
- More complex, e.g, if there are special kind of distributed matrices that are being used by the jupyter/ipython application

Extend from a Spark <=> MPI interface to a X <=> MPI interface.

RISELab's Ray Project

A platform for high-performance distributed execution

- Philipp Moritz, Robert Nishihara, Richard Liaw, Ion Stoica

Problem: AI/ML applications require more than neural networks

- Simulations, streaming and processing sensor data, serving decisions, rendering actions (e.g., actuate robotic joins)
- ML practitioners often build their own systems infrastructure in addition to deep learning frameworks

Ray is a distributed execution framework for AI applications.

- High throughput, low latency tasks
- Fine-grained, nested, heterogeneous tasks

Ray provides Task parallel API and actor API built on dynamic task graphs

- These APIs are used to build distributed **applications**, **libraries** and **systems**

(They want input---we coordinate with them as Alchemist winds up.)

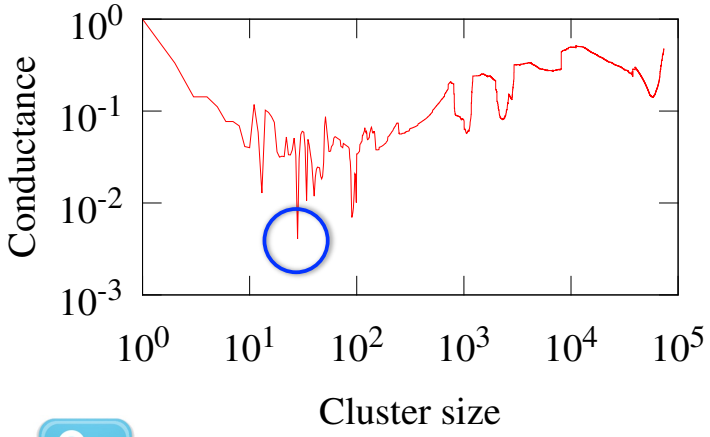
Large-scale Graph Processing

"Parallel Local Clustering Algorithms"

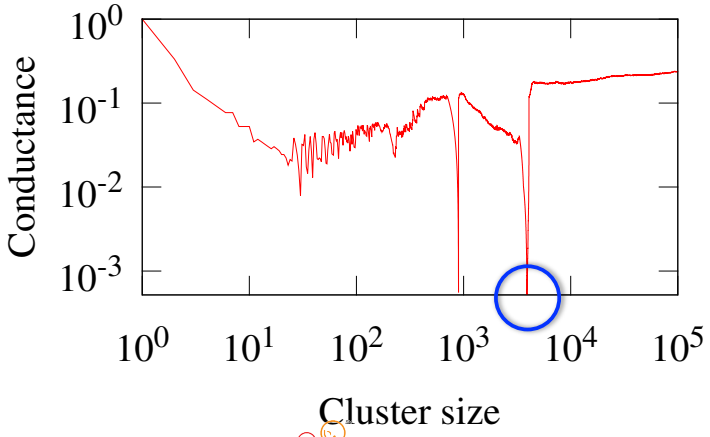
- Shun, Roosta-Khorasani, Fountoulakis, and Mahoney (arXiv:1604.07515)
- Code: https://github.com/jshun/ligra/

"LocalGraphClustering"

- Fountoulakis, Gleich, Mahoney (Proc IEEE 2017)
- Code: https://github.com/kfoynt/LocalGraphClustering



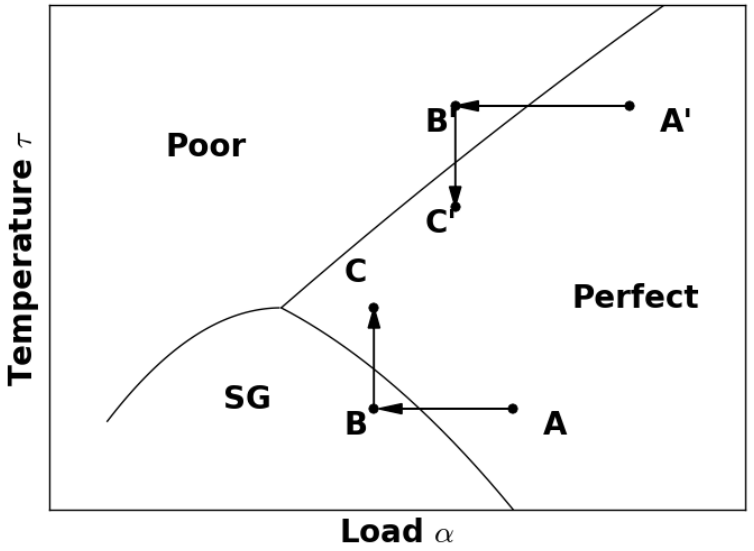
42M vertices
1.2B edges

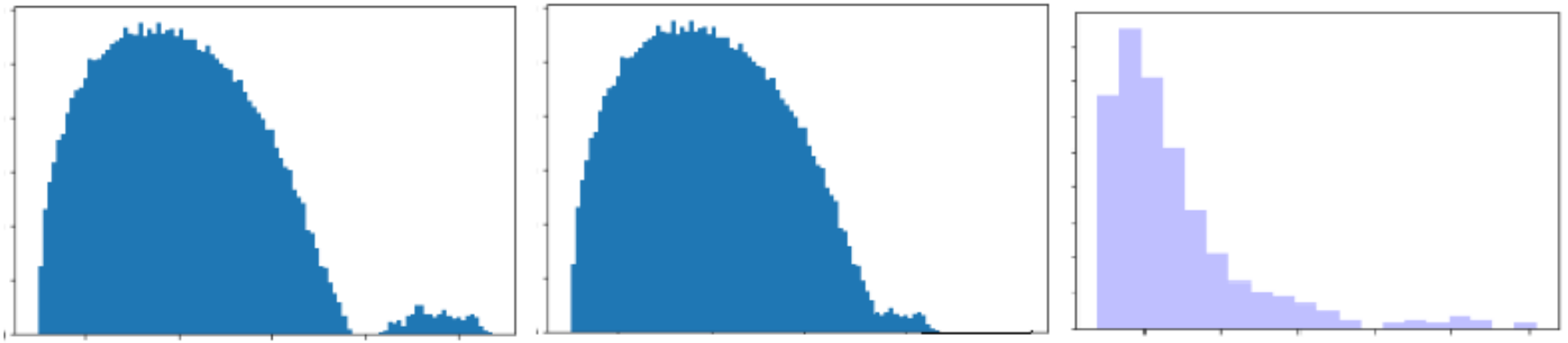


125M vertices
1.8B edges

Neural Network Learning

- Characterize properties on NN learning algorithms i.t.o. hyperparameters
- Develop interpretable NNs that incorporate domain science
- Provide GPU implementations of stochastic optimization algorithms

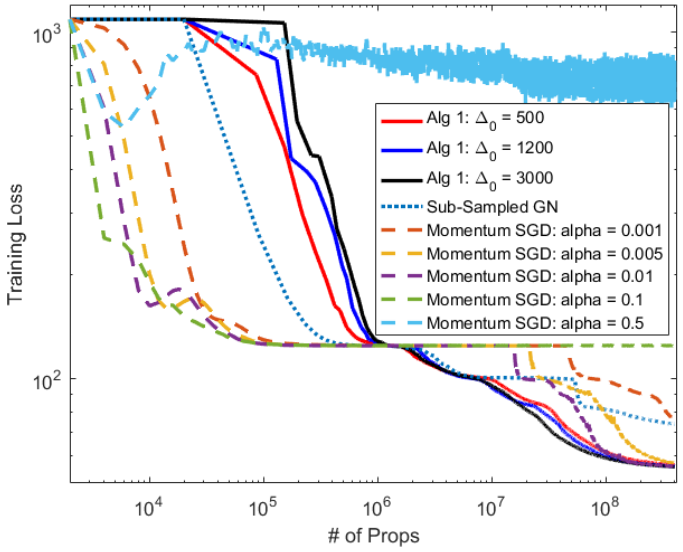


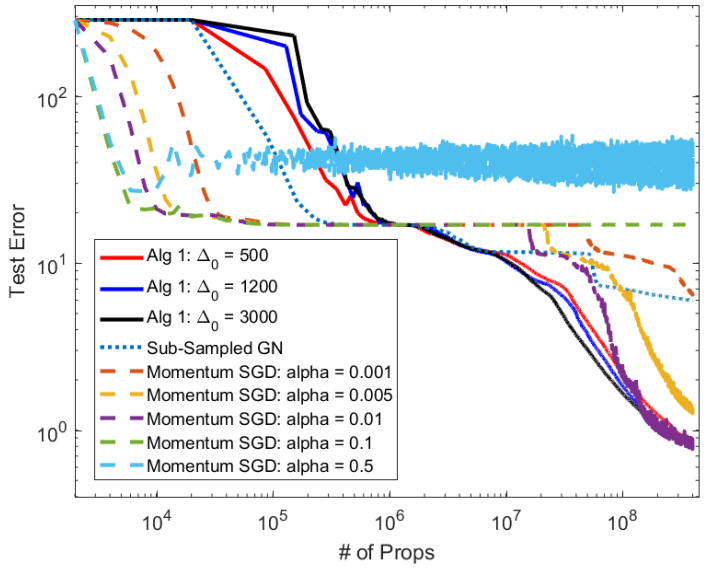


Scalable Second-order Optimization Methods

Use second derivative information: more expensive & more powerful

- Resiliency to problem ill-conditioning
- Good generalization error and robustness to hyper-parameter tuning





- Ability to escape undesirable saddle-points
- Low-communication costs in distributed settings
- Computational advantages offered by leveraging the power of GPUs

Conclusion

Goal
- Do computationally-intensive scientific machine learning at scale

Progress Report
- Characterize performance loss
- Develop Alchemist: An Apache Spark <=> MPI Interface

Future Directions
- A jupyter/ipython + MPI interface
- RISELab's Ray Project
- Large-scale Graph Processing
- Neural Network Learning
- Scalable Second-order Optimization Methods

And THANKS to Cray, DARPA, and NSF for financial support!