Terabyte-scale Computational Statistics

Michael W. Mahoney

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

November 2016

Overview

Thoughts on data, large data, massive data, big data, etc.

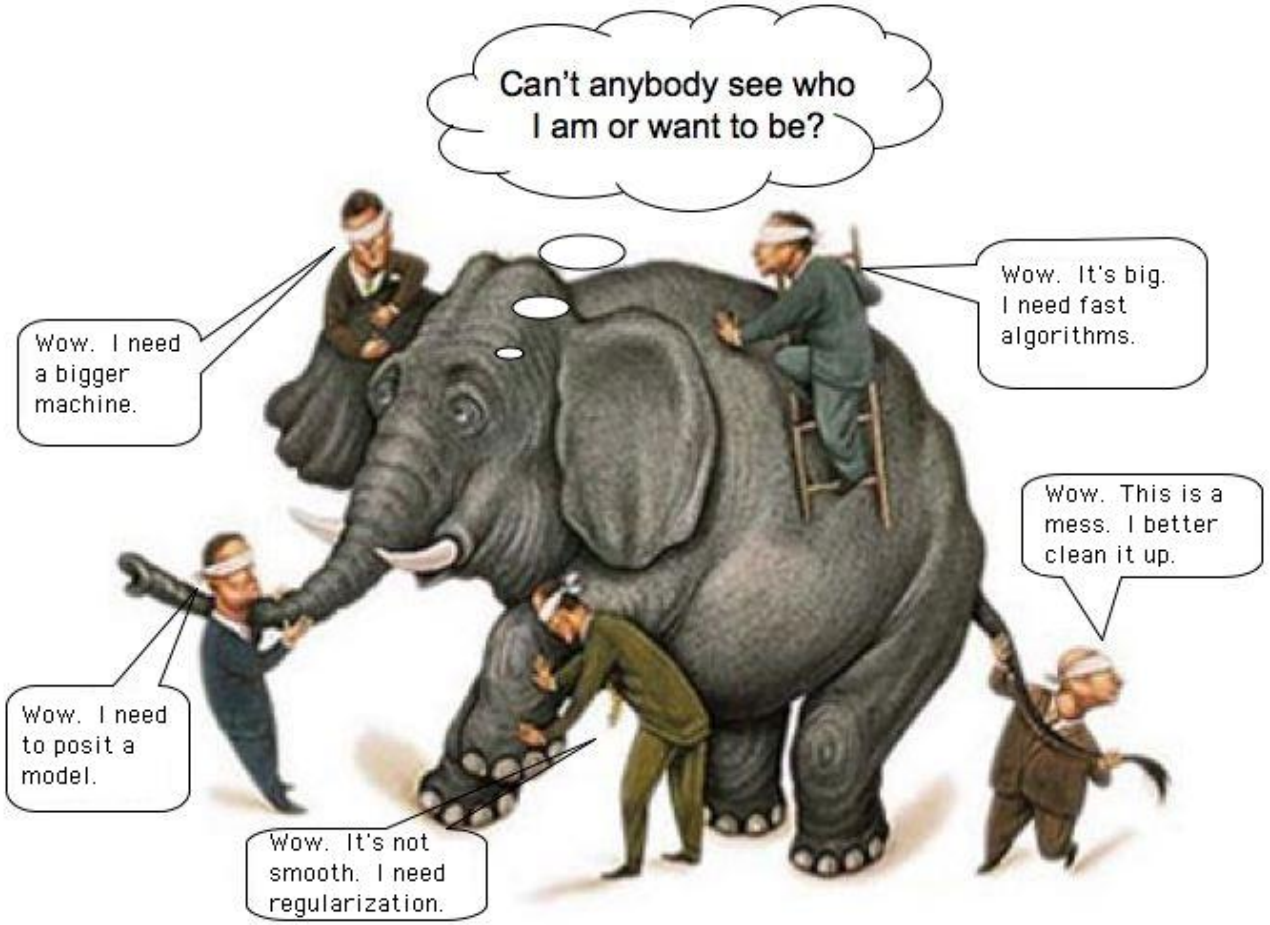
Thoughts on scientific data: problems, running times, and choosing good columns as features.

Linear Algebra in Spark for science problems

- CX and SVD/PCA implementations and performance
- Applications of the CX and PCA matrix decompositions
- To mass spec imaging, climate science, etc.

The Next Step: Alchemist

How do we view BIG data?



Algorithmic & Statistical Perspectives ...

Lambert (2000), Mahoney (2010)

Computer Scientists

- *Data*: are a record of everything that happened.
- *Goal*: process the data to find interesting patterns and associations.
- *Methodology*: Develop approximation algorithms under different models of data access since the goal is typically computationally hard.

Statisticians (and Natural Scientists, etc)

- *Data*: are a particular random instantiation of an underlying process describing unobserved patterns in the world.
- *Goal*: is to extract information about the world from noisy data.
- *Methodology*: Make inferences (perhaps about unseen events) by positing a model that describes the random variability of the data around the deterministic model.

... are VERY different **paradigms**

Statistics, natural sciences, scientific computing, etc:

- Problems often involve computation, but the study of computation *per se* is secondary
- Only makes sense to develop algorithms for well-posed* problems
- First, write down a model, and think about computation later

Computer science:

- Easier to study computation *per se* in discrete settings, e.g., Turing machines, logic, complexity classes
- Theory of algorithms divorces computation from data
- First, run a fast algorithm, and ask what it means later

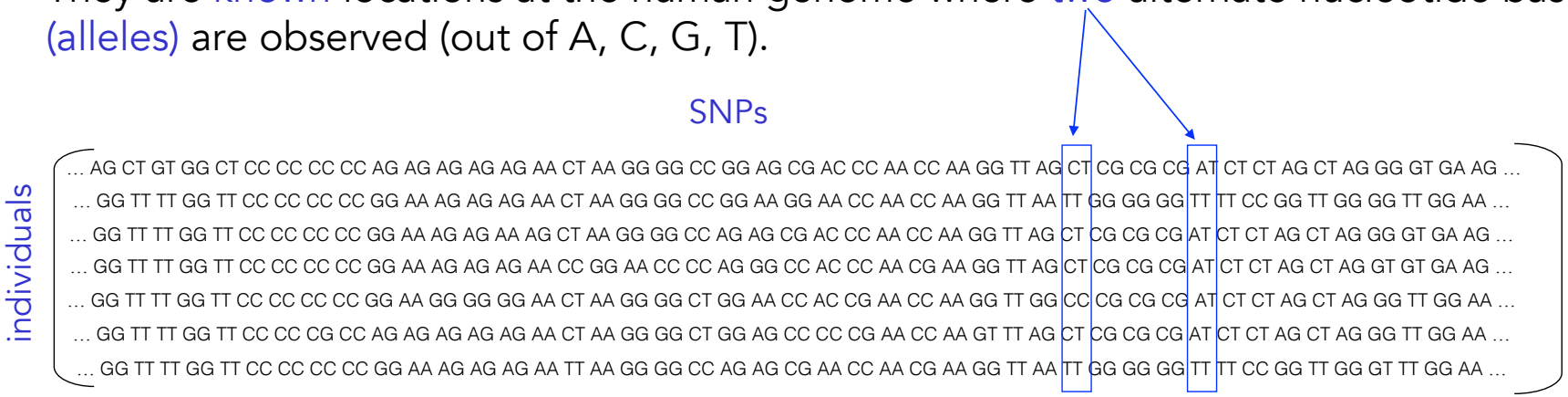
*Solution exists, is unique, and varies continuously with input data

Scientific data and choosing good columns as features

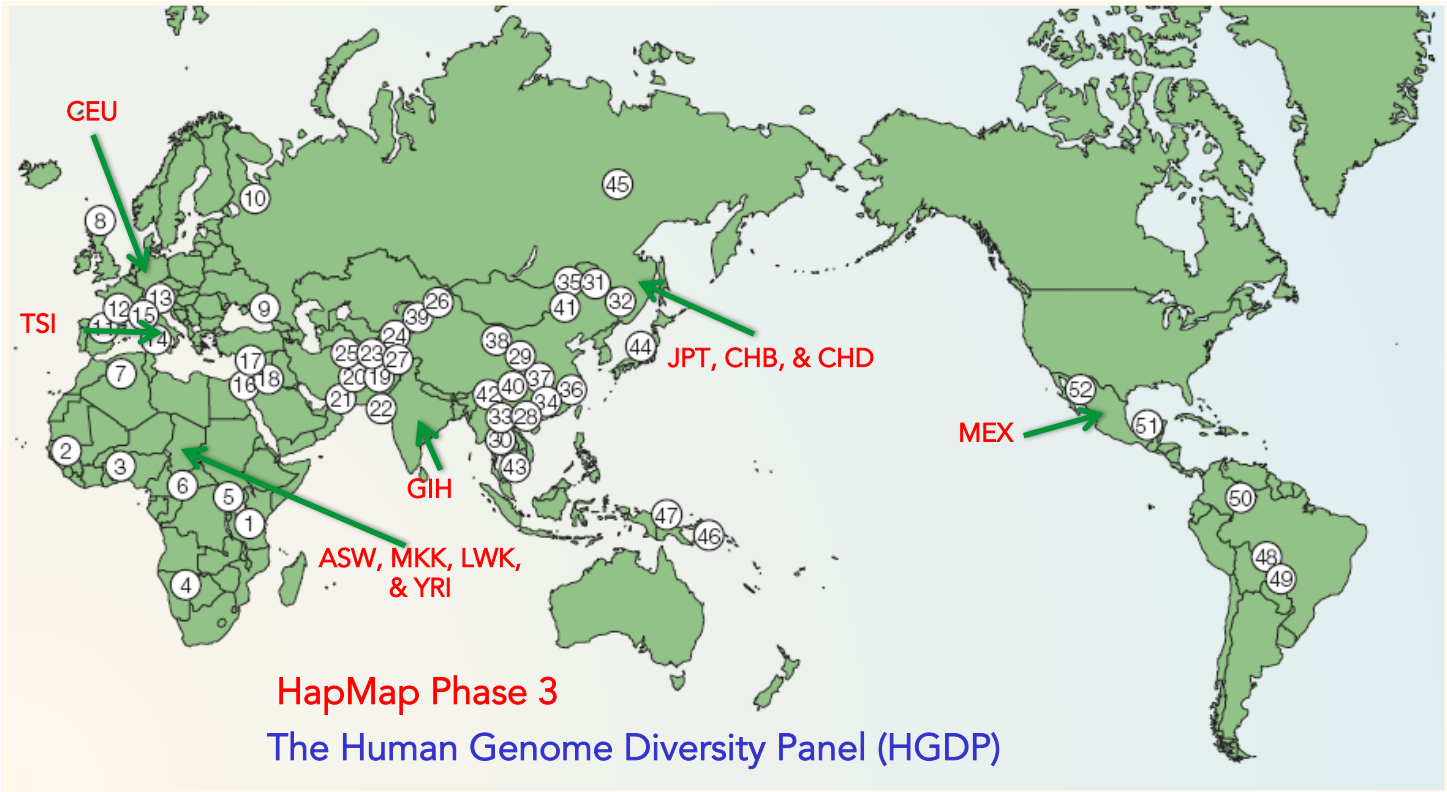
E.g., application in: Human Genetics

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:

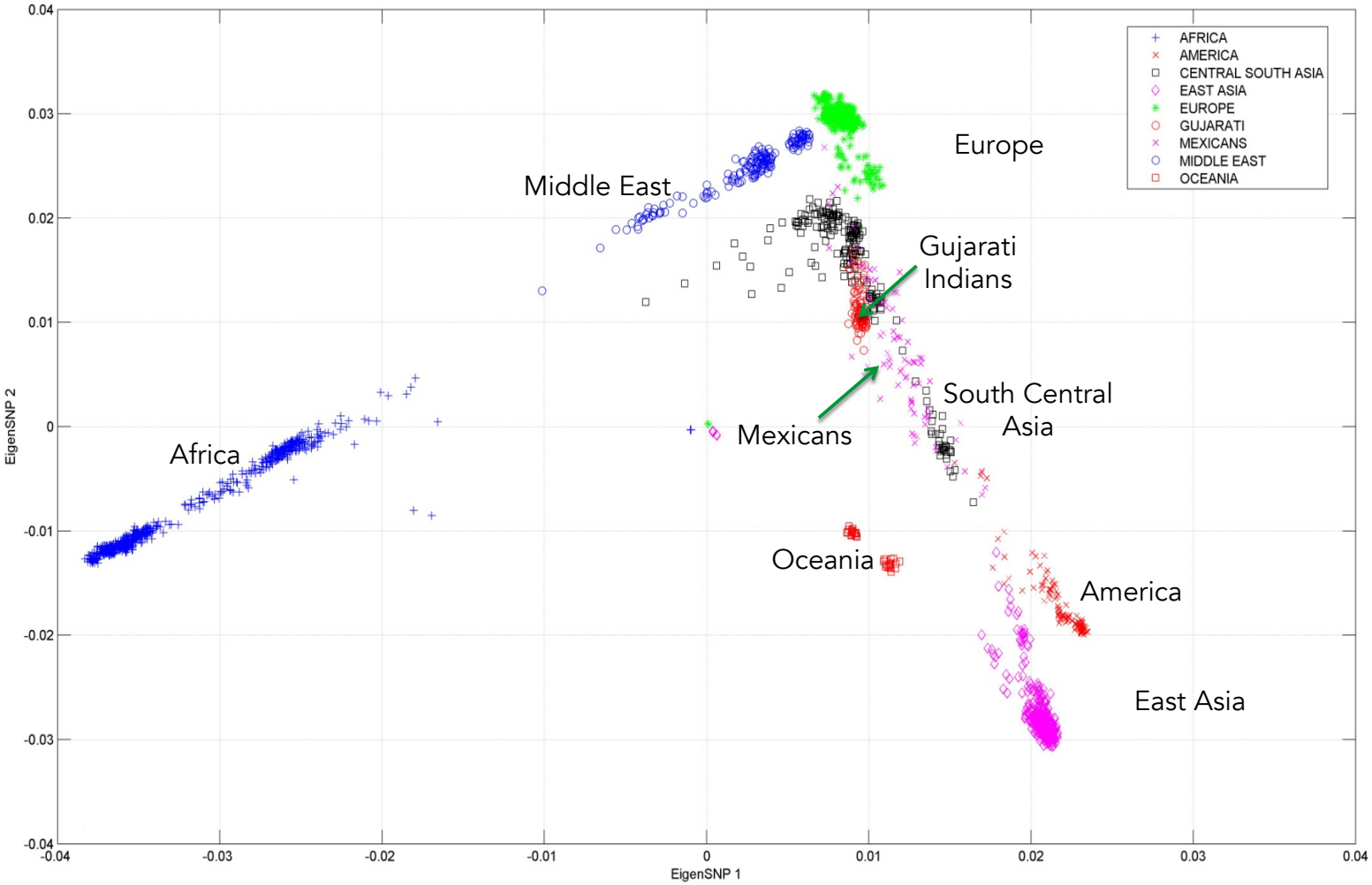
2,240 subjects (rows)

447,143 SNPs (columns)

Dense matrix:

over one billion entries

Paschou, et al (2010) J Med Genet



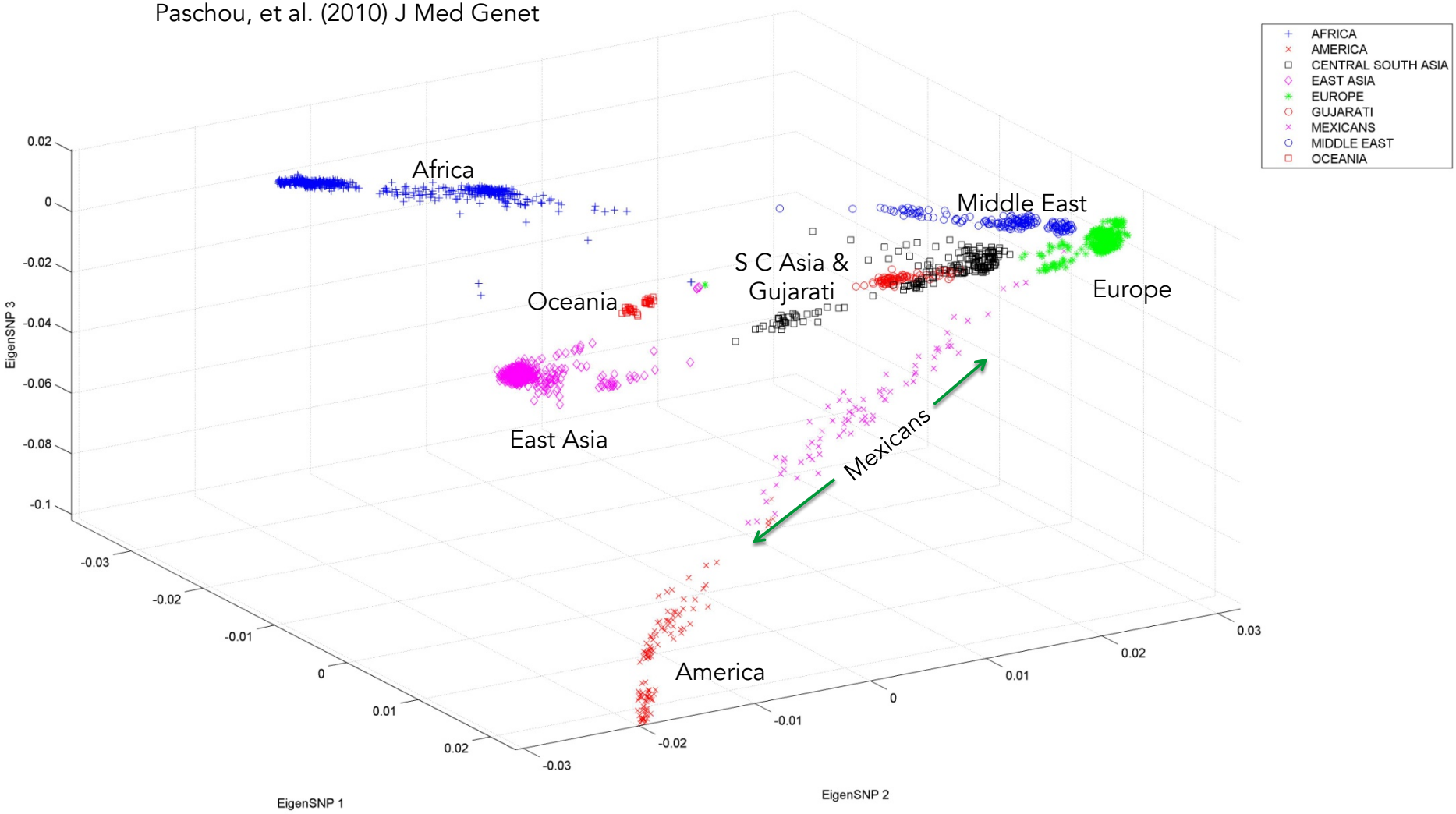
• Top two Principal Components (PCs or eigenSNPs)

(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))

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%

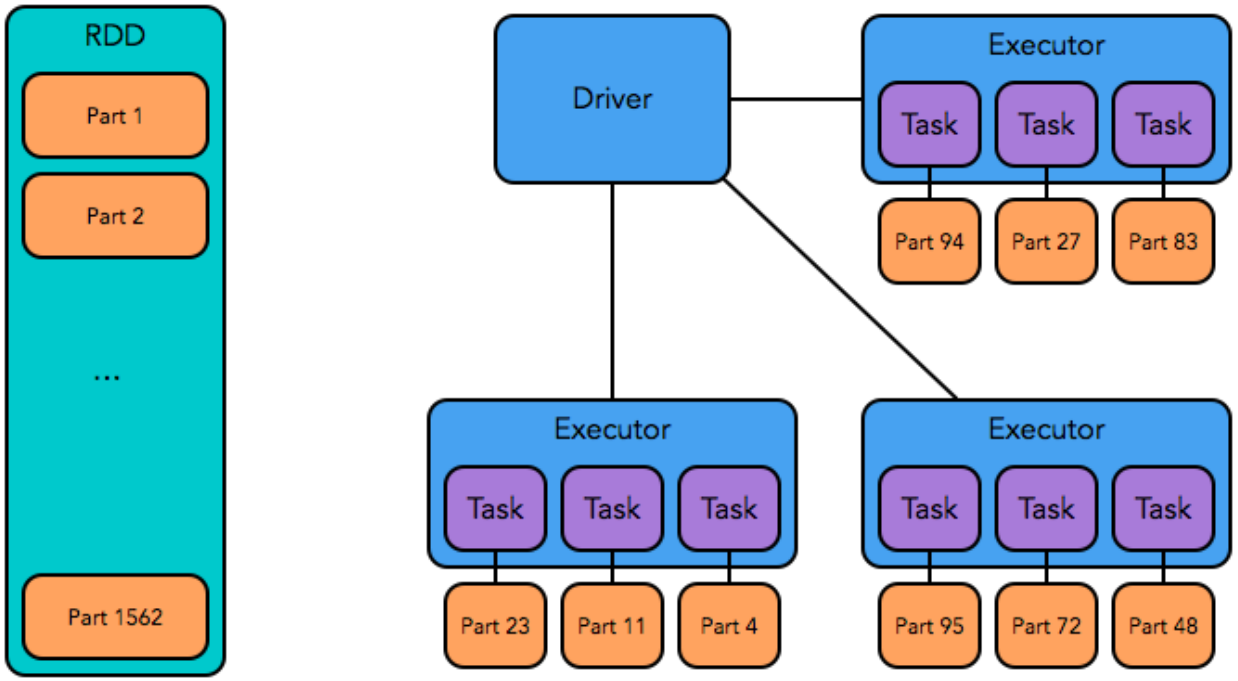
Supercomputer

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

Distributed data center

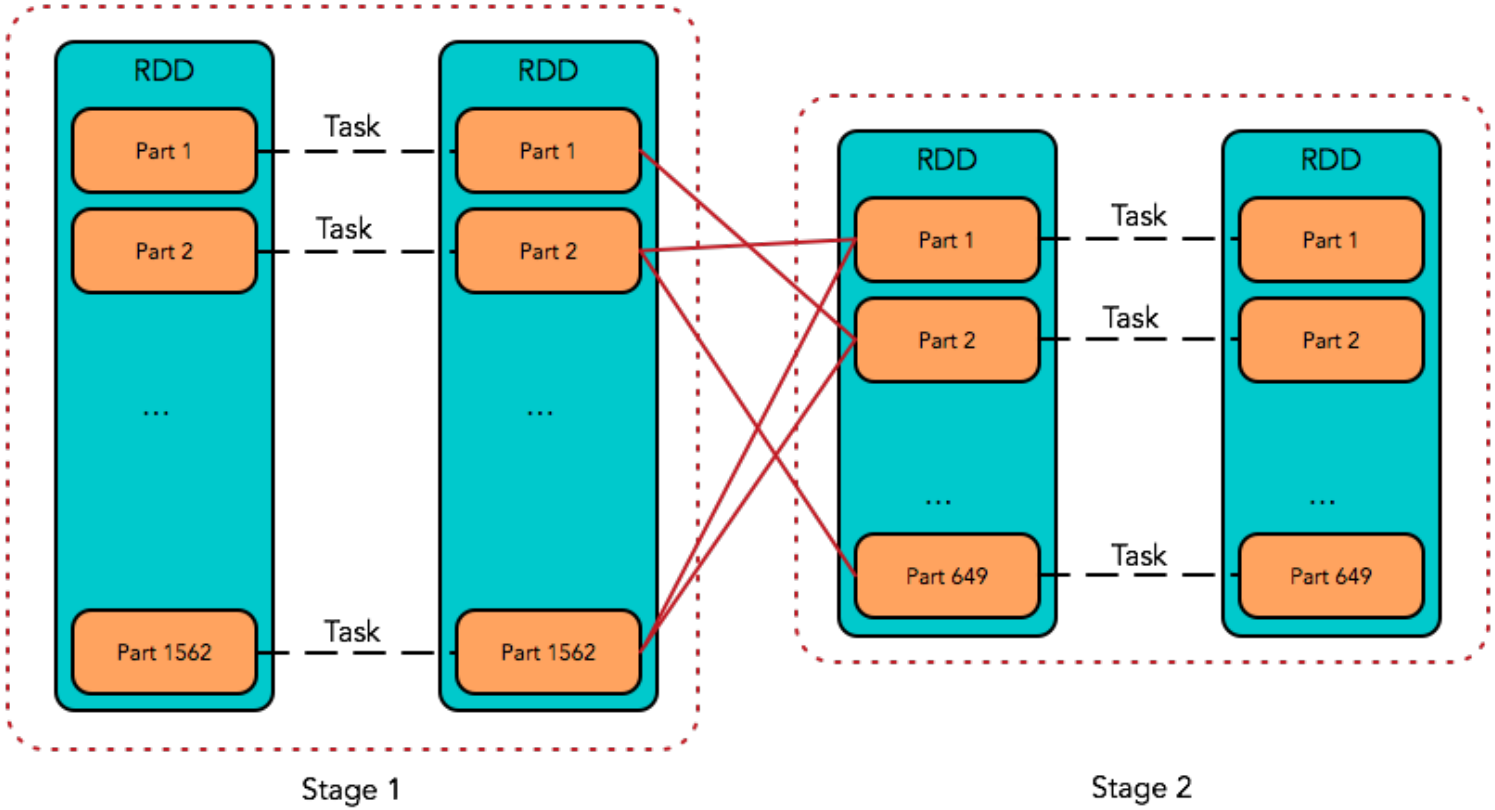
- High end, data-intensive
- BIG emphasis on HPC (High Productivity Computing)
- Databases, MapReduce/Hadoop, Spark, etc.

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:

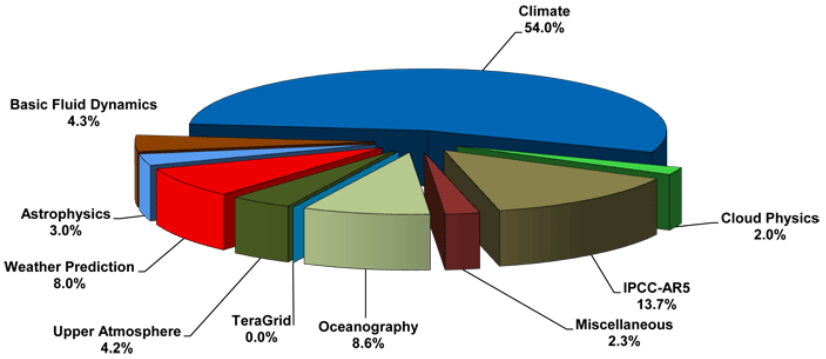
- Classical MPI-based linear algebra algorithms are faster and more efficient
- No way, currently, to leverage legacy parallel linear algebra codes
- JVM matrix size restrictions, and RDD rigidity

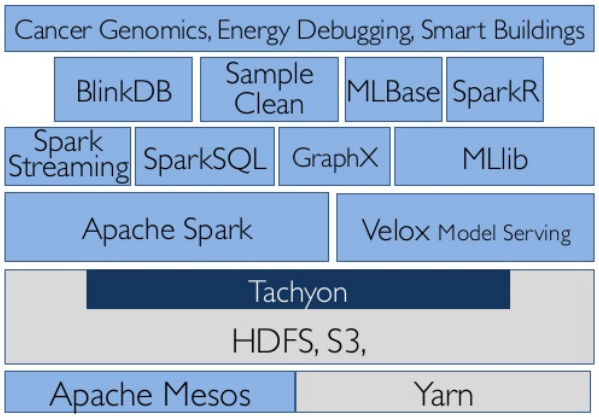
Our Goals

- ***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

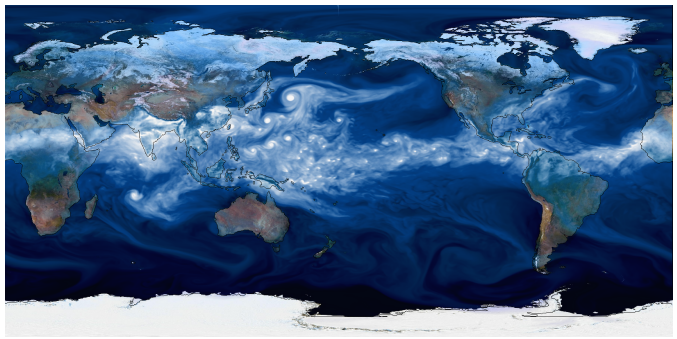
Motivation

- **NERSC**: Spark for data-centric workloads and scientific analytics
- **AMPLab**: characterization of linear algebra in Spark (MLlib, MLMatrix)
- **Cray**: customers demand for Spark; understand performance concerns



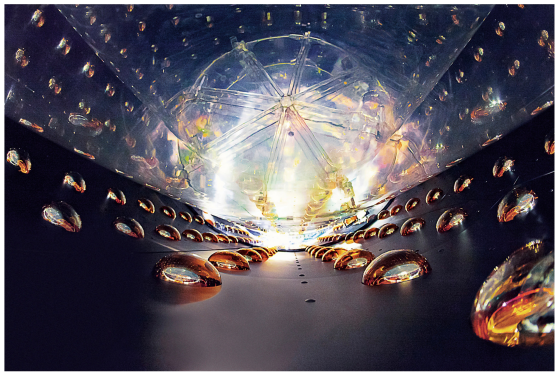


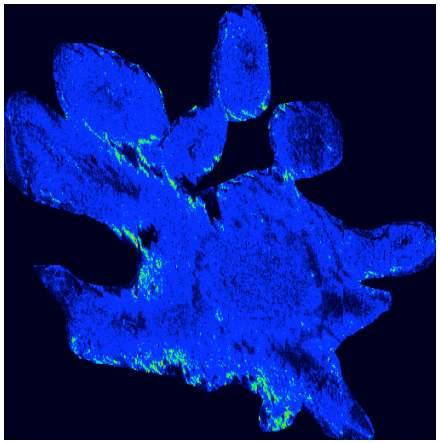
Three Science Drivers



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

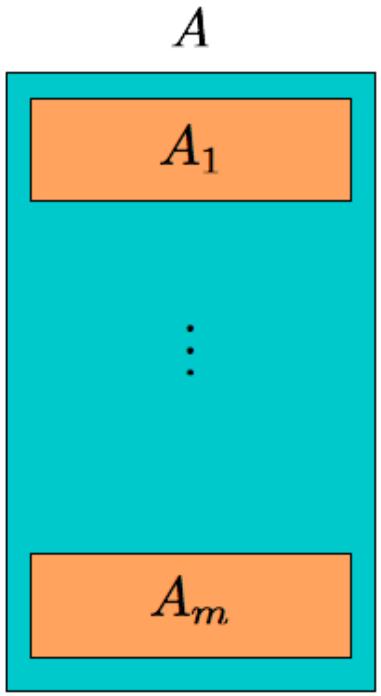
Science Area	Format/Files	Dimensions	Size
MSI	Parquet/2880	$8,258,911 \times 131,048$	1.1TB
Daya Bay	HDF5/1	$1,099,413,914 \times 192$	1.6TB
Ocean	HDF5/1	$6,349,676 \times 46,715$	2.2TB
Atmosphere	HDF5/1	$26,542,080 \times 81,600$	16TB

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

Experiments



1. Compare EC2 and two HPC platforms using CX implementation
2. More detailed analysis of Spark vs C+MPI scaling for PCA and NMF on the two HPC platforms

Some details:

- All datasets are tall and skinny
- The algorithms work with row-partitioned matrices
- Use H5Spark to read dense matrices from HDF5, so MPI and Spark reading from same data source

Platform comparisons

Two Cray HPC machines and EC2, using CX

Randomized CX/CUR Decomposition

- Dimensionality reduction is a ubiquitous tool in science (bio-imaging, neuro-imaging, genetics, chemistry, climatology, …), typical approaches include PCA and NMF which give approximations that rely on non-interpretable combinations of the data points in A

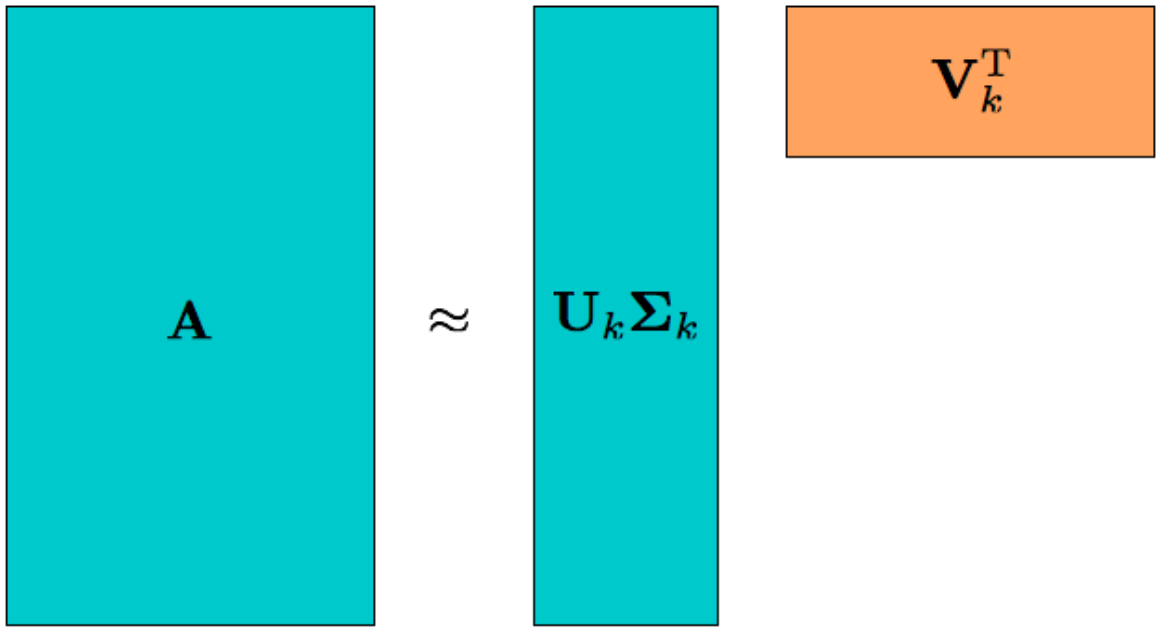
- PCA, NMF lack reifiability. Instead, CX matrix decompositions identify **exemplar** data points (columns of A) that capture the same information as the top singular vectors, and give approximations of the form

$$\mathbf{A} \approx \mathbf{CX}$$

The Randomized CX Decomposition

- To get accuracy comparable to the truncated rank-k SVD, the randomized CX algorithm *randomly* samples O(k) columns with replacement from A according to the *leverage scores*

$$p_i = \frac{\|\mathbf{v}_i\|_2^2}{k}, \qquad \text{where} \qquad \mathbf{V}_k^T = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$$



The Randomized CX Decomposition

- It is expensive to compute the right singular vectors
- Since the algorithm is already randomized, we use a randomized algorithm to quickly approximate them

---

CXDECOMPOSITION

Input: $A \in \mathbb{R}^{m \times n}$, rank parameter $k \leq \text{rank}(A)$, number of power iterations q.

Output: C.

1: Compute an approximation of the top-k right singular vectors of A denoted by $\tilde{V}_k$, using RANDOMIZEDSVD with q power iterations.

2: Let $\ell_i = \sum_{j=1}^{k} \tilde{v}_{ij}^2$, where $\tilde{v}_{ij}^2$ is the (i,j)-th element of $\tilde{V}_k$, for $i = 1, \ldots, n$.

3: Define $p_i = \ell_i / \sum_{j=1}^{d} \ell_j$ for $i = 1, \ldots, n$.

4: Randomly sample c columns from A in i.i.d. trials, using the importance sampling distribution $\{p_i\}_{i=1}^{n}$.

The Randomized SVD algorithm

The matrix analog of the power method:

$$\mathbf{x}_{t+1} = \frac{\mathbf{A}^T\mathbf{A}\mathbf{x}_t}{\|\mathbf{A}^T\mathbf{A}\mathbf{x}_t\|_2} \rightarrow \mathbf{v}_1$$

$$\mathbf{Q}_{t+1}, \_ = \mathrm{QR}(\mathbf{A}^T\mathbf{A}\mathbf{Q}_t) \rightarrow \mathbf{V}_k$$

RANDOMIZEDSVD Algorithm

Input: $A \in \mathbb{R}^{m \times n}$, number of power iterations $q \geq 1$, target rank $k > 0$, slack $p \geq 0$, and let $\ell = k + p$.

Output: $U\Sigma V^T \approx A_k$.

1: Initialize $B \in \mathbb{R}^{n \times \ell}$ by sampling $B_{ij} \sim \mathcal{N}(0, 1)$.
2: **for** q times **do**
3: $\quad B \leftarrow A^T AB$
4: $\quad (B, \_) \leftarrow \mathrm{THINQR}(B)$
5: **end for**
6: Let Q be the first k columns of B.
7: Let $M = AQ$.
8: Compute $(U, \Sigma, \tilde{V}^T) = \mathrm{THINSVD}(M)$.
9: Let $V = Q\tilde{V}$.

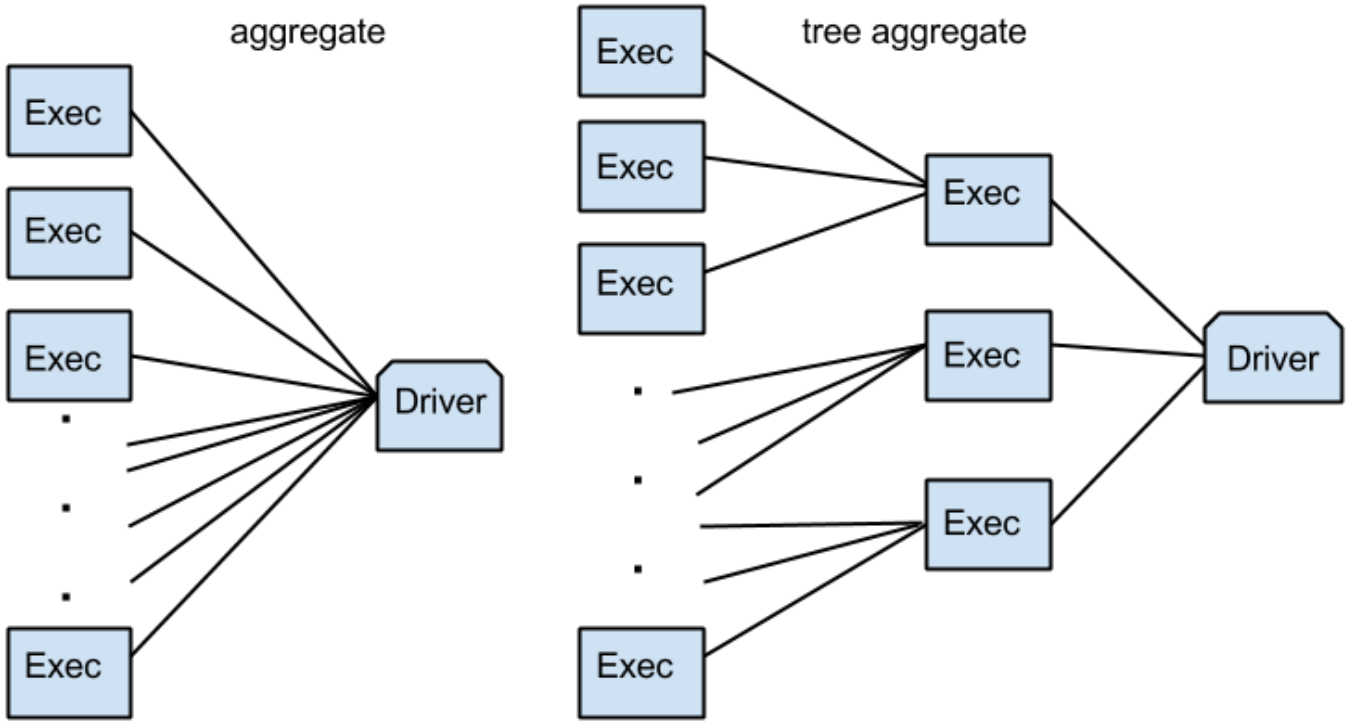
requires only matrix-matrix multiplies against A^TA

assumes B fits on one machine

Computing the power iterations using Spark

$$(\mathbf{A}^T\mathbf{A})\mathbf{B} = \sum_{i=1}^{m} \mathbf{a}_i(\mathbf{a}_i^T\mathbf{B})$$

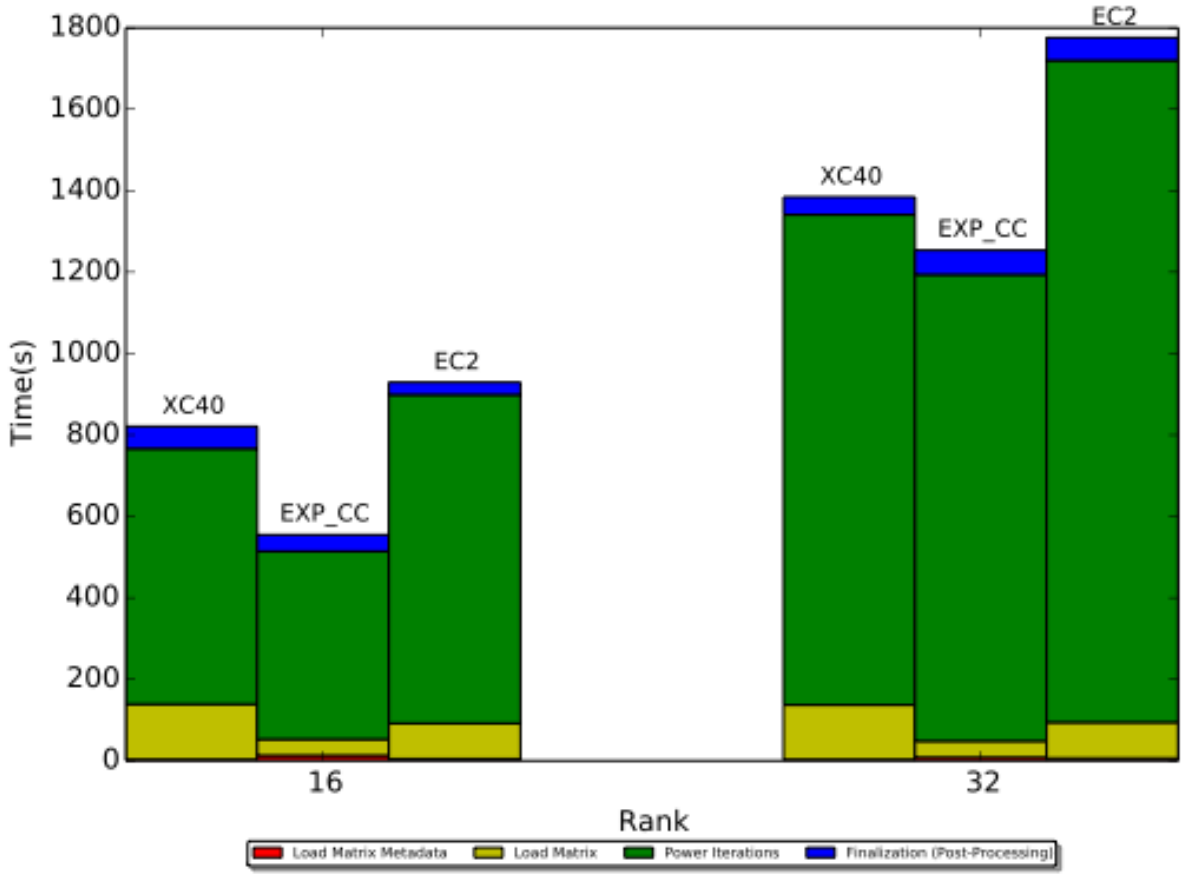
is computed using a treeAggregate operation over the RDD



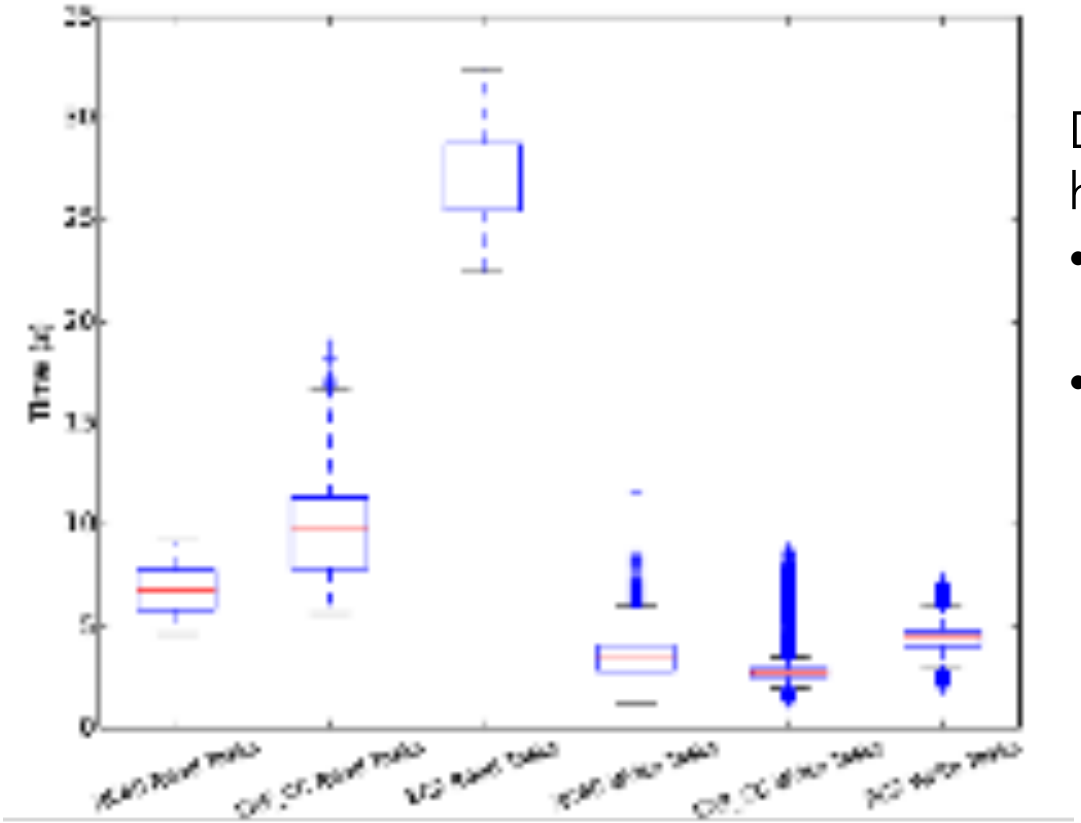
[src: https://databricks.com/blog/2014/09/22/spark-1-1-mllib-performance-improvements.html]

CX run-times: 1.1Tb

Platform	Total Cores	Core Frequency	Interconnect	DRAM	SSDs
Amazon EC2 `r3.8xlarge`	960 (32 per-node)	2.5 GHz	10 Gigabit Ethernet	244 GiB	2 x 320 GB
Cray XC40	960 (32 per-node)	2.3 GHz	Cray Aries [20], [21]	252 GiB	None
Experimental Cray cluster	960 (24 per-node)	2.5 GHz	Cray Aries [20], [21]	126 GiB	1 x 800 GB



Timing breakdowns



Differences in write timings have more impact:

- 4800 write tasks per iteration
- 68 read tasks per iteration

Platform	Total Runtime	Load Time	Time Per Iteration	Average Local Task	Average Aggregation Task	Average Network Wait
Amazon EC2 `r3.8xlarge`	24.0 min	1.53 min	2.69 min	4.4 sec	27.1 sec	21.7 sec
Cray XC40	23.1 min	2.32 min	2.09 min	3.5 sec	6.8 sec	1.1 sec
Experimental Cray cluster	15.2 min	0.88 min	1.54 min	2.8 sec	9.9 sec	2.7 sec

Observations

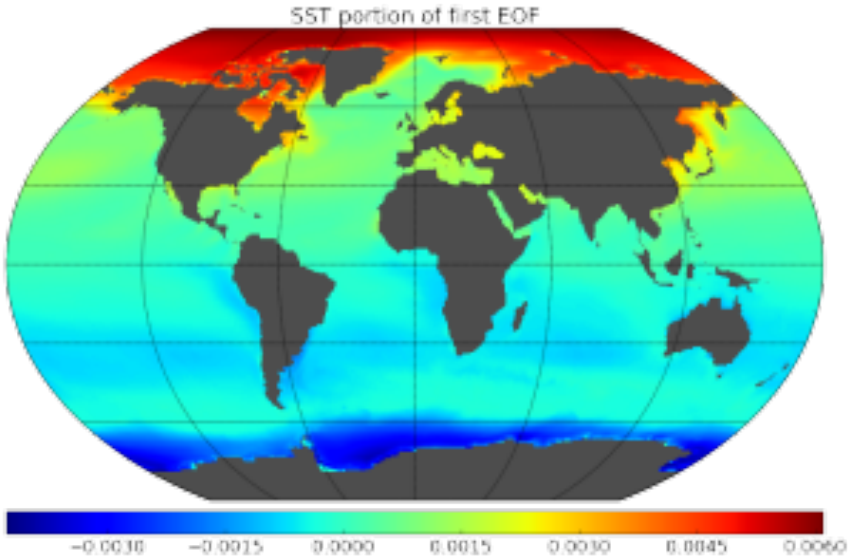
- EXP_CC outperforms EC2 and XC40 because of local storage and faster interconnect
- On HPC platforms, can focus on modifying Spark to mitigate drawbacks of the global filesystem:

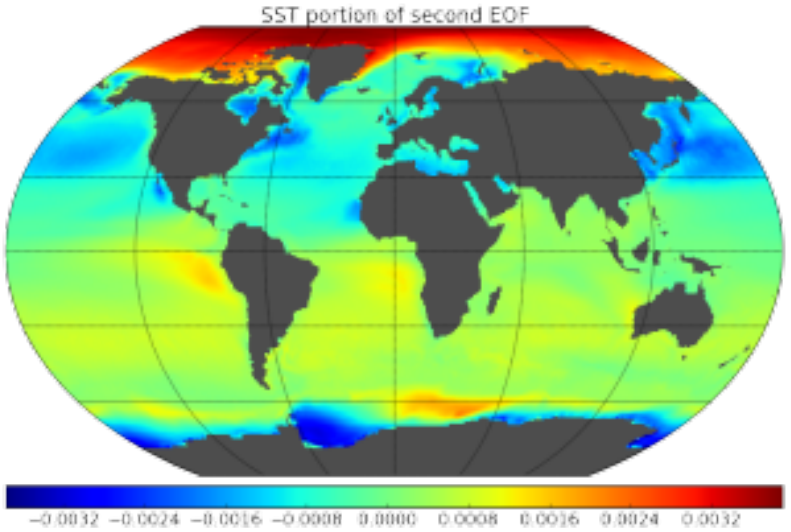
1. **clean scratch more often** to help fit scratch entirely in RAM, no need to spill to Lustre
2. allow user to **specify order to fill scratch directories** (RAM disk, *then* Lustre)
3. **exploit fact that scratch on shared filesystem is global**, to avoid wasted communication

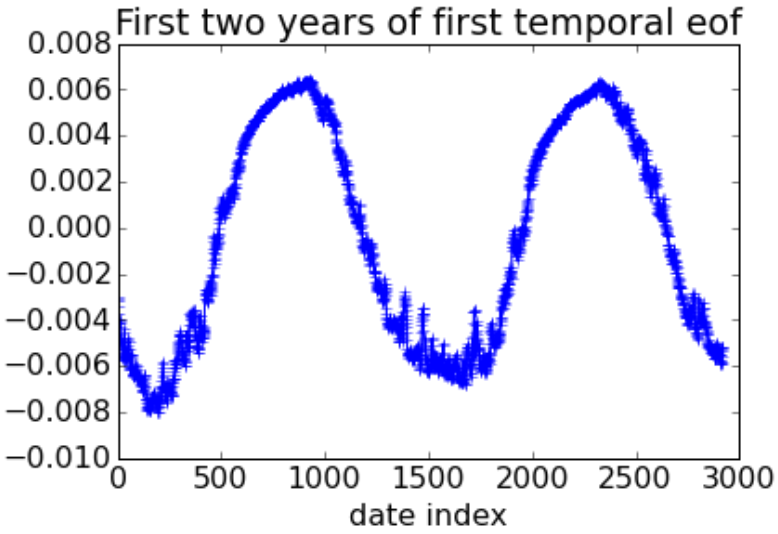
Spark vs MPI

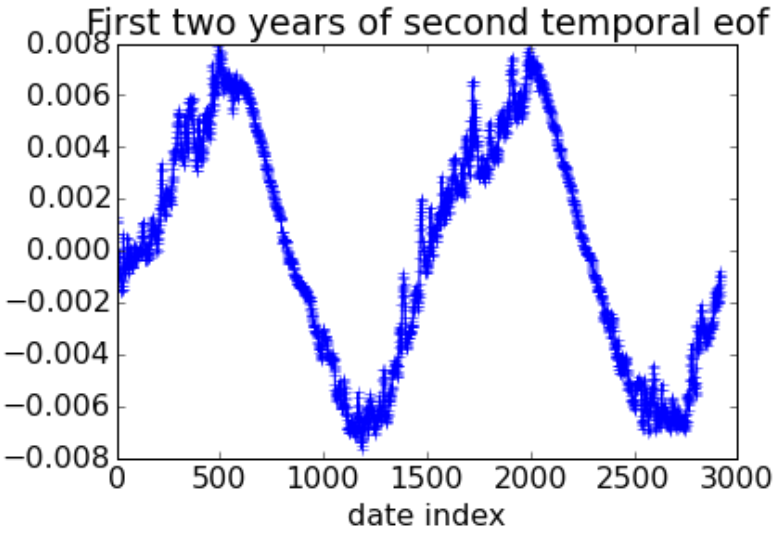
PCA and NMF, on NERSC's Cori supercomputer

CFSR Ocean Temperature Dataset (II)

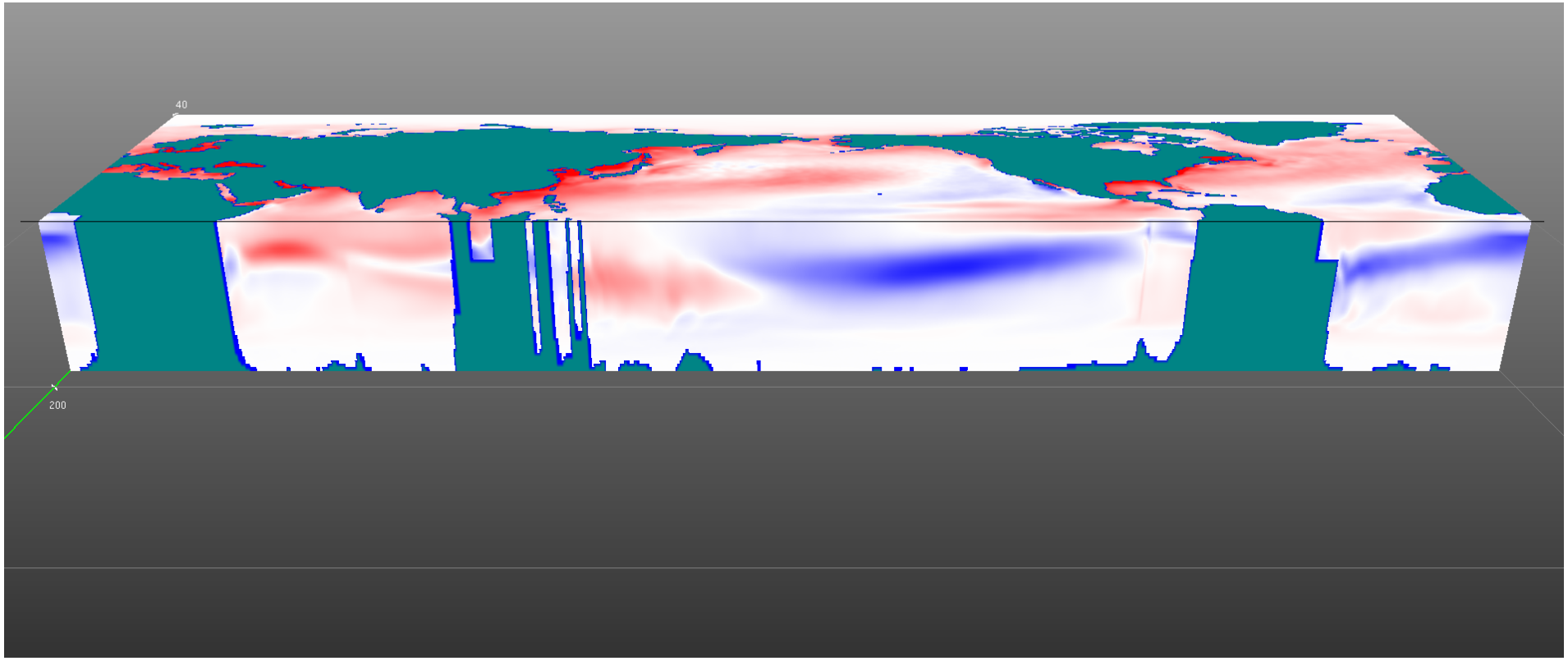








Climate Science Results on Ocean (CFSRO) dataset



- First principal component of temperature field at 180 degree latitude.
- Clear that there is a significant vertical component to the PCs which are lost when you do the traditional surface-only analyses

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.

Computing the truncated PCA

Often (for dimensionality reduction, physical interpretation, etc.), the rank-k truncated PCA (SVD) is desired. It is defined as

$$\mathbf{A}_k = \operatorname{argmin}_{\operatorname{rank}(\mathbf{B})=k} \|\mathbf{A} - \mathbf{B}\|_F^2$$

The two steps in computing the truncated PCA of A are:

use Lanczos: requires only matrix vector multiplies

1. Compute the truncated EVD of A^TA to get V_k
2. Compute the SVD of AV_k to get Σ_k and V_k

assume this is small enough that the SVD can be computed locally

Computing the Lanczos iterations using Spark

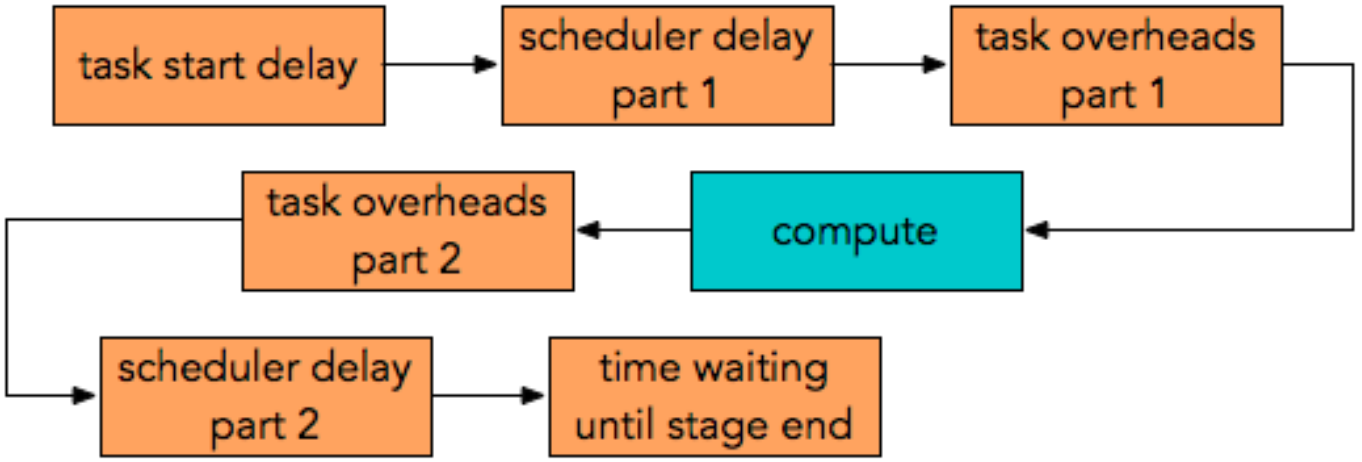
We call the `spark.mllib.linalg.EigenvalueDecomposition` interface to the ARPACK implementation of the Lanczos method

This requires a function which computes a matrix-product against A^TA

If $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_2^T \end{bmatrix}$ then the product can be computed as

$$(\mathbf{A}^T\mathbf{A})\mathbf{x} = \sum_{i=1}^{m} \mathbf{a}_i(\mathbf{a}_i^T\mathbf{x})$$

Spark Overheads: the view of one task



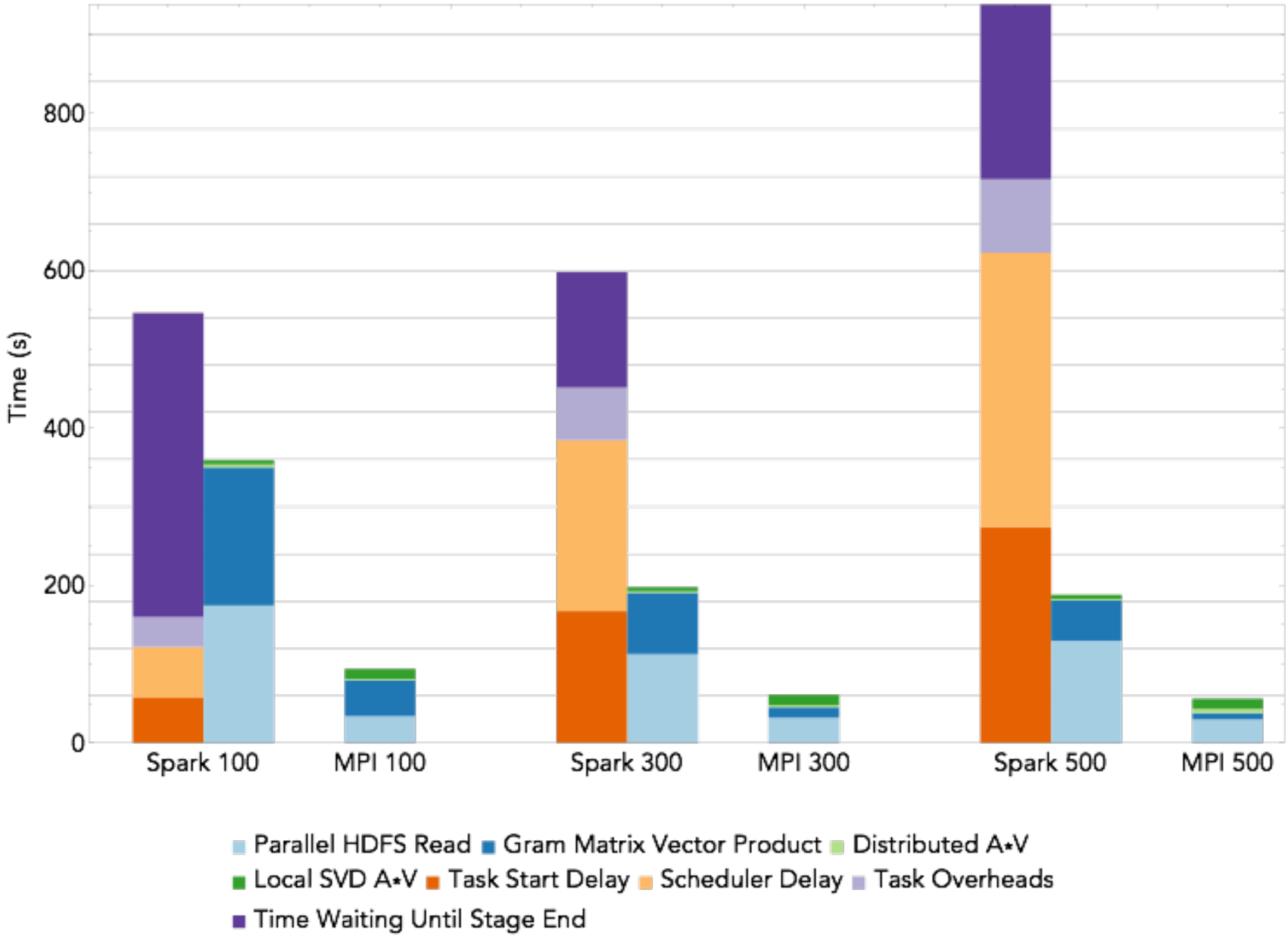
task start delay = (time between stage start and when driver sends task to executor)

scheduler delay = (time between task being sent and time starts deserializing)+ (time between task result serialization and driver receiving task's completion message)

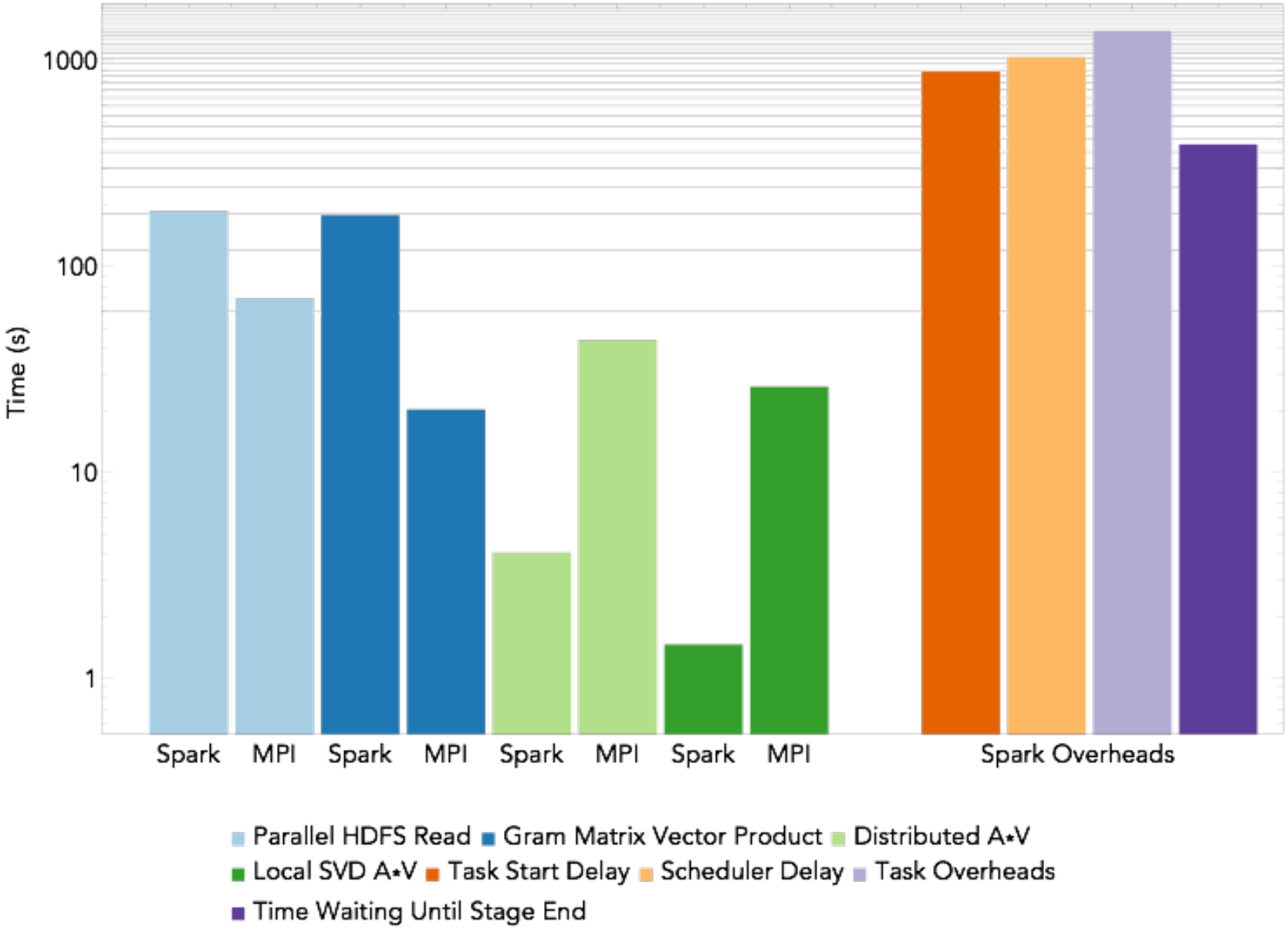
task overhead time = (fetch wait time) + (executor deserialize time) + (result serialization time) + (shuffle write time)

time waiting until stage end = (time waiting for final task in stage to end)

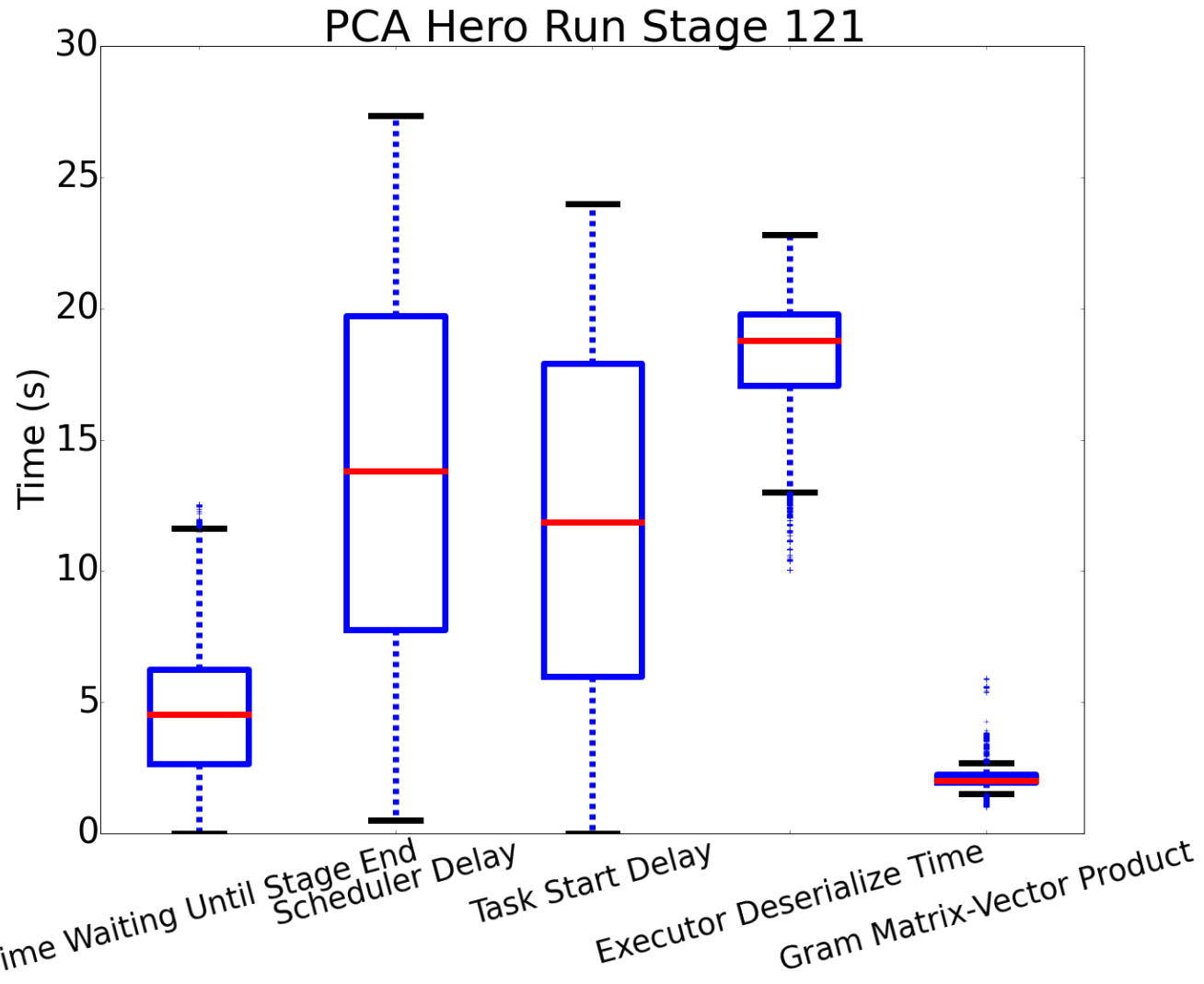
PCA Run Times: rank 20 PCA of 2.2TB Climate



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



Spark PCA Overheads: 16 TB Climate,1522 nodes



Nonnegative Matrix Factorization

Useful when the observations are positive, and assumed to be positive combinations of basis vectors (e.g., medical imaging modalities, hyperspectral imaging)

$$(\mathbf{W}, \mathbf{H}) = \operatorname{argmin}_{\substack{\mathbf{W}\geq 0 \\ \mathbf{H}\geq 0}} \|\mathbf{A} - \mathbf{WH}\|_F$$

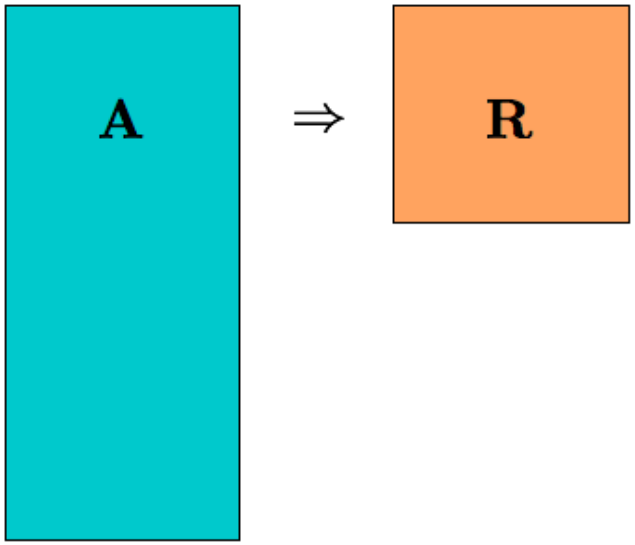
In general, NMF factorizations are non-unique and NP-hard to compute for a fixed rank.

We use the one-pass approach of Benson et al. 2014

Nearly-Separable NMF

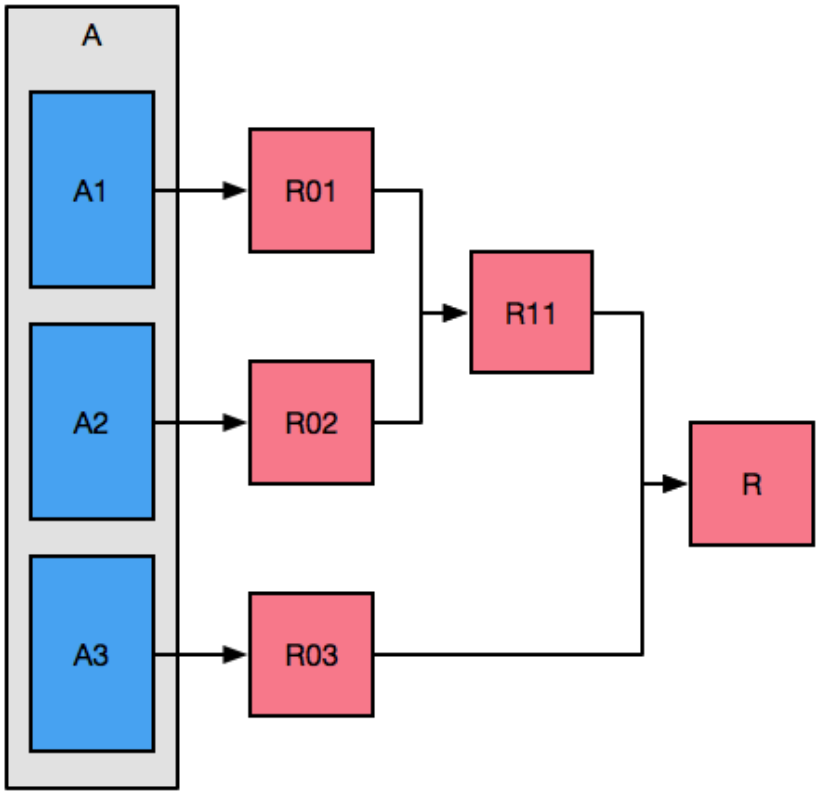
Assumption: *some k-subset of the columns of A comprise a good W*

Key observation of Benson et al. : finding those columns of A can be done on the R factor from the QR decomposition of A



So the problem reduces to a distributed QR on a tall matrix A, then a local NMF on a much smaller matrix

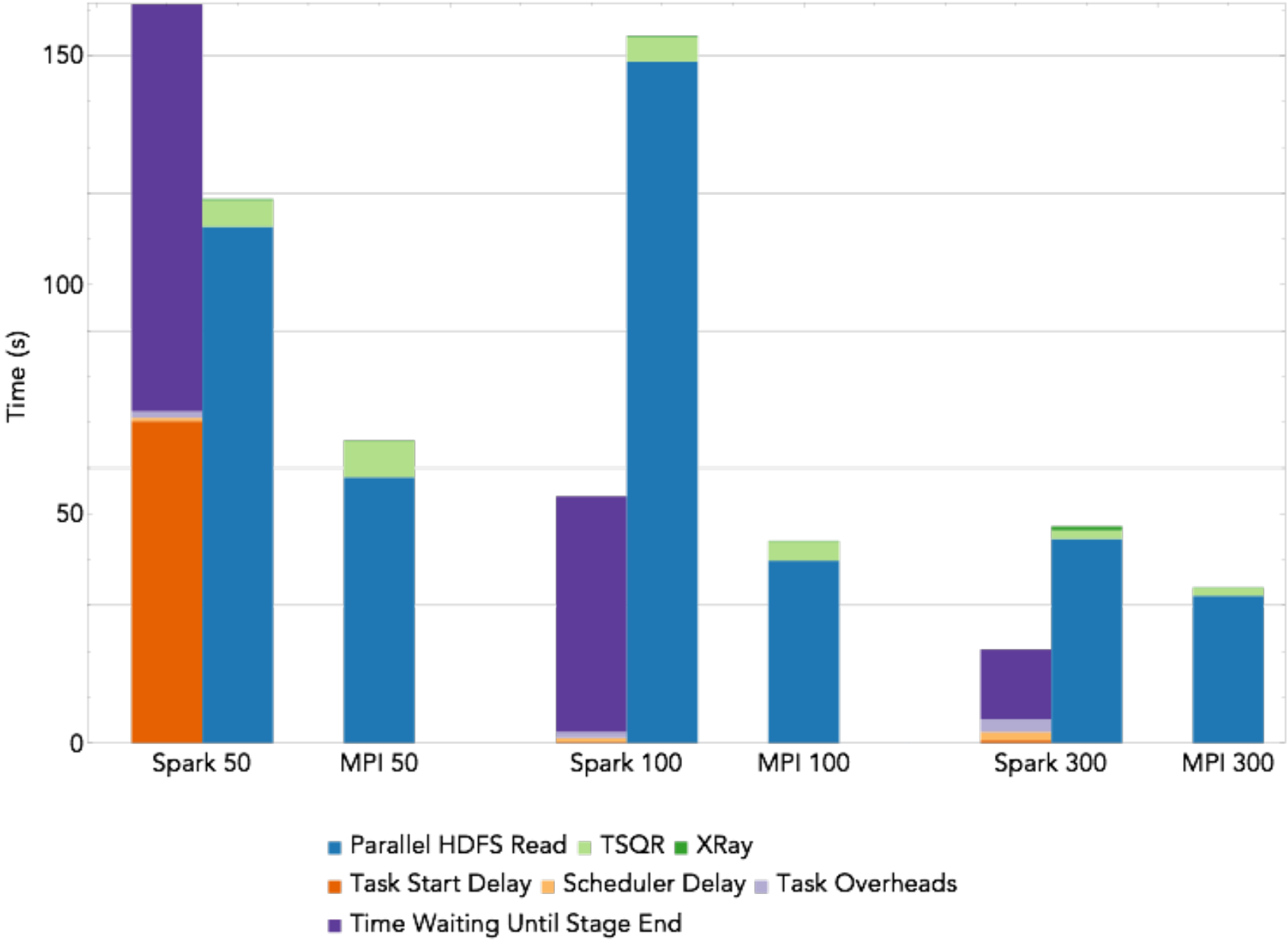
Tall-Skinny QR (TSQR)



When A is tall and skinny, you can efficiently compute R:

- uses a tree reduce
- requires only one pass over A

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



MPI vs Spark: Lessons Learned

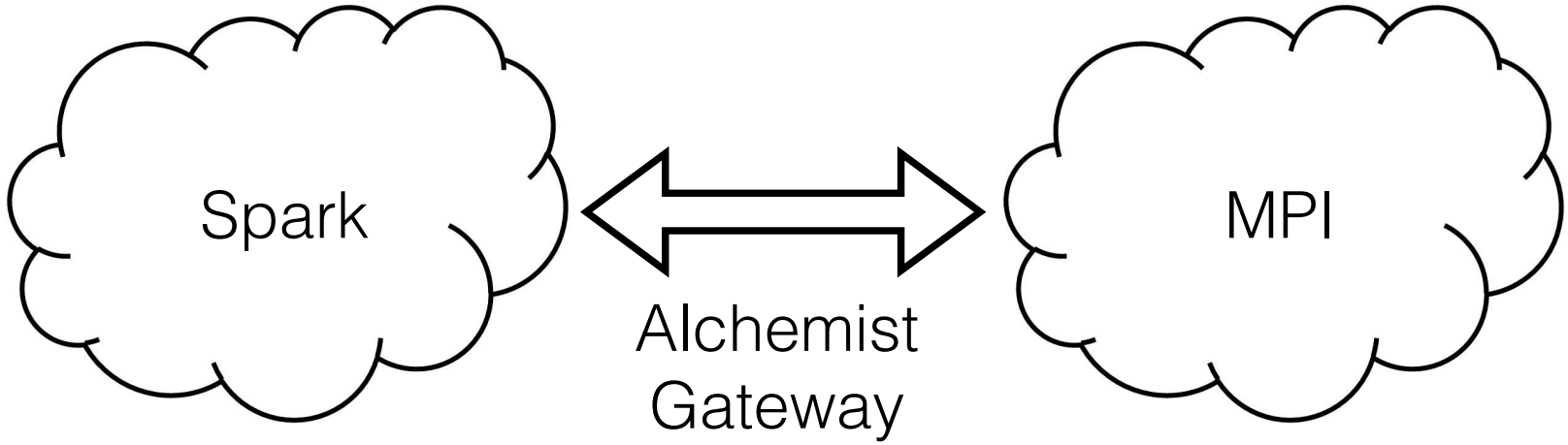
- With favorable data (tall and skinny) and well-adapted algorithms, ***Spark LA is 4x-26x slower than MPI when IO is included***
- ***Spark overheads are orders of magnitude higher than the computations*** in PCA (time till stage end, scheduler delay, task start delay, executor deserialize time).
- ***H5Spark performance is inconsistent*** this needs more work
- The large gaps mean it is worthwhile to ***investigate efficiently interfacing MPI-based codes with Spark***

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)

	File I/O	RAM	Network Usage	Computational Efficiency
HDFS	writes to disk	2x RAM	manual shuffling	yes
Apache Ignite	none	2-3x RAM	intelligent	restricted partitioning
Alluxio	none	2-3x RAM	intelligent	restricted partitioning
Alchemist	none	2x RAM	intelligent	yes

Alchemist Architecture



Spark:

1) Send metadata for input/output matrices to the Alchemist gateway
2) Sends the matrix to the Alchemist gateway using RDD.pipe()
3) Waits on a matrix from the Alchemist gateway using RDD.pipe()

Alchemist:

1) repartitions the matrix for MPI
2) executes the MPI codes
3) repartitions the output and returns to Spark

Each call to Alchemist will be single stage in the execution of Spark jobs

What Alchemist Will Enable

- Use MPI NLA/ML Codes from Spark: libSkylark, MaTeX, etc.

```
…
val xMat = alcMat(xRDD)
val yMat = alcMat(yRDD)

// Elemental NLA
val (u,s,v) =
alchemist.SVD(xMat,k).toIndexRowMatrices()

// libSkylark ML
val (rffweights,alpha)  =
alchemist.RFFRidgeRegression(xMat,yMat,lambda,D)

// MaTeX ML
val clusterIndicators = alchemist.kMeans(xMat,k)
…
```

It will be easy to write lightweight wrappers around APIs of existing MPI codes.

Multi-platform evaluation of randomized CX/CUR (IPDPS 2016 Workshop)

Poster presentations at climate science venues

Technical Report on Spark performance for Terabyte-scale Matrix Decompositions (submitted): *https://arxiv.org/abs/1607.01335*

Attendant MPI and Spark codes:
https://github.com/alexgittens/SparkAndMPIFactorizations

End-to-end 3D Climate EOF codes:
https://github.com/alexgittens/climate-EOF-suite

CUG 2016 Paper on H5Spark:
https://github.com/valiantljk/h5spark/files/261834/h5spark-cug16-final.pdf

For more info, contact me or Alex Gittens: *gittens@icsi.berkeley.edu*

Conclusion

Sophisticated statistical data analysis involves strong control over linear algebra.

Most workflows/applications currently do not demand much of the linear algebra.

Low-rank matrix algorithms for interpretable scientific analytics on scores of terabytes of data!

Complex data workflows invoke linear algebra as one step (of a multi-stage pipeline), and data/communication optimization might be needed across function calls.

What is the “right” way to do linear algebra for large-scale data analysis?