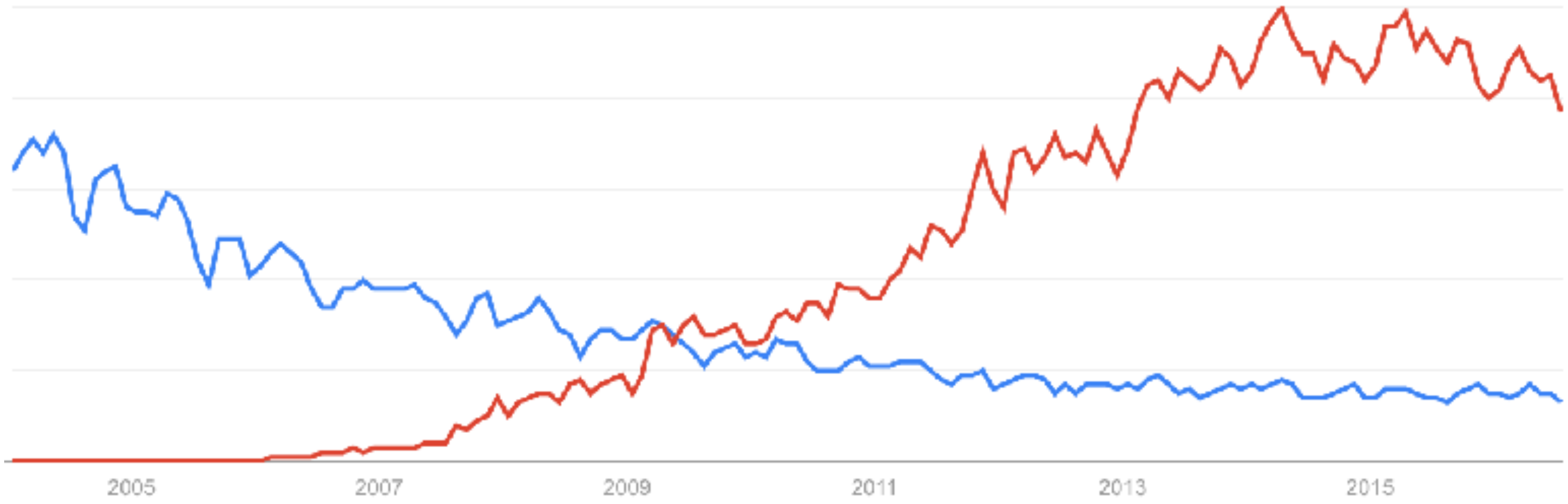
Terabyte-scale Numerical Linear Algebra in Spark: Current Performance, and Next Steps

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

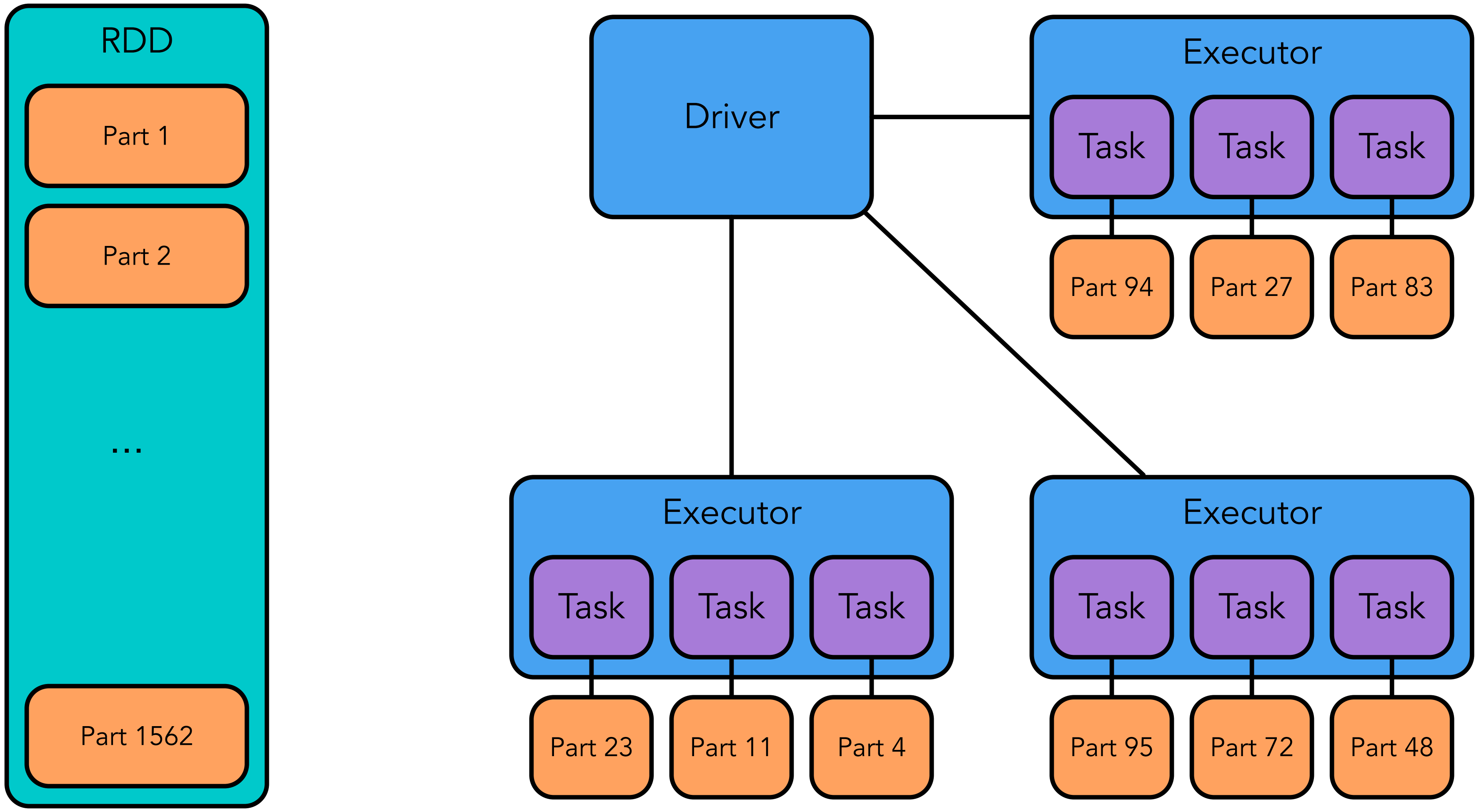
SKA-GridPP workshop talk
November 2, 2016

Why Spark?



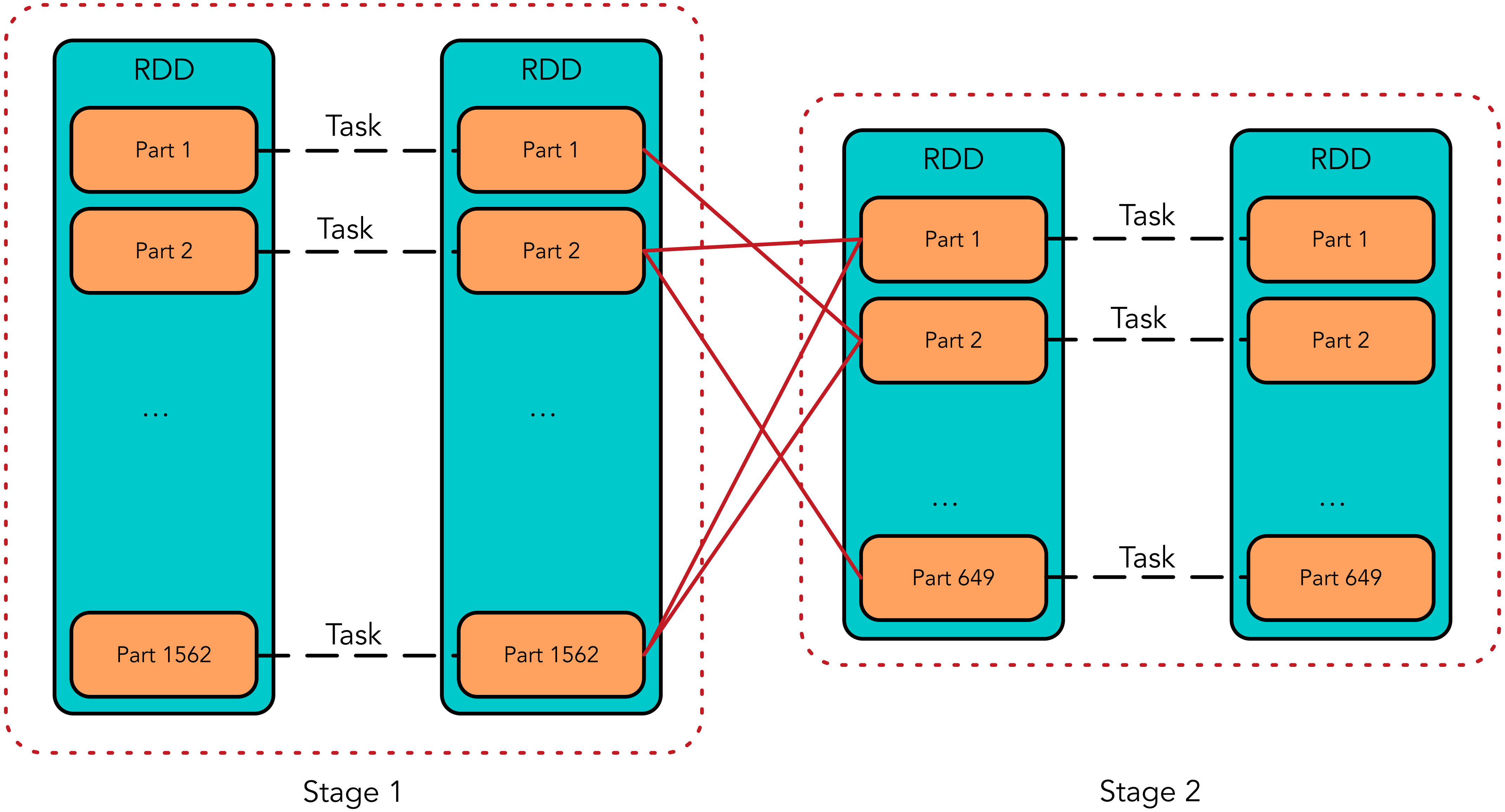
Google trends popularity: MPI vs Hadoop

Spark Architecture



- Data parallel programming model
- Resilient distributed datasets (RDDs); optionally cached in memory
- Driver forms DAG, schedules tasks on executors

Spark Communication

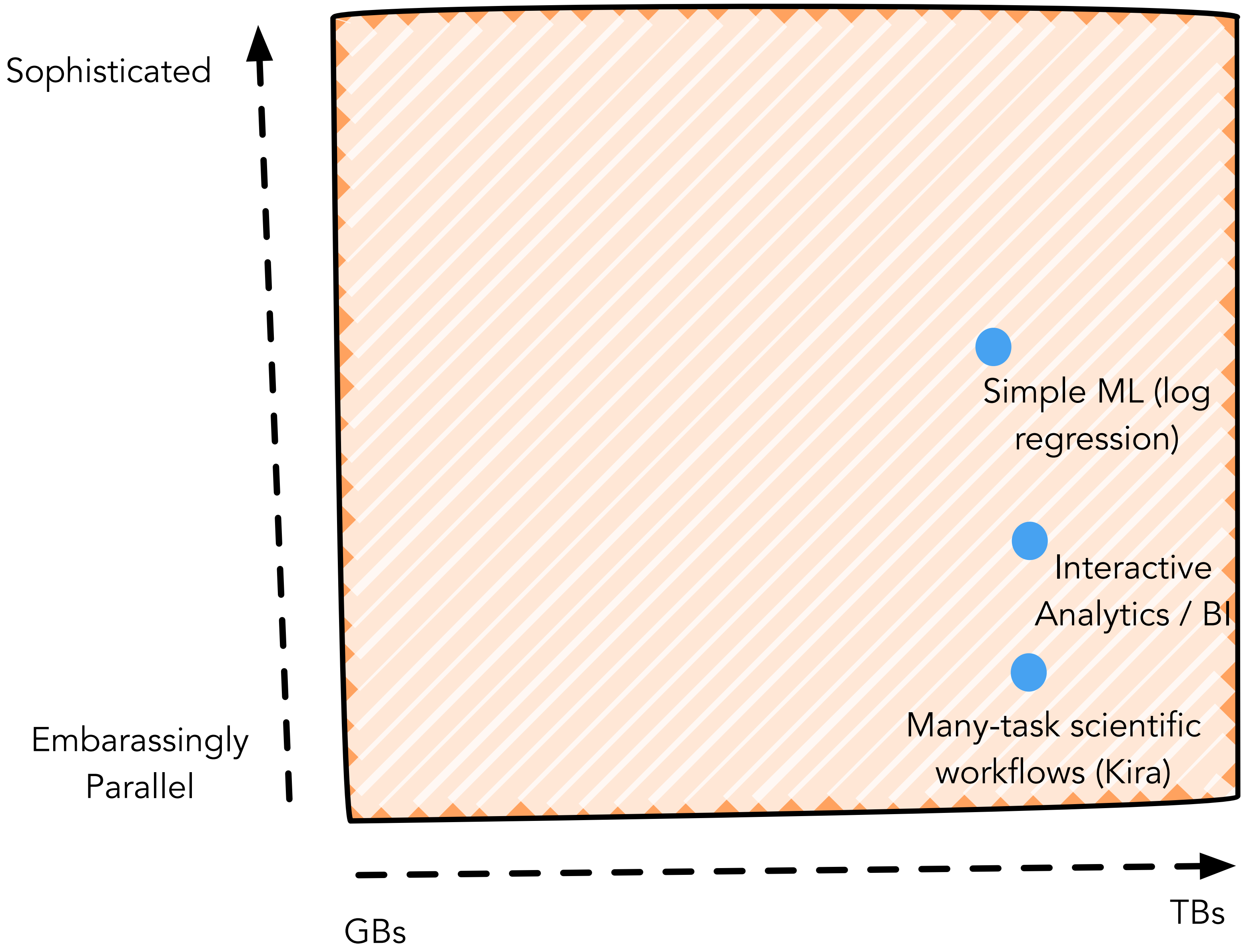


Bulk Synchronous Programming Model:

- Each overall job (DAG) broken into stages
- Stages broken into parallel, independent tasks
- Communication happens only between stages

Spark Use Cases

Performance depends on **problem scale** and **level of parallelism**


Sophisticated
Embarassingly Parallel
Simple ML (log regression)
Interactive Analytics / BI
Many-task scientific workflows (Kira)
GBs
TBs

What about large-scale linear algebra?

Why do linear algebra in Spark?

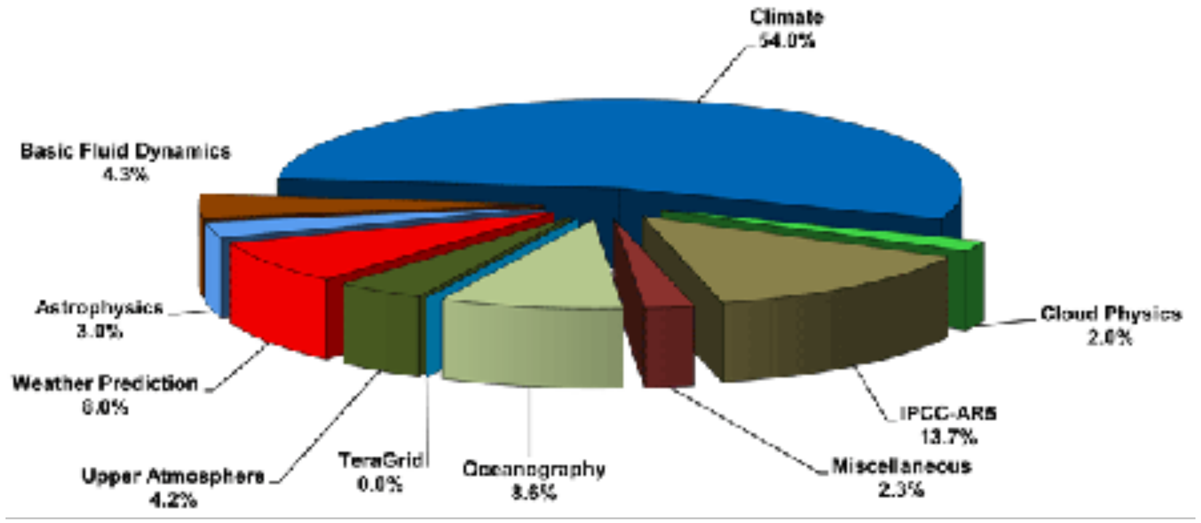
Con: Classical MPI-based linear algebra implementations will be faster and more efficient

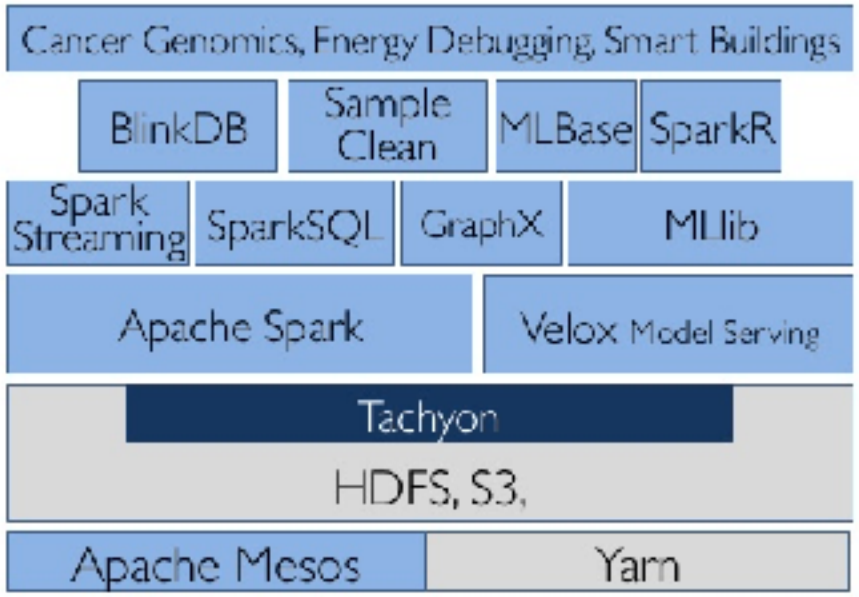
Pros:

- Faster development, easier reuse
- One abstract uniform interface
- An entire ecosystem that can be used before and after the NLA computations
- Spark can take advantage of available local linear algebra codes
- Automatic fault-tolerance, out-of-core support

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




Cancer Genomics, Energy Debugging, Smart Buildings

BlinkDB | Sample Clean | MLBase | SparkR

Spark Streaming | SparkSQL | GraphX | MLlib

Apache Spark | Velox Model Serving

Tachyon

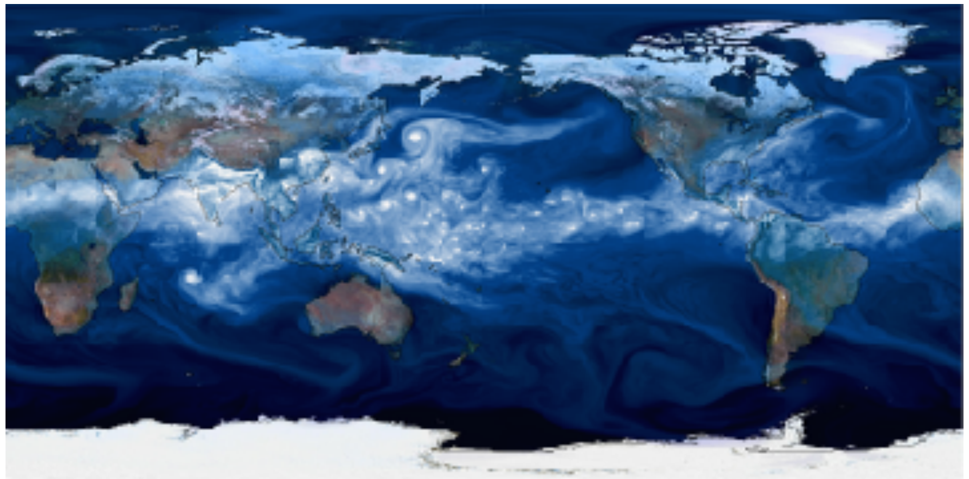
HDFS, S3,

Apache Mesos | Yarn

Our Goals

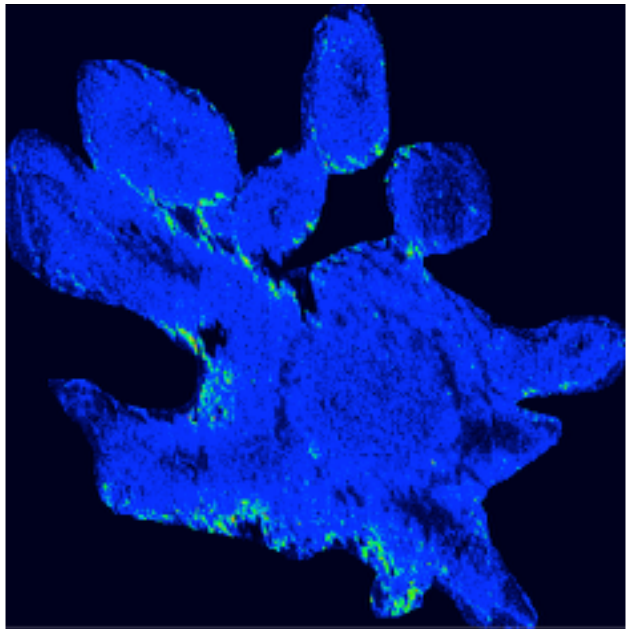
- Apply low-rank matrix factorization methods ***to TB-scale scientific datasets*** in Spark
- Understand ***Spark performance on commodity clusters vs HPC platforms***
- ***Quantify the gaps*** between C+MPI and Spark implementations
- ***Investigate the scalability*** of current Spark-based linear algebra on HPC platforms

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

CFSR Datasets

- Consists of multiyear (1979—2010) global gridded representations of atmospheric and oceanic variables, generated using constant data assimilation and interpolation using a fixed model

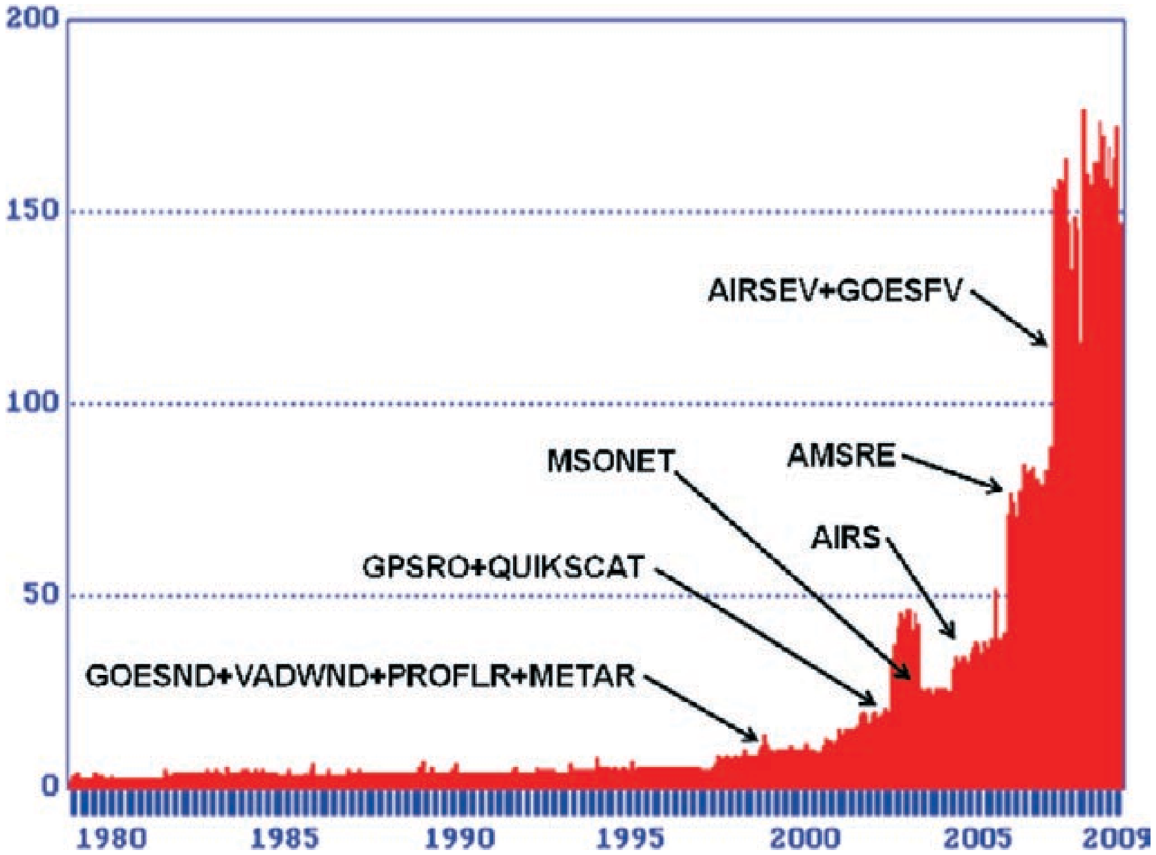
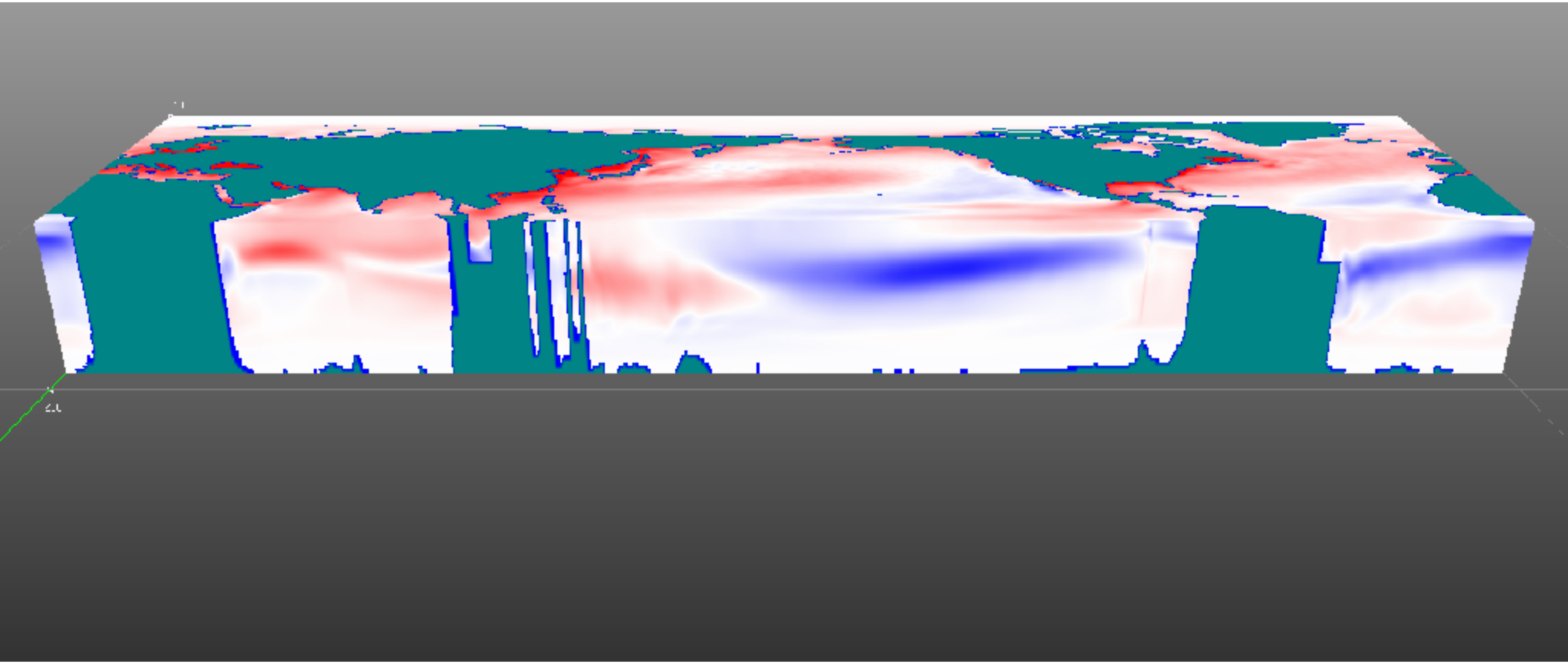


FIG. 2. Diagram illustrating CFSR data dump volumes, 1978–2009 (GB month^{-1}).

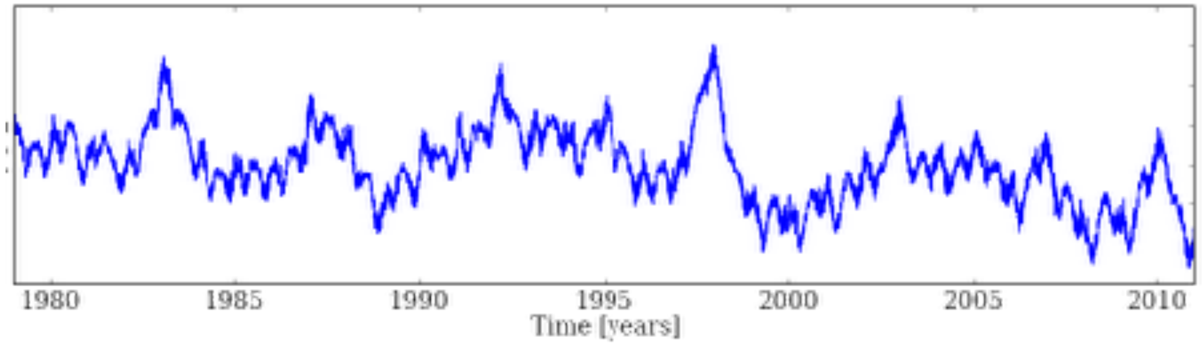
[src: http://cfs.ncep.noaa.gov/cfsr/docs/]

Climate Science : 3D EOFs

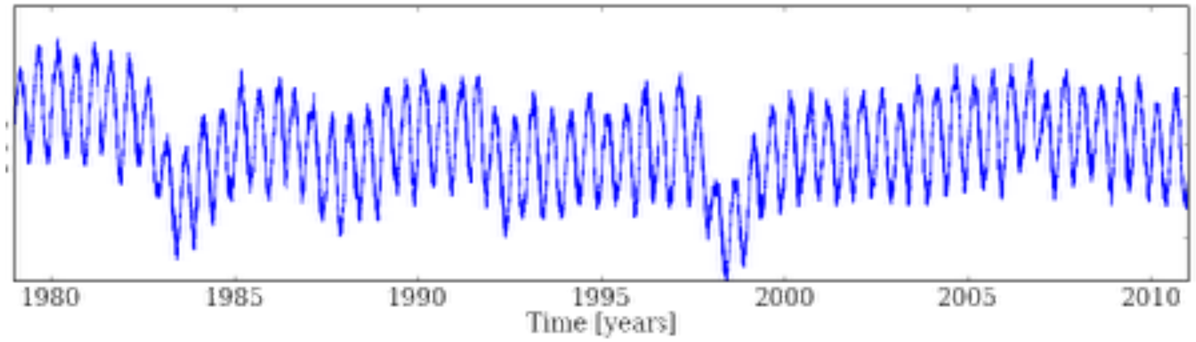


Climate Science : Time Series

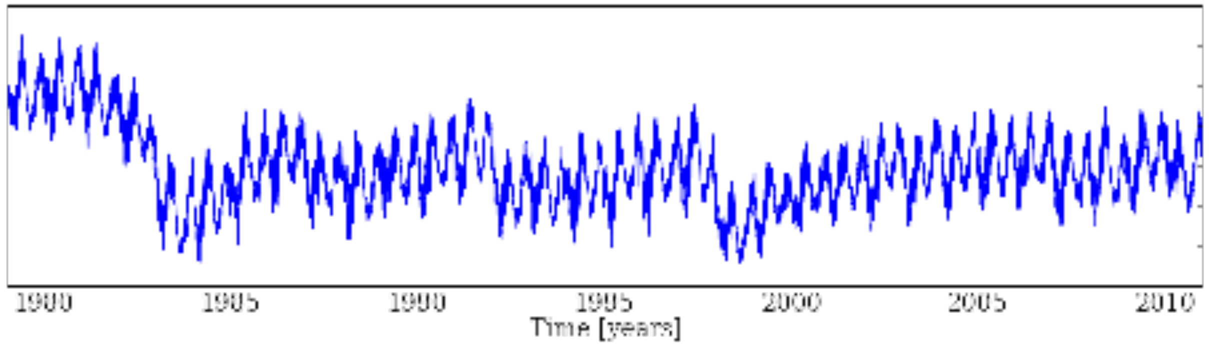
mode 3



mode 6

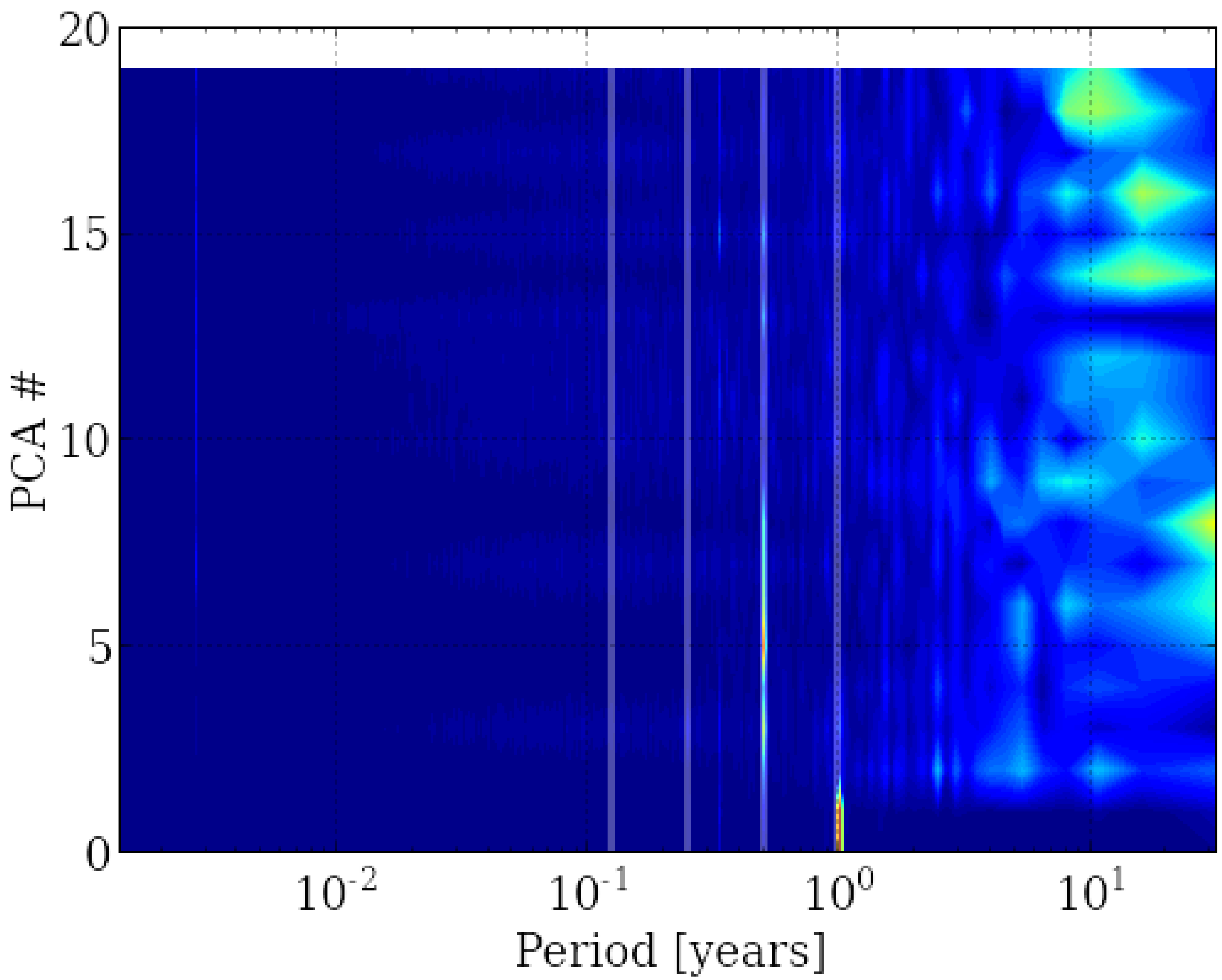


mode 7



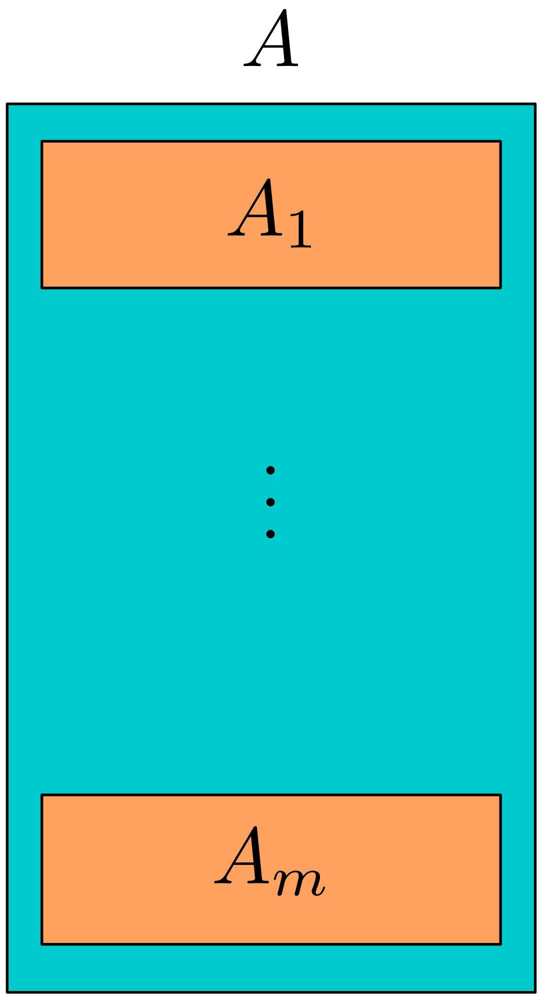
The time series reflect chaotic but periodic nature of oceanic variability and reflect the abrupt changes due to the 1983 ENSO (El Niño), and the record-breaking ENSO of 1997-98.

Climate Science: Power Spectra



The first two modes fully capture the annual cycle, while the higher modes contain low frequency content. The interplay of frequencies in the intermediate modes is currently under investigation. The vertical stripes seem to be artifacts of the reanalysis.

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

The Randomized CX 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

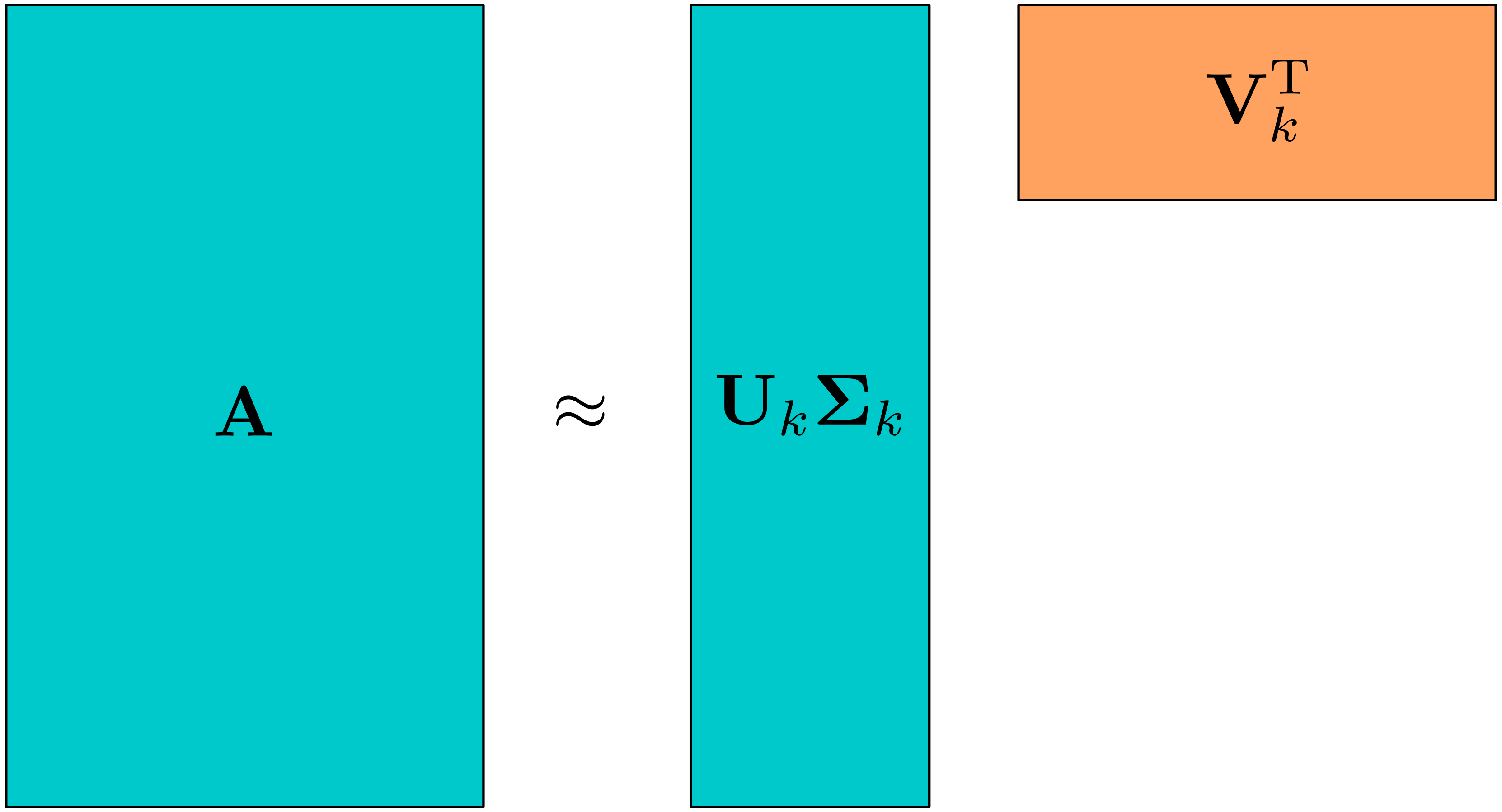
- PCA, NMF lack reifiability. Instead, CX matrix decompositions identify **exemplar** data points (columns of A) that capture the same information as the top singular vectors, and give approximations of the form

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

The Randomized CX Decomposition

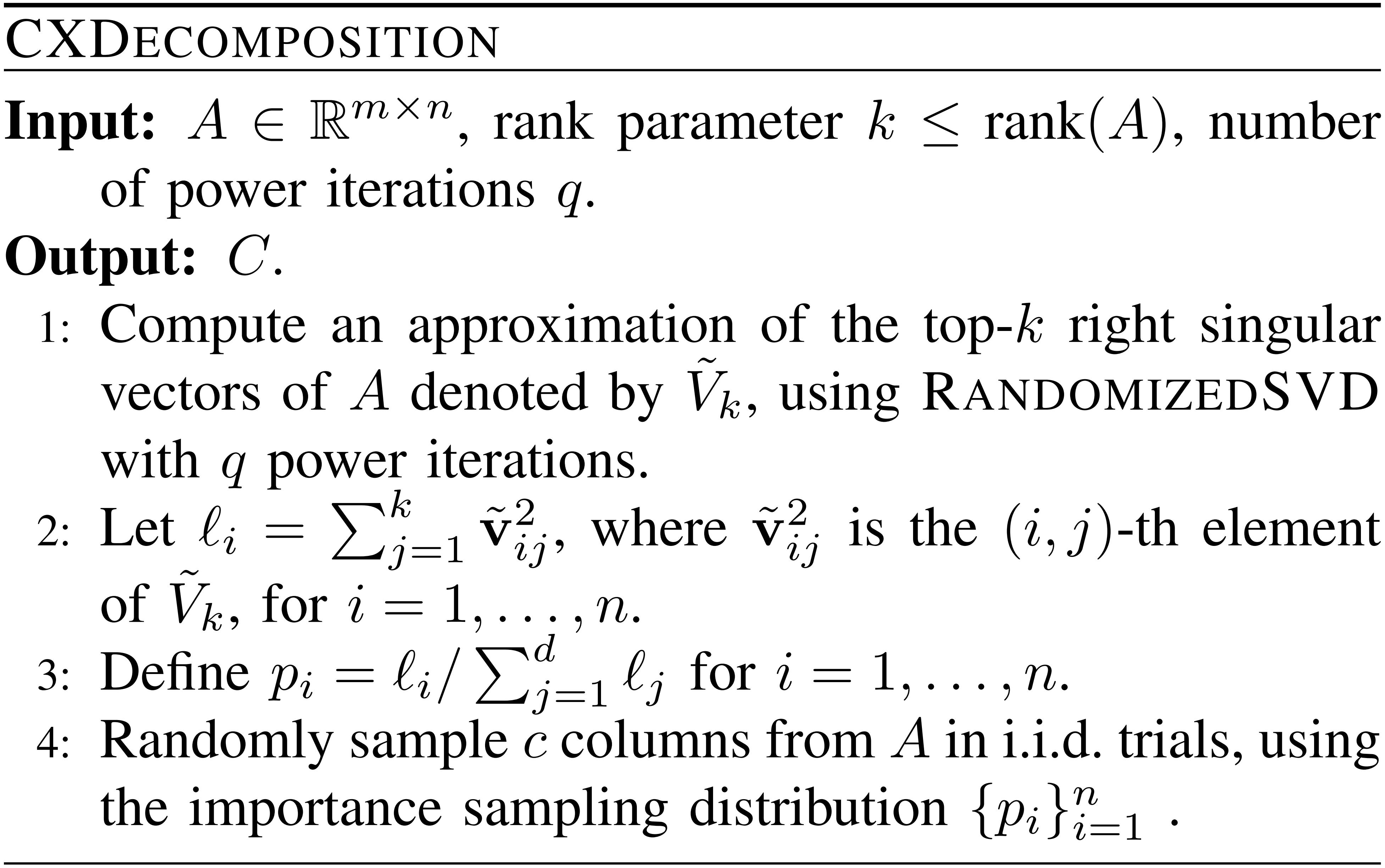
- To get accuracy comparable to the truncated rank-k SVD, the randomized CX algorithm *randomly* samples O(k) columns with replacement from A according to the *leverage scores*

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



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 $\ell_i = \sum_{j=1}^{k} \tilde{\mathbf{v}}_{ij}^2$, where $\tilde{\mathbf{v}}_{ij}^2$ 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:

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

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

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: $U\Sigma V^T \approx A_k$.
Initialize $B \in \mathbb{R}^{n \times \ell}$ by sampling $B_{ij} \sim \mathcal{N}(0,1)$.
for $q$ times do
    $B \leftarrow A^T A B$
    $(B, \_) \leftarrow \textsc{ThinQR}(B)$
end for
Let $Q$ be the first $k$ columns of $B$.
Let $M = AQ$.
Compute $(U, \Sigma, \tilde{V}^T) = \textsc{ThinSVD}(M)$.
Let $V = Q\tilde{V}$.
```

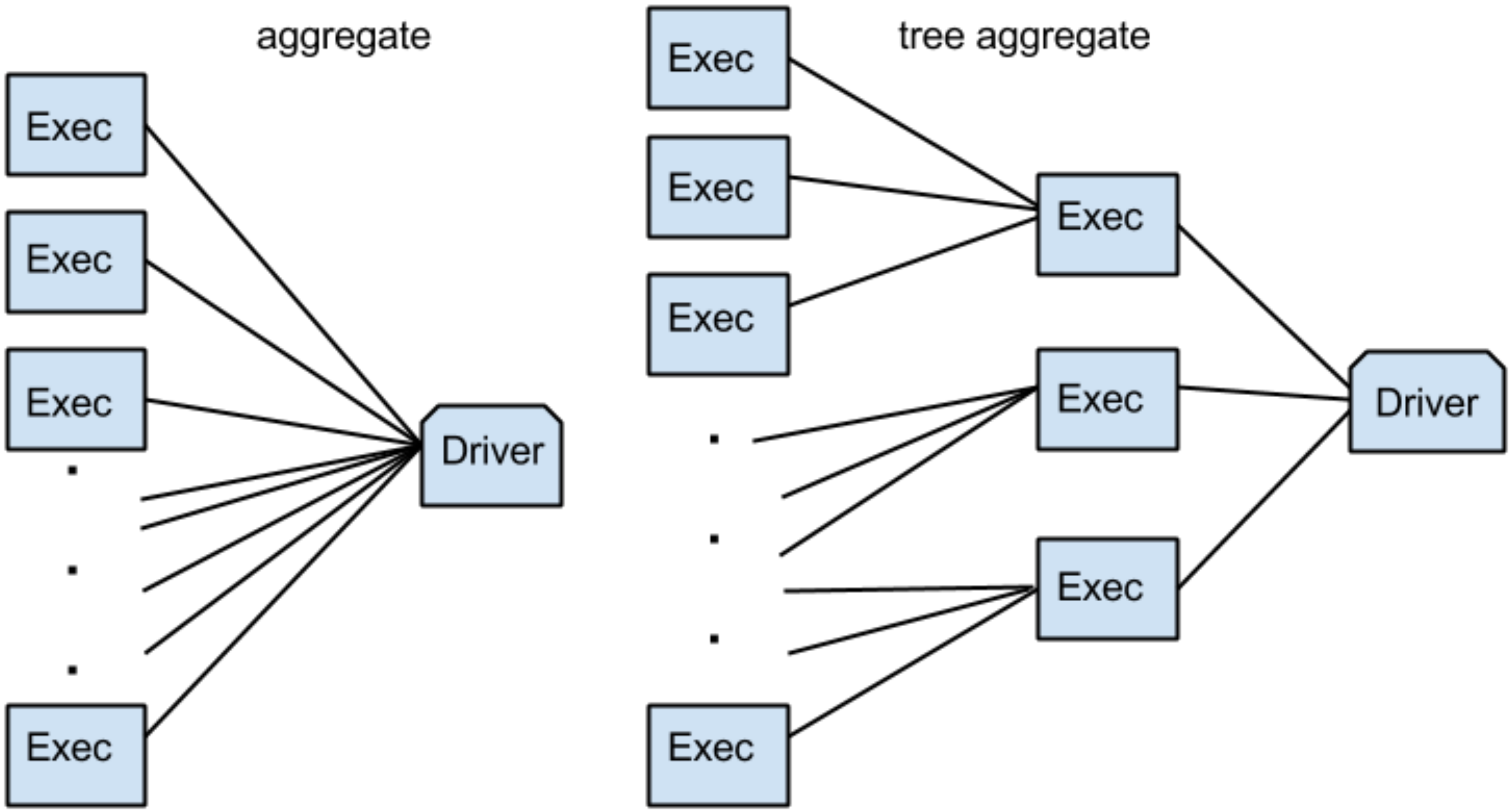
requires only matrix-matrix multiplies against A^TA

assumes B fits on one machine

Computing the power iterations using Spark

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

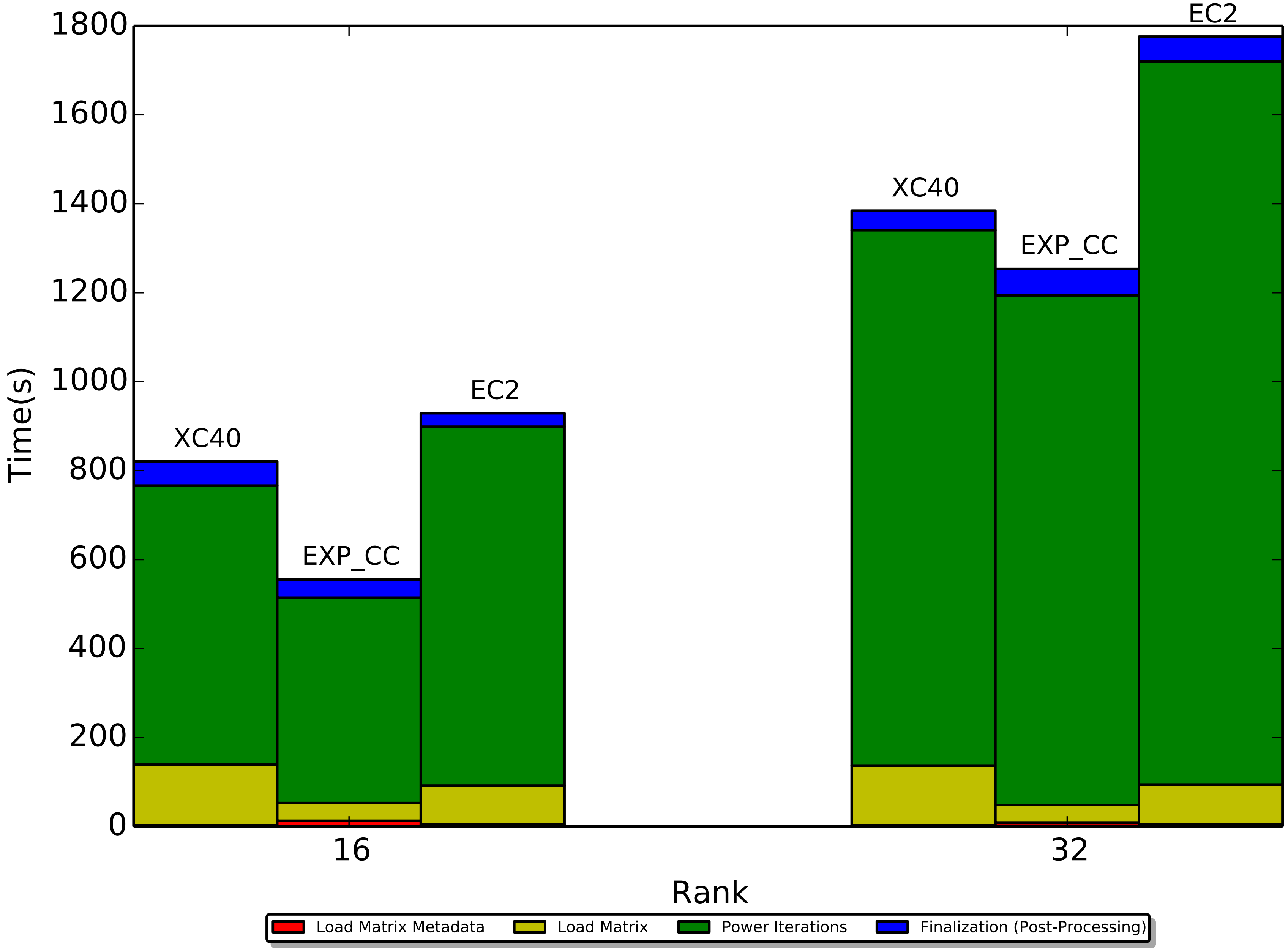
is computed using a treeAggregate operation over the RDD



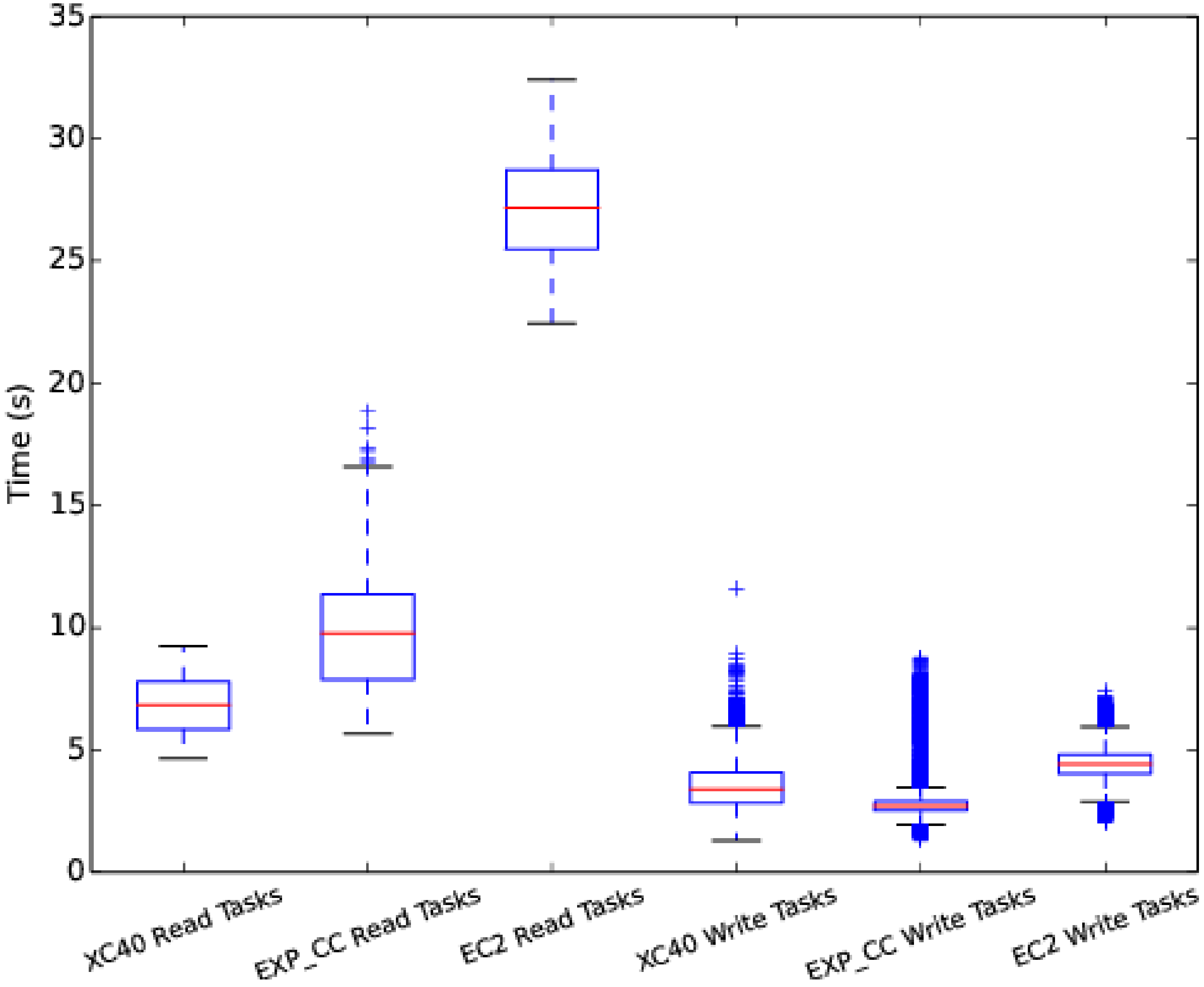
[src: https://databricks.com/blog/2014/09/22/spark-1-1-mllib-performance-improvements.html]

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

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

Computing the truncated PCA

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

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

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

use Lanczos: requires only matrix vector multiplies

1. Compute the truncated EVD of $A^T A$ 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

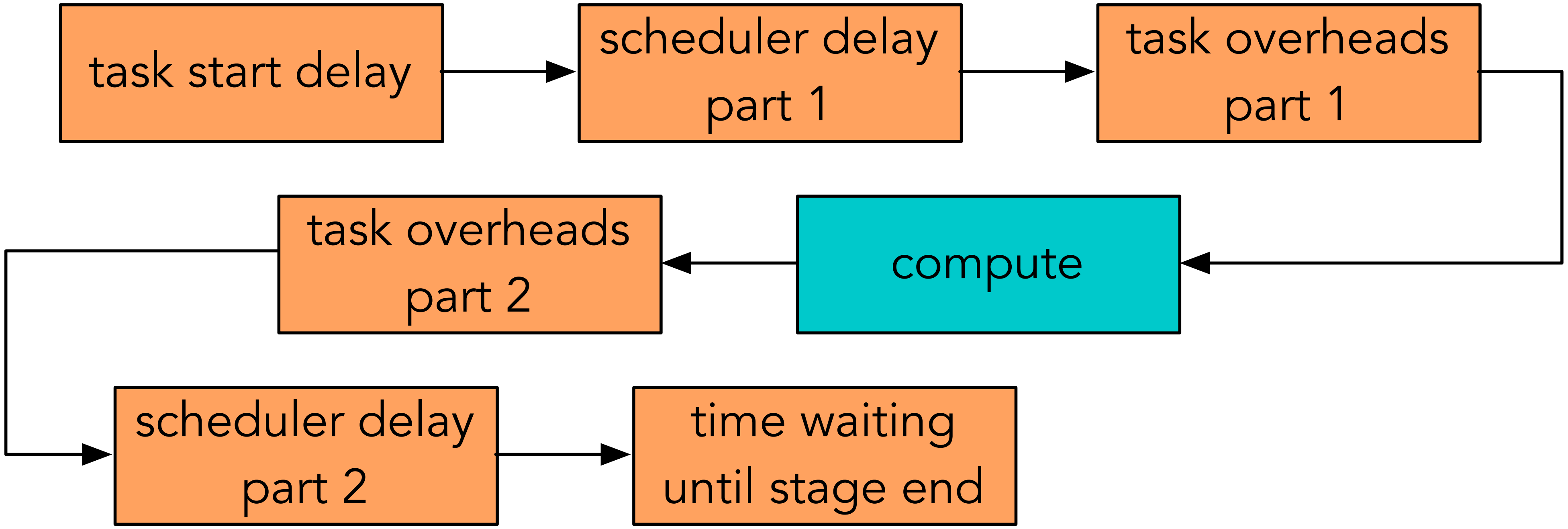
We call the `spark.mllib.linalg.EigenvalueDecomposition` interface to the ARPACK implementation of the Lanczos method

This requires a function which computes a matrix-product against A^{T}A

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

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

Spark Overheads: the view of one task



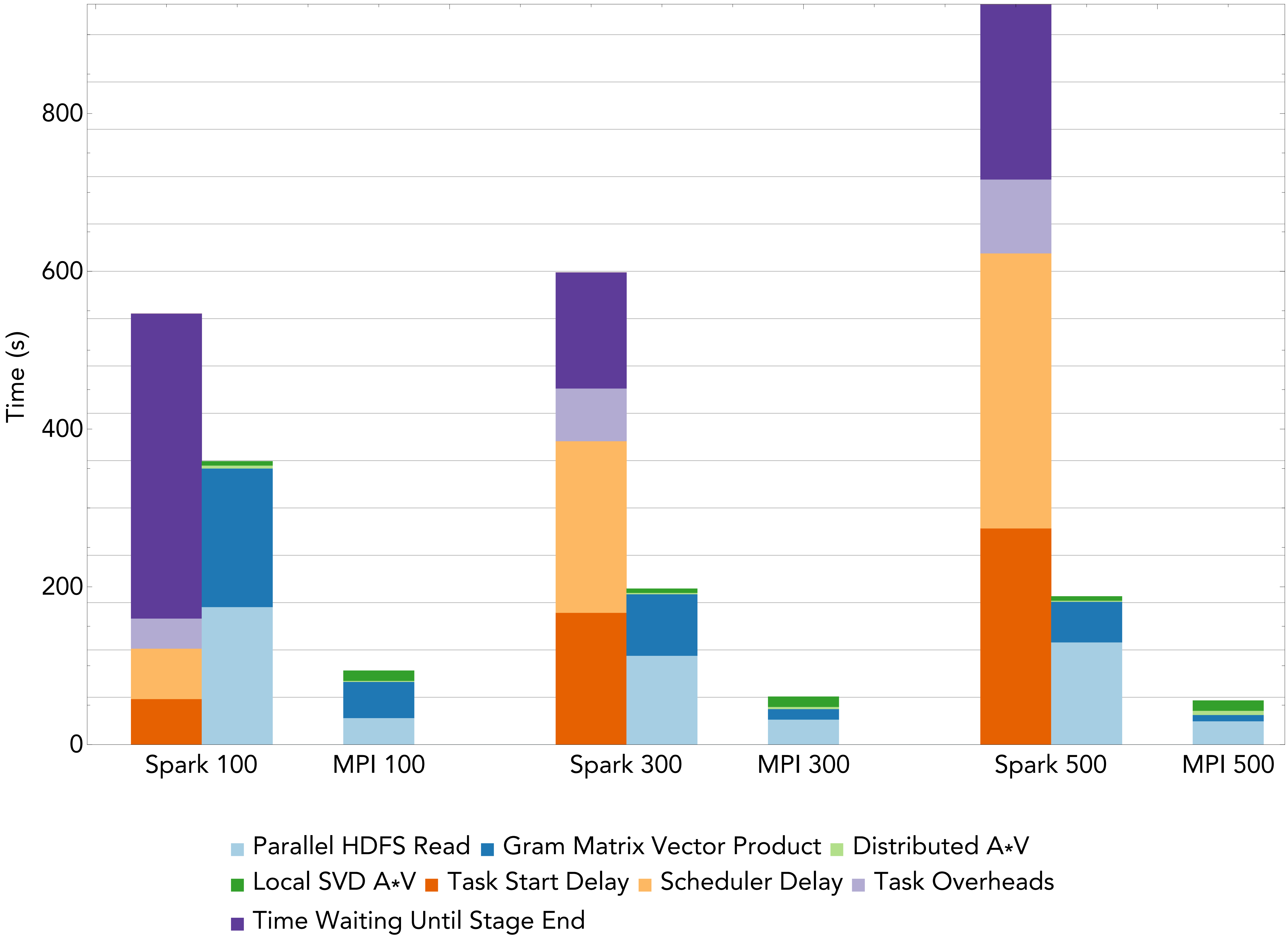
task start delay = (time between stage start and when driver sends task to executor)

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

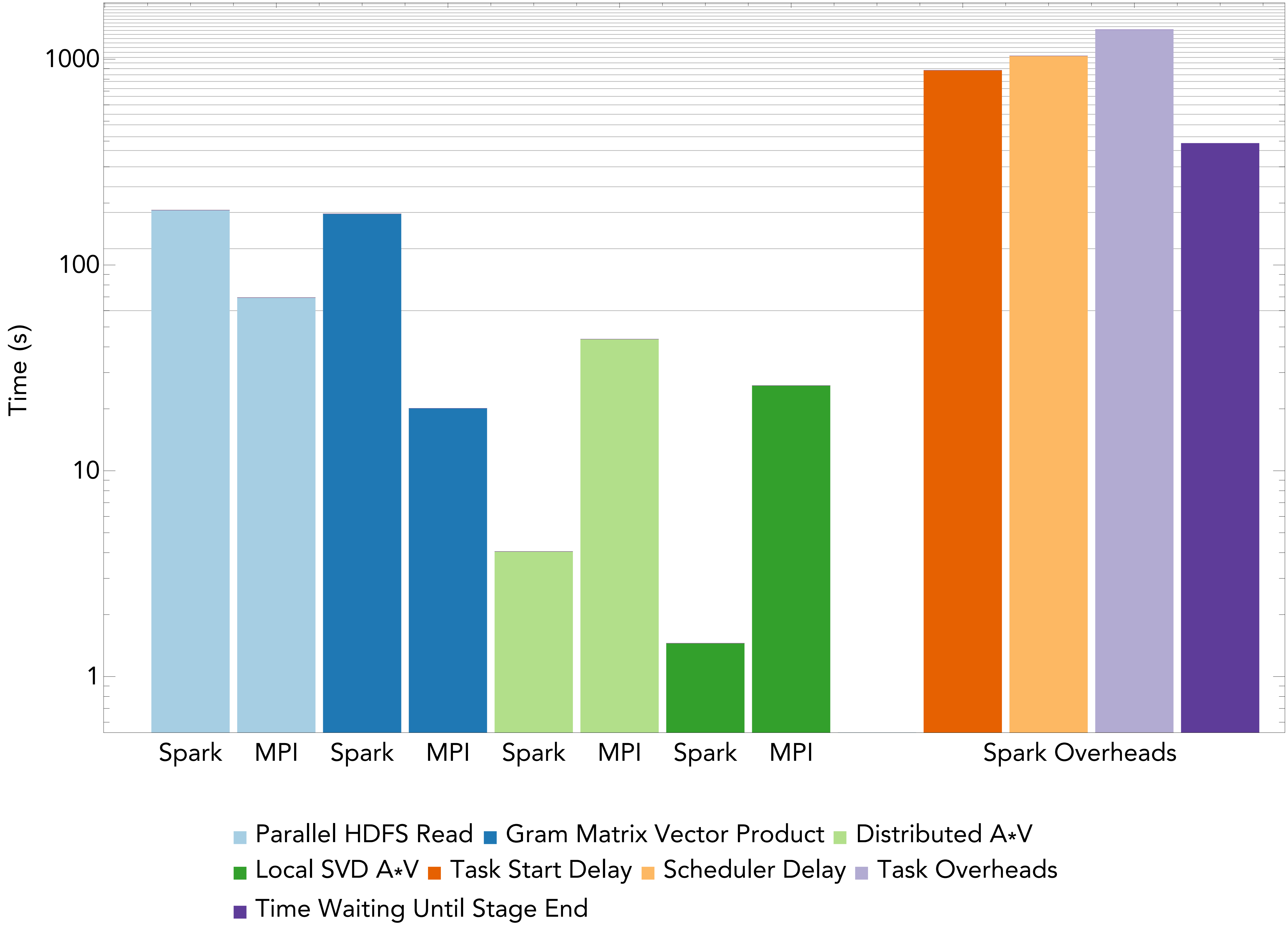
task overhead time = (fetch wait time) + (executor deserialize time) + (result serialization time) + (shuffle write time)

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

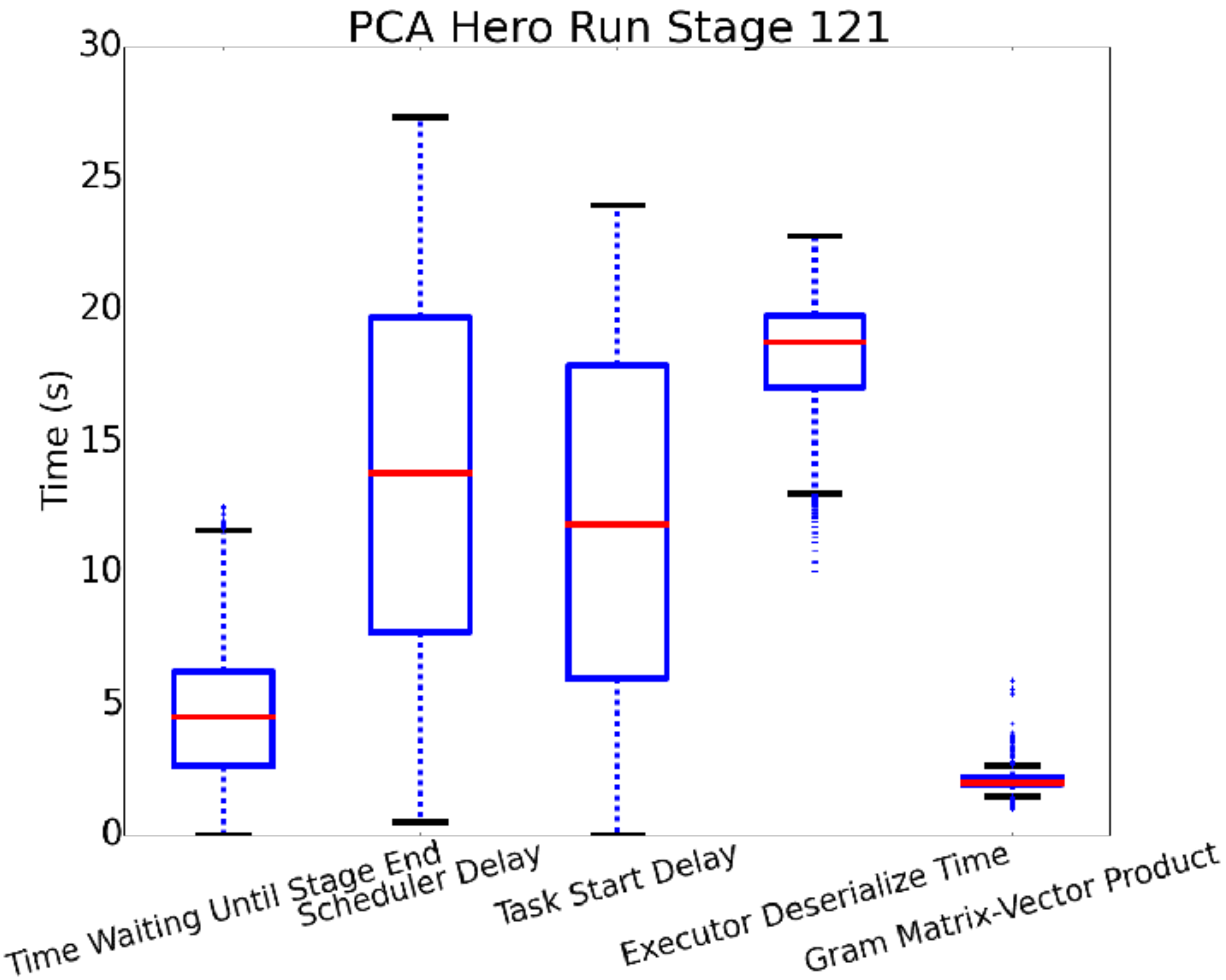
PCA Run Times: rank 20 PCA of 2.2TB Climate



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)

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

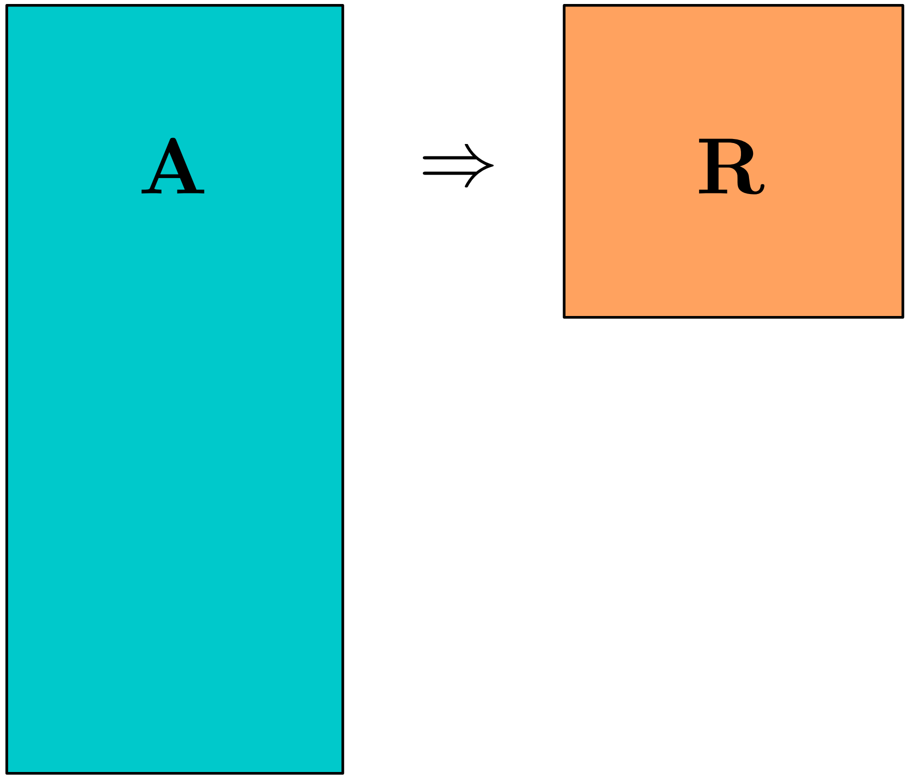
In general, NMF factorizations are non-unique and NP-hard to compute for a fixed rank.

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

Nearly-Separable NMF

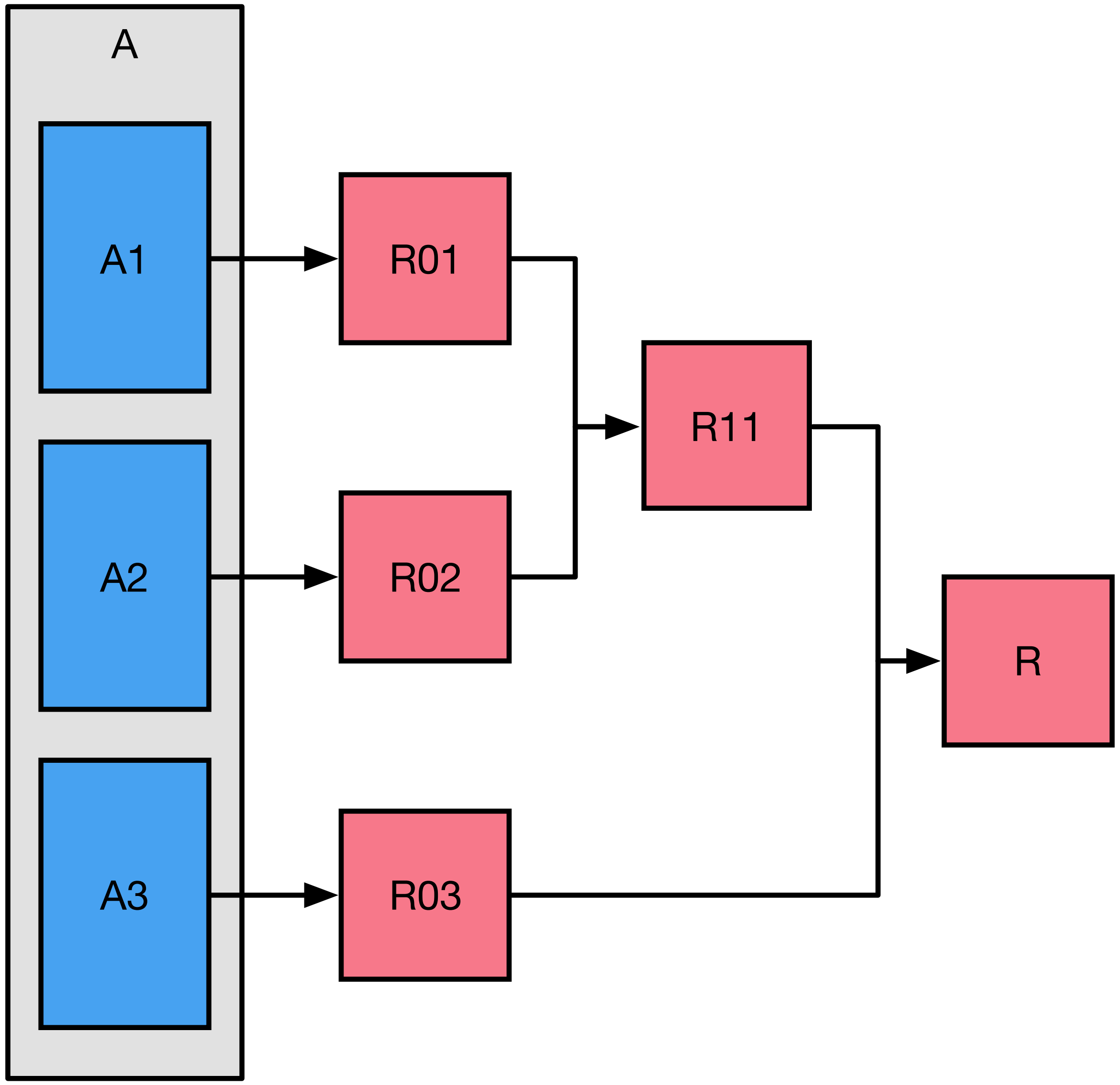
Assumption: *some k-subset of the columns of A comprise a good W*

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

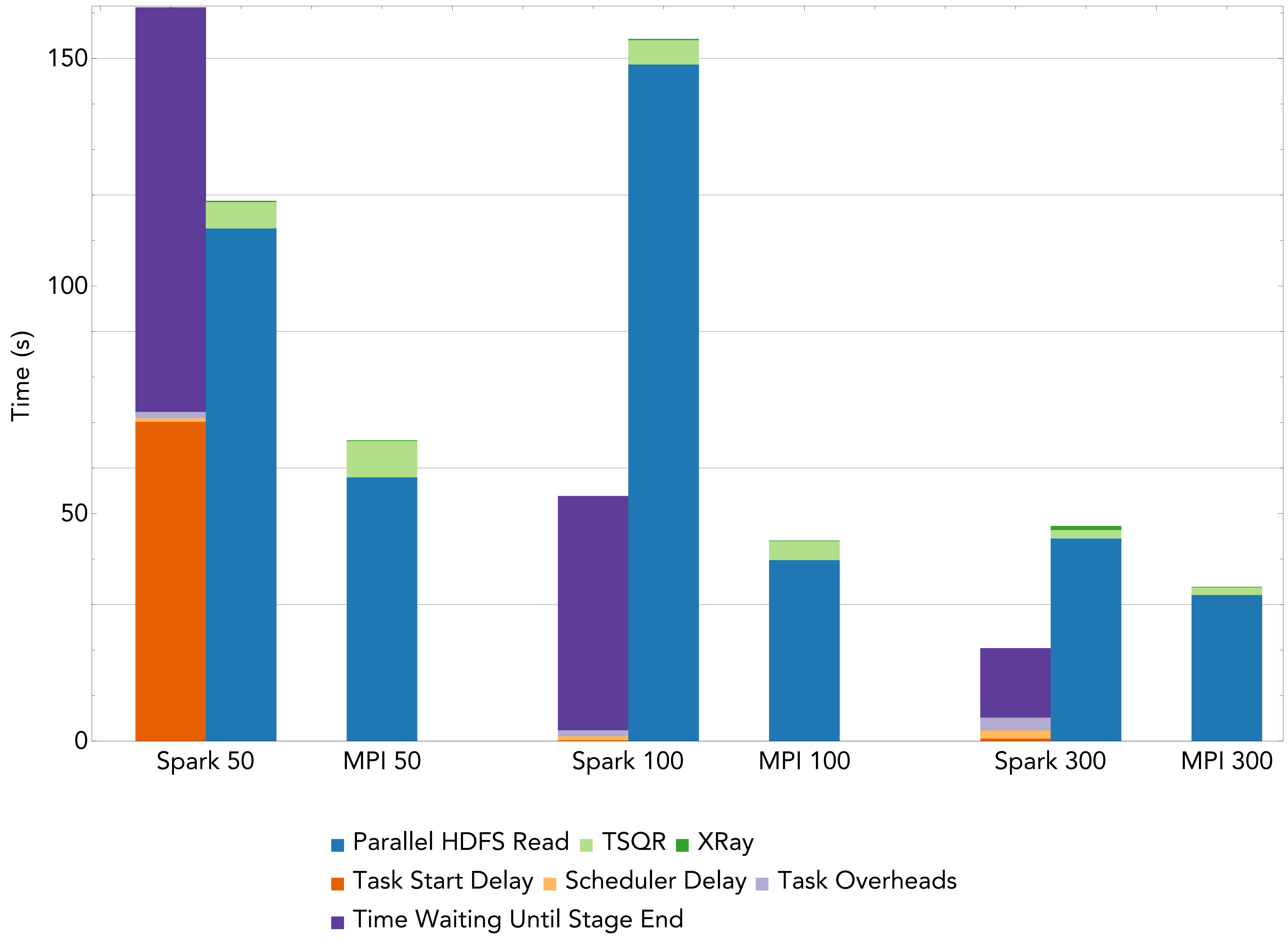
Tall-Skinny QR (TSQR)



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 2x-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). A more efficient algorithm is needed

- ***H5Spark performance is inconsistent*** this needs more work

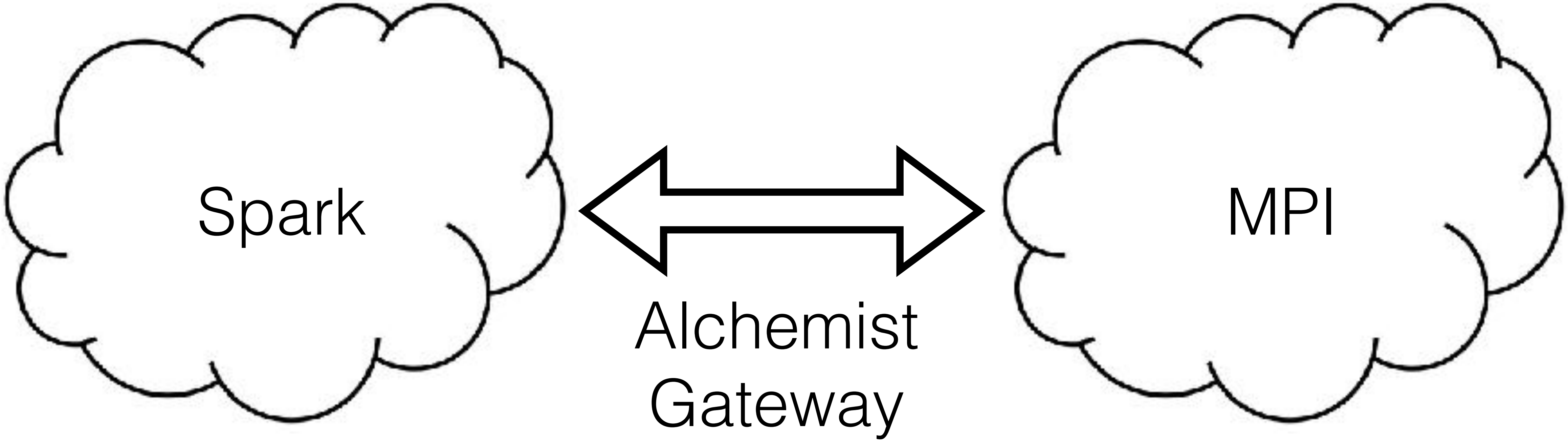
- The gaps in performance suggests it may be better to ***investigate efficiently interfacing MPI-based codes with Spark***

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) Sends the metadata for input and 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

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)
…
```

- Technical Report on Spark performance for Terabyte-scale Matrix Decompositions (accepted to IEEE BigData): *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*

Contact me at *gittens@icsi.berkeley.edu*

Thank you