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

Alex Gittens[1] Aditya Devarakonda[2] Evan Racah[3] Michael Ringenburg[4]
Lisa Gerhardt[3] Jey Kottaalam[5] Jialin Liu[3] Kristyn Maschhoff[4]
Shane Canon[3] Jatin Chhugani[6] Pramod Sharma[4] Jiyan Yang[7]
James Demmel[8] Jim Harrell[4] Venkat Krishnamurthy[4] Michael W. Mahoney[1]
Prabhat[3*†‡§]

July 1, 2016

Abstract

We explore the trade-offs of performing linear algebra using Apache Spark, compared to traditional C and MPI implementations on HPC platforms. Spark is designed for data analytics on cluster computing platforms with access to local disks and is optimized for data-parallel tasks. We examine three widely-used and important matrix factorizations: NMF (for physical plausability), PCA (for its ubiquity) and CX (for data interpretability). We apply these methods to TB-sized problems in particle physics, climate modeling and bioimaging. The data matrices are tall-and-skinny which enable the algorithms to map conveniently into Spark's data-parallel model. We perform scaling experiments on up to 1600 Cray XC40 nodes, describe the sources of slowdowns, and provide tuning guidance to obtain high performance.

1 Introduction

Modern scientific progress relies upon experimental devices, observational instruments, and scientific simulations. These important modalities produce massive amounts of complex data: in High Energy Physics, the LHC project produces PBs of data; smaller-scale projects such as Daya Bay produce 100s of TBs. In Climate science, the worldwide community relies upon distributed access to the CMIP-5 archive, which is several PBs in size. In Biosciences, multi-modal imagers can acquire 100GBs-TBs of data. These projects spend a considerable amount of effort in data movement and data management issues, but the key step in gaining scientific insights is *data analytics*. Several scientific domains are currently rate-limited by access to productive and performant data analytics tools.

Some of the most important classes of scientific data analytics methods rely on matrix algorithms. Matrices provide a convenient mathematical structure with which to model data arising in a broad range of applications: an $m \times n$ real-valued matrix A provides a natural structure for

[*1]ICSI and Department of Statistics, UC Berkeley
[†2]EECS, UC Berkeley
[‡3]NERSC, Lawrence Berkeley National Laboratory
[§4]Cray, Inc.

encoding information about m objects, each of which is described by n features; alternatively, an $n \times n$ real-values matrix A can be used to describe the correlations between all pairs of n data points. Matrix factorizations are common in numerical analysis and scientific computing, where the emphasis is on running time, largely since they are used simply to enable the rapid solution of linear systems and related problems. In statistical data analysis, however, matrix factorizations are typically used to obtain some form of lower-rank (and therefore simplified) approximation to the data matrix A to enable better understanding the structure of the data [21]. In particular, rather than simply providing a mechanism for solving another problem quickly, the actual components making up a factorization are of prime concern. Thus, it is of interest to understand how popular factorizations interact with other aspects of the large-scale data analysis pipeline.

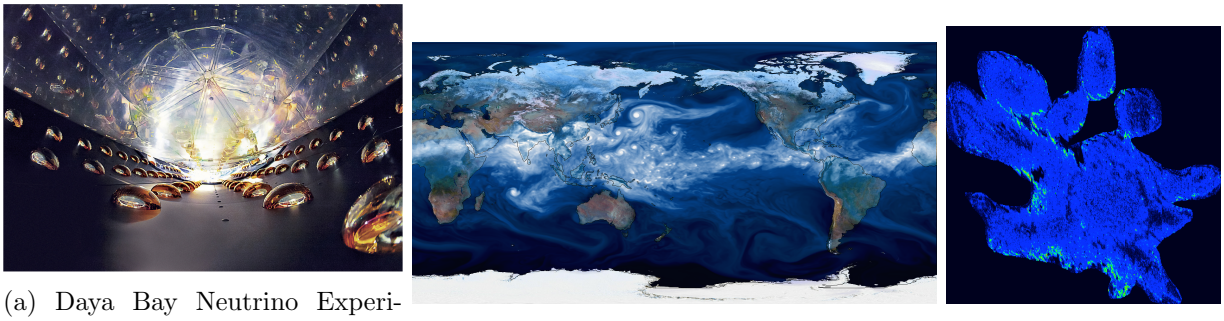
Along these lines, we have recently seen substantial progress in the development and adoption of Big Data software frameworks such as Hadoop/MapReduce [10] and Spark [59]. These frameworks have been developed for industrial applications and commodity datacenter hardware; and they provide high productivity computing interfaces for the broader data science community. Ideally, the scientific data analysis and high performance computing (HPC) communities would leverage the momentum behind Hadoop and Spark. Unfortunately, the performance of such frameworks at scale on conventional HPC hardware has not been investigated extensively. For linear algebraic computations more broadly, and matrix factorizations in particular, there is a gap between the performance of well-established libraries (ScaLAPACK, LAPACK, BLAS, PLASMA, MAGMA, etc. [4, 3]) and toolkits available in Spark. Our work takes on the important task of testing nontrivial linear algebra and matrix factorization computations in Spark for real-world, large-scale scientific data analysis applications. We compare and contrast its performance with C+MPI implementations on HPC hardware. The main contributions of this paper are as follows:

- We develop parallel versions of three leading matrix factorizations (PCA, NMF, CX) in Spark and C+MPI; and we apply them to several TB-sized scientific data sets.
- We conduct strong scaling tests on a XC40 system, and we test the scaling of Spark on up to 1600 nodes.
- We characterize the performance gap between Spark and C+MPI for matrix factorizations, and comment on opportunities for future work in Spark to better address large scale scientific data analytics on HPC platforms.

2 Science Drivers and Data sets

In this study, we choose leading data sets from experimental, observational, and simulation sources, and we identify associated data analytics challenges. The properties of these data sets are summarized in Table 1.

The Daya Bay Neutrino Experiment. The Daya Bay Neutrino Experiment (Figure 1a) is situated on the southern coast of China. It detects antineutrinos produced by the Ling Ao and Daya Bay nuclear power plants and uses them to measure theta-13, a fundamental constant that helps describe the flavor oscillation of neutrinos. In 2012 the Daya Bay experiment measured this with unprecedented precision. This result was named one of the Science magazines top ten breakthroughs of 2012, and this measurement has since been refined considerably [5].



(a) Daya Bay Neutrino Experiment (b) CAM5 Simulation (c) Mass-Spec Imaging

Figure 1: Sources of various data sets used in this study

Table 1: Summary of the matrices used in our study

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

The Daya Bay Experiment consists of eight smaller detectors, each with 192 photomultiplier tubes that detect light generated by interaction of anti-neutrinos from the nearby nuclear reactors. Each detector records the total charge in each of the 192 photomultiplier tubes, as well as the time the charge was detected. For this analysis we used a data array comprised of the sum of the charge for every photomultiplier tube from each Daya Bay detector. This data is well suited to NMF analysis since accumulated charge will always be positive (with the exception of a few miscalibrated values). The extracted data was stored as HDF5 files with 192 columns, one for each photomultiplier tube, and a different row for each discrete event in the detectors. The resulting data set is a sparse 1.6 TB matrix. The specific analytics problem that we tackle in this paper is that of finding characteristic patterns or signatures corresponding to various event types. Successfully "segmenting" and classifying a multiyear long timeseries into meaningful events can dramatically improve the productivity of scientists and enable them to focus on anomalies, which can in turn result in new physics results.

Climate Science. Climate scientists rely on HPC simulations to understand past, present and future climate regimes. Vast amounts of 3D data (corresponding to atmospheric and ocean processes) are readily available in the community. Traditionally, the lack of scalable analytics methods and tools has prevented the community from analyzing full 3D fields; typical analysis is thus performed only on 2D spatial averages or slices. The most widely used tool for extracting important patterns from the measurements of atmospheric and oceanic variables is the Empirical Orthogonal Function (EOF) technique. EOFs are popular because of their simplicity and their ability to reduce the dimensionality of large nonlinear, high-dimensional systems into fewer dimensions while preserving the most important patterns of variations in the measurements. Mathematically, EOFs

are exactly PCA decompositions.

In this study, we consider the Climate Forecast System Reanalysis Product [50]. Global Ocean temperature data, spatially resolved at 360 x 720 x 41 (latitude x longitude x depth) and 6-hour temporal resolution is analyzed. The CFSR data set yields a dense 2.2TB matrix. We also process a CAM5 0.25-degree atmospheric humidity data set [53] (Figure 1b). The grid is 768 x 1158 x 30 (latitude x longitude x height) and data is stored every 3 hours. The CAM5 data set spans 28 years, and it yields a dense 16TB matrix. The specific analytics problem that we tackle is finding the principal causes of variability in large scale 3D fields. A better understanding of the dynamics of large-scale modes of variability in the ocean and atmosphere may be extracted from full 3D EOFs.

Mass-Spectrometry Imaging. Mass spectrometry measures ions derived from the molecules present in a biological sample. Spectra of the ions are acquired at each location (pixel) of a sample, allowing for the collection of spatially resolved mass spectra. This mode of analysis is known as *mass spectrometry imaging (MSI)*. The addition of *ion-mobility separation (IMS)* to MSI adds another dimension, drift time. The combination of IMS with MSI is finding increasing applications in the study of disease diagnostics, plant engineering, and microbial interactions. Unfortunately, the scale of MSI data and complexity of analysis presents a significant challenge to scientists: a single 2D-image may be many gigabytes and comparison of multiple images is beyond the processing capabilities available to many scientists. The addition of IMS exacerbates these problems.

We analyze one of the largest (1TB sized) mass-spec imaging data sets in the field, obtained from a sample of the plant *Lewis Dalisay Peltatum* (Figure 1c). The MSI measurements are formed into a sparse matrix whose rows and columns correspond to pixel and (τ, m/z) values of ions, respectively. Here τ denotes drift time and m/z is the mass-to-charge ratio. The sheer size of this data set has previously made complex analytics intractable. CX decompositions allow for the possibility of identifying small numbers of columns (ions) in the original data that reliably explain a large portion of the variation in the data.

3 Methods

Given an $m \times n$ data matrix A, low-rank matrix factorization methods aim to find two or more smaller matrices such that their product is a good approximation to A. That is, they aim to find matrices Y and Z such that

$$\underset{m\times n}{A} \approx \underset{m\times k}{Y} \times \underset{k\times n}{Z}.$$

Low-rank matrix factorization methods are important tools in linear algebra and numerical analysis, and they find use in a variety of scientific fields and scientific computing. These methods have the following advantages:

- They are often useful in data compression, as smaller matrices can be stored more efficiently.
- In some cases, the results of analysis using them are more interpretable. For example, in imaging analysis, the original images can be reconstructed using linear combination of basis images.
- They can be viewed as a basic dimension reduction technique.

- In many modern applications, data sets containing a massive number of rows or columns are becoming more common, which makes it difficult for data visualization or applying classic algorithms, but low-rank approximation methods express every data point in a low-dimensional space defined by only a few features.

Throughout, we assume the data matrix A has size $m \times n$ and rank r, with $r \ll n \ll m$; this "tall-skinny", highly rectangular setting is common in practice.

Matrix factorizations are also widely-used in statistical data analysis [21]. Depending on the particular application, various low-rank factorization techniques are of interest. Popular choices include the singular value decomposition [16], principal component analysis [23], rank-revealing QR factorization [17], nonnegative matrix factorization (NMF) [31], and CX/CUR decompositions [40]. In this work, we consider the PCA decomposition, due to its ubiquity, as well as the NMF and CX/CUR decompositions, due to their usefulness in scalable and interpretable data analysis. In the remainder of the section, we briefly describe these decompositions and the algorithms we used in our implementations, and we also discuss related implementations.

Prior Work. The body of theoretical and practical work surrounding distributed low-rank matrix factorization is large and continuously growing. The HPC community has produced many high quality packages specifically for computing partial SVDs of large matrices: Propack [29], Blopex [26], and Anasazi [6], among others. We refer the interested reader to [20] for a well-written survey. As far as we are aware, there are no published HPC codes for computing CX decompositions, but several HPC codes exist for NMF factorization [24].

The machine learning community has produced many packages for computing a variety of low-rank decompositions, including NMF and PCA, typically using either an alternating least squares (ALS) or a stochastic gradient descent approach [13, 57, 27]. ALS algorithms can produce high precision decompositions, but have a high computational and communication cost, while SGD algorithms produce low precision decompositions with comparatively lower costs. We mention a few of the high-visibility efforts in this space. The earlier work [37] developed and studied a distributed implementation of the NMF for general matrices under the Hadoop framework, while [8] introduced a scalable NMF algorithm that is particularly efficient when applied to tall-and-skinny matrices. We implemented a variant of the latter algorithm in Spark, as our data matrices are tall-and-skinny. The widely used MLLib library, packaged with Spark itself, provides some linear algebra datatypes (vectors and matrices) and implementations of basic linear algebra routines [45]; we note that the PCA algorithm implemented in MLLib is almost identical to our concurrently developed implementation. The Sparkler system introduces a memory abstraction to the Spark framework which allows for increased efficiency in computing low-rank factorizations via distributed SGD, [35], but such factorizations are not appropriate for scientific applications which require high precision.

The 2011 report on the DOE Magellan cloud computing project [55] discusses qualitative experience implementing numerical linear algebra in Hadoop, specifically relating to the tall-skinny QR algorithm. Our contribution is the provision of, for the first time, a detailed investigation of the scalability of three low-rank factorizations using the linear algebra tools and bindings provided in Spark's baseline MLLib [45] and MLMatrix [58] libraries. By identifying the causes of the slowdowns in these algorithms that exhibit different bottlenecks (e.g. I/O time versus synchronization overheads), we provide a clear indication of the issues that one encounters attempting to do serious distributed linear algebra using Spark. To ensure that our comparison of Spark to MPI is fair, we

Algorithm 1 PCA Algorithm

Require: $A \in \mathbb{R}^{m \times n}$, rank parameter $k \leq \text{rank}(A)$.
Ensure: $U_k \Sigma_k V_k^T = \text{PCA}(A, k)$.
1: Let $(V_k, _) = \text{IRAM}(\textsc{MultiplyGramian}(A, \cdot), k)$.
2: Let $Y = \textsc{Multiply}(A, V_k)$.
3: Compute $(U_k, \Sigma_k, _) = \text{SVD}(Y)$.

Algorithm 2 MultiplyGramian Algorithm

Require: $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times k}$.
Ensure: $X = A^T AB$.
1: Initialize $X = 0$.
2: **for** each row a in A **do**
3: $\quad X \leftarrow X + aa^T B$.
4: **end for**

implemented the same algorithms in Spark and MPI, drawing on a common set of numerical linear algebra libaries for which Spark bindings are readily available (Blas, Lapack, and Arpack).

Principal Components Analysis. Principal component analysis (PCA) is closely related to the singular value decomposition (SVD). In particular, the PCA decomposition of a matrix A is the SVD of the matrix formed by centering each column of A (i.e., removing the mean of each column) and considering $A^T A$ (or AA^T). The SVD is the most fundamental low-rank matrix factorization because it provides the best low-rank matrix approximation with respect to any unitarily invariant matrix norm. In particular, for any target rank $k \leq r$, the SVD provides the minimizer of the optimization problem

$$\min_{\text{rank}(\tilde{A})=k} \|A - \tilde{A}\|_F, \tag{1}$$

where the Frobenius norm $\|\cdot\|_F$ is defined as $\|X\|_F^2 = \sum_{i=1}^m \sum_{j=1}^n X_{ij}^2$. The solution to (1) is given by the truncated SVD, i.e., $A_k = U_k \Sigma_k V_k^T$, where the columns of U_k and V_k are the top k *left and right singular vectors*, respectively, and Σ_k is a diagonal matrix containing the corresponding top k *singular values*.

Direct algorithms for computing the PCA decomposition scale as $\mathcal{O}(mn^2)$, so are not feasible for the scale of the problems we consider. Instead, we use the iterative algorithm presented in Algorithm 1: in step 1, a series of distributed matrix-vector products against $A^T A$ (MultiplyGramian) are used to extract V_k by applying the implicitly restarted Arnoldi method (IRAM) [33], then in step 2 a distributed matrix-matrix product followed by a collect is used to bring AV_k to the driver. Step 3 occurs on the driver, and computes a final SVD on AV_k to extract the top left singular vectors U_k and the corresponding eigenvalues Σ_k. Here QR and SVD compute the "thin" versions of the QR and SVD decompositions [16]. (Algorithm 1 calls MultiplyGramian, which is summarized in Algorithm 2).

ML-Lib, Spark's machine learning library, provides implementations of the SVD and PCA, as well as an alternating least squares algorithm for low-rank factorization [44] (the PCA algorithm used in ML-Lib is very similar to Algorithm 1, but explicitly computes $A^T A$). Similarly, the Apache Mahout project provides Hadoop and Spark implementations of the PCA, RandomizedSVD, and

Algorithm 3 NMF Algorithm

Require: $A \in \mathbb{R}^{m \times n}$ with $A \geq 0$, rank parameter $k \leq \text{rank}(A)$.
Ensure: $WH \approx A$ with $W, H \geq 0$
1: Let $(_, R) = \text{TSQR}(A)$.
2: Let $(\mathcal{K}, H) = \textsc{Xray}(R, k)$.
3: Let $W = A(:, \mathcal{K})$.

ALS algorithms. However, to our knowledge, there are no published investigations into the impact Spark or MapReduce's infrastructure has on the performance of these algorithms.

Nonnegative Matrix Factorization. Although the PCA provides a mathematically optimal low-rank decomposition in the sense of (1), in some scientific applications retaining sparseness and interpretability is as important as explaining variability. Various nonnegative matrix factorizations (NMFs) provide interpretable low-rank matrix decompositions when the columns of A are nonnegative and can be viewed as additive superpositions of a small number of positive factors [14]. NMF has found applications, among other places, in medical imaging [32], facial recognition [18], chemometrics [47], hyperspectral imaging [15], and astronomy [49].

The basic optimization problem solved by NMF is

$$\min_{W,H \geq 0} \|A - WH\|_F, \tag{2}$$

where $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ are entry-wise nonnegative matrices. Typical approaches attempt to solve this non-convex problem by using block coordinate optimizations that require multiple passes over A [25]. We adopt the one-pass algorithm of [8]. This approach makes the assumption that W can be formed by *selecting* columns from A. In this setting, the columns of A constituting W as well as the corresponding H can be computed directly from the (much smaller) R factor in a thin QR factorization of A. More details are given in Algorithm 3: in step 1, a one pass distributed tall-skinny QR (TSQR) factorization [11] is used to compute the R factor of A; in step 2, which occurs on the driver, the Xray algorithm of [28] is applied to R to simultaneously compute H and the column indices $\mathcal{K}$ of W in A. Finally, W can be explicitly computed once $\mathcal{K}$ is known.

The ML-Lib and Mahout libraries provide alternating least squares-based NMF implementations in Spark and MapReduce, respectively, and several other NMF implementations are available for the MapReduce framework [37, 36, 8]. We note that [8] introduced Algorithm 3. None of these works quantified the performance costs of implementing these algorithms in the Spark or MapReduce frameworks.

CX/CUR decompositions. CX (and related CUR) decompositions are low-rank matrix decompositions that are expressed in terms of a small number of actual columns/rows, i.e, actual data elements, as opposed to eigencolumns/eigenrows. As such, they have been used in scientific applications where coupling analytical techniques with domain knowledge is at a premium, including genetics [48], astronomy [56], and mass spectrometry imaging [54]. To find a CX decomposition, we seek matrices C and X such that the approximation error $\|A - CX\|_F$ is small and C is an $m \times k$ matrix comprising of k actual columns of A and X is a $k \times n$ matrix.

The randomized algorithm of [12] generates a C whose approximation error is, with high probability, within a multiplicative factor of $(1 + \varepsilon)$ of the optimal error obtainable with a low-rank

Algorithm 4 CX Algorithm

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

Ensure: C.

1: Initialize $B \in \mathbb{R}^{n \times \ell}$ by sampling $B_{ij} \sim \mathcal{N}(0, 1)$.
2: **for** q times **do**
3: $\quad B \leftarrow \textsc{MultiplyGramian}(A, B)$
4: $\quad (B, _) \leftarrow \mathrm{QR}(B)$
5: **end for**
6: Let Q be the first k columns of B.
7: Let $Y = \textsc{Multiply}(A, Q)$.
8: Compute $(U, \Sigma, \tilde{V}^T) = \mathrm{SVD}(Y)$.
9: Let $V = Q\tilde{V}$.
10: Let $\ell_i = \sum_{j=1}^{k} v_{ij}^2$ for $i = 1, \ldots, n$.
11: Define $p_i = \ell_i / \sum_{j=1}^{d} \ell_j$ for $i = 1, \ldots, n$.
12: Randomly sample k columns from A in i.i.d. trials, using the importance sampling distribution $\{p_i\}_{i=1}^{n}$.

decomposition:

$$\|A - CX\|_F \leq (1 + \varepsilon)\|A - A_k\|_F.$$

This algorithm takes as input the (approximate or exact) *leverage scores* of the columns of A. The leverage score of the j-th column of A is defined in terms of V_k, the matrix of top k right singular vectors:

$$\ell_i = \sum_{j=1}^{k} (V_k)_{ij}^2; \tag{3}$$

the leverage scores can be approximated using an approximation to V_k. The CX algorithm uses those scores as a sampling distribution to select k columns from A to form C; once the matrix C is determined, the optimal matrix X that minimizes $\|A - CX\|_F$ can be computed accordingly; see [12] for the details of this construction.

The computational cost of the CX decomposition is determined by the cost of computing V_k exactly or approximately. To approximate V_k, we use the RandomizedSVD algorithm introduced in [42, 43]. We refer the readers to [19, 39] for more details. Importantly, the algorithm runs in $\mathcal{O}(mn \log k)$ time and needs only a constant number of passes over the data matrix (q+1), where q is an input in Algorithm 4). The RandomizedSVD algorithm comprises the first nine steps of Algorithm 4. The running time cost for RandomizedSVD is dominated by a distributed matrix-matrix multiplication appearing in Steps 3 and 7 of Algorithm 4. After Step 7, Y is collected the remaining computations are carried out on the driver.

To the best of our knowledge, this is the first published work to investigate the performance of the CX algorithm on any large-scale distributed/parallel platform.

4 Implementation

Spark is a parallel computing framework, built on the JVM, that adheres to the data parallelism model. A Spark cluster is composed of a driver process and a set of executor processes. The driver schedules and manages the work, which is carried out by the executors. The basic unit of work in Spark is called a task. A single executor has several slots for running tasks (by default, each core of an executor is mapped to one task) and runs several concurrent tasks in the course of calculations. Spark's primitive datatype is the resilient distributed data set (RDD), a distributed array that is partitioned across the executors. The user-defined code that is to be run on the Spark cluster is called an application. When an application is submitted to the cluster, the driver analyses its computation graph and breaks it up into jobs. Each job represents an action on the data set, such as counting the number of entries, returning data set entries, or saving a data set to a file. Jobs are further broken down into stages, which are collections of tasks that execute the same code in parallel on a different subset of data. Each task operates on one partition of the RDD. Communication occurs only between stages, and takes the form of a shuffle, where all nodes communicate with each other, or a collect, where all nodes send data to the driver.

Spark employs a lazy evaluation strategy for efficiency. All Spark operations that have no immediate side-effects other than returning an RDD are deferred if possible. Instead, deferrable operations simply create an entry in the program's computation graph, recording the input dependencies and capturing any closures and values needed. This approach allows Spark to defer computations as much as possible, and when the evaluation is unavoidable the entire Spark job can be examined by Spark's scheduler. This allows the Spark execution engine to batch together related operations, optimize data locality, and perform better scheduling. A major benefit of Spark over MapReduce is the use of in-memory caching and storage so that data structures may be reused rather than being recomputed. Because Spark tracks the computation graph and the dependencies required for the generation of all RDDs, it natively provides fault-tolerance: given the lineage of the RDD, any lost partitions of that RDD can be recomputed as needed.

Implementing Matrix Factorizations in Spark. All three matrix factorizations store the matrices in a row-partitioned format. This enables us to use data parallel algorithms and match Spark's data parallel model.

The MultiplyGramian algorithm is the computational core of the PCA and CX algorithms. This algorithm is applied efficiently in a distributed fashion by observing that if the i-th executor of ℓ stores the block of the rows of A denoted by $A_{(i)}$, then $A^T AB = \sum_{i=1}^{\ell} A_{(i)}^T A_{(i)} B$. Thus MultiplyGramian requires only one round of communication. The local linear algebra primitives QR and SVD needed for PCA and CX are computed using the LAPACK bindings of the Breeze numerical linear algebra library. The Netlib-Java binding of the ARPACK library supplies the IRAM primitive required by the PCA algorithm.

The NMF algorithm has as its core the tall-skinny QR factorization, which is computed using a tree reduction over the row-block partitioned A. We used the TSQR implementation available in the Ml-Matrix package. To implement the XRay algorithm, we use the ML-Lib non-negative least squares solver.

Implementing Matrix Factorizations in C+MPI. NMF, PCA and CX require linear algebra kernels that are available in widely-used libraries such as Intel MKL, Cray LibSci, and arpack-ng.

We use these three libraries in our implementations of the matrix factorizations. The data matrices are represented as 1D arrays of double-precision floating point numbers and are partitioned across multiple nodes using a block row partitioned layout. The 1D layout enables us to use matrix-vector products and TSQR as our main computational kernels. We use MPI collectives for inter-processor communication and perform independent I/O using the Cray HDF5 parallel I/O library.

5 Experimental Setup

All performance tests reported in this paper were conducted on the Cori system at NERSC. Cori Phase I is a Cray XC40 system with 1632 dual-socket compute nodes. Each node consists of two 2.3GHz 16-core Haswell processors and 128GB of DRAM. The Cray Aries high-speed interconnect is configured in a "Dragonfly' topology. We use a Lustre scratch filesystem with 27PB of storage, and over 700 GB/s peak I/O performance.

Spark Configuration. We use the Standalone Cluster Manager to run the Spark cluster. This is a collection of scripts that start the driver process and use ssh to start the executor processes on each node. Once the executors are started, they communicate with the driver via akka-tcp. When an application is submitted to the Spark cluster a second java process is spawned by each executor that controls the computation for that application. Sometimes this second process will fail to start and the executor does not participate in the calculation. The exact cause of this is not well known. Running Spark in an encapsulated Shifter image reduces the rate of these failures, which suggests this could be due to a race condition in the code.

H5Spark: Loading HDF5 data natively into Spark. The Daya Bay and climate data sets are stored in HDF5. We utilized the H5Spark [38] package to read this data in as one RDD object. H5Spark provides a parallel I/O interface that efficiently loads TBs of data into the workers' memory and constructs a single RDD. An MPI-like independent I/O is performed in H5Spark to balance the workload. H5Spark partially relies on the Lustre file system striping to achieve high I/O bandwidth. We chose a Lustre configuration optimal for each data set: we stored the Daya Bay data on 72 OSTs and the climate data sets on 140 OSTs, both with striping size of 1MB.

Shifter. Shifter is a framework that delivers docker-like functionality to HPC [22]. It works by extracting images from native formats (such as a Docker image) and converting them to a common format that is optimally tuned for the HPC environment. Shifter allows users with a complicated software stack to easily install them in the environment of their choosing. It also offers considerable performance improvements because metadata operations can be more efficiently cached compared to a parallel file system and users can customize the shared library cache (ldconfig) settings to optimize access to their analysis libraries. In particular, shared library performance, which has long been a pain point on Cray systems, is dramatically improved. For this analysis we used two separate Shifter images. A generic "CCM" image which only contained SSH functionality (which is otherwise absent by default on Cray compute nodes) and a full spark image which contained version 1.5.1 of Spark compiled with OpenBLAS [2] and SSH. The Spark image is available on Docker Hub [1].

Spark Tuning Parameters. Shifter provides a user-controlled option to create a writeable temporary space that is private to each node. This has performance characteristics similar to a local disk. This is created by mounting a writeable loop-back mounted file system which is backed by the parallel file system. This feature is very useful for frameworks like Spark that assume the presence of a local disk that can be used to store node local temporary files and spills. Metadata operations and small I/O transactions can be more efficiently cached on the compute node since, unlike the Lustre scratch file system, it doesn't have to maintain coherency of this file system with other nodes. Most importantly, as the Spark cluster size is scaled up, this approach helps avoid additional pressure on the Lustre Metadata Servers which are the least scalable components of the file system. Since Spark opens and closes files many times, using the loop-back mounted file system as a writable cache can improve performance [9].

We followed general Spark guidelines for Spark configuration values. The driver and executor memory were both set to 100 GB, a value chosen to maximize the memory available for data caching and shuffling while still leaving a buffer to hedge against running the nodes out of memory. Generally we found that fetching an RDD from another node was detrimental to performance, so we turned off speculation (a function that restarts tasks on other nodes if it looks like the task is taking longer than average). We also set the spark locality wait to two minutes, this ensures that the driver will wait at least two minutes before scheduling a task on a node that doesn't have the task's RDD. The total number of spark cores was chosen such that there was a one-to-one correspondence between spark cores and physical cores on each node (with the exception of the 50-node NMF run which used a factor of two more partitions because it ran into hash table size issues). We used the KryoSerializer for deserialization of data. We compiled Spark to use multi-threaded OpenBLAS for PCA.

C+MPI Tuning Parameters. The NMF algorithm uses the Tall-Skinny QR (TSQR) [7, 11] factorization implemented as part of the Communication-Avoiding Dense Matrix Computations (CANDMC) library [51] which links to Intel MKL for optimized BLAS routines using the Fortran interface and ensured that loops were auto-vectorized when possible. We explored multi-threading options with OpenMP but found that it did not significantly improve performance. Applying TSQR on the Daya Bay data set results in a 192×192 upper-triangular matrix. Due to the small size we utilized a sequential non-negative least squares solver by Lawson and Hanson [30] in the XRay algorithm. PCA requires EVD, SVD, matrix-vector products, and matrix-matrix products. We use arpack-ng [34] for the SVD and link to single-threaded Cray LibSci for optimized BLAS routines using the C interface. All experiments were conducted using a flat-MPI configuration with one MPI process per physical core and disabled TurboBoost.

Spark Overheads. To report the overheads due to Spark's communication and synchronization costs, we group them into the following bins, illustrated in Figure 2:

- *Task Start Delay*: the time between the stage start and when the driver sends the task to an executor.
- *Scheduler Delay*: the sum of the time between when the task is sent to the executor and when it starts deserializing on the executor and the time between the completion of the serialization of the result of the task and the driver's reception of the task completion message.
- *Task Overhead Time*: the sum of the fetch wait times, executor deserialize times, result serialization times, and shuffle write times.

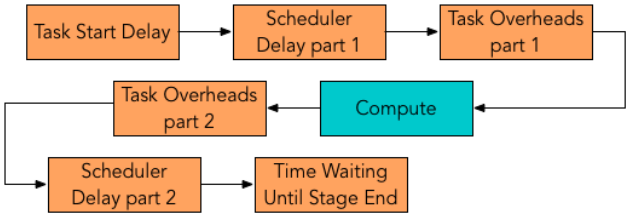


Figure 2: Per task chronological breakdown of Spark overheads.

- *Time Waiting Until Stage End*: the time spent waiting on the final task in the stage to end.

6 Results

6.1 NMF applied to the Daya Bay matrix

The separable NMF algorithm we implemented fits nicely into a data parallel programming model. After the initial distributed TSQR the remainder of the algorithm is computed serially on the driver. The Daya Bay matrix is especially amenable to this approach, as the extreme aspect ratio of the data set implies that the TSQR is particularly efficient.

C+MPI vs. Spark. The TSQR algorithm used performs a single round of communication using a flat binary tree. Because there are few columns, the NMF algorithm is entirely I/O-bound. Figure 3 gives the running time breakdown when computing rank 10 approximations using the MPI implementation of NMF on 50 nodes, 100 nodes, and 300 nodes. Each bin represents the sum, over all stages, of the time spent in that bin by the average task within a stage.

The running time for NMF is overwhelmingly dominated by reading the input. In comparison, TSQR and XRay have negligible running times. Figure 3 shows that the HDF5 read time does not scale linearly with the number of nodes and is the primary source of inefficiency – this is due to saturating the system bandwidth for 72 OSTs. XRay, which is computed on the driver, is a sequential bottleneck and costs 100ms at all node counts. TSQR only improves by tens of milliseconds, costing 501ms, 419ms, and 378ms on 50, 100, and 300 nodes, respectively. This poor scaling can be attributed to hitting a communication bottleneck. Forming the TSQR binary tree is expensive for small matrices, especially using flat MPI. We did not tune our TSQR reduction tree shapes or consider other algorithms since TSQR is not the limiting factor to scalabilty. These results illustrate the importance of I/O scalability when performing terabyte-scale data parallel analytics on a high-performance architecture using MPI.

Figure 3 also illustrates the running time breakdown for the Spark implementation of NMF on 50, 100, and 300 nodes. Unlike the MPI implementation, the Spark implementation incurs significant overheads due to task scheduling, task start delays, and idle time caused by Spark stragglers. For the 50 node run we configured Spark to use double the number of partitions as physical cores because we encountered out-of-memory errors using fewer partitions— this incurs a task start delay overhead because some only half of the total tasks can be executed concurrently. The number of partitions was not doubled for the 100 and 300 node runs, so the task start delay overhead is much smaller for these runs. Similar to the MPI results, most of the running time

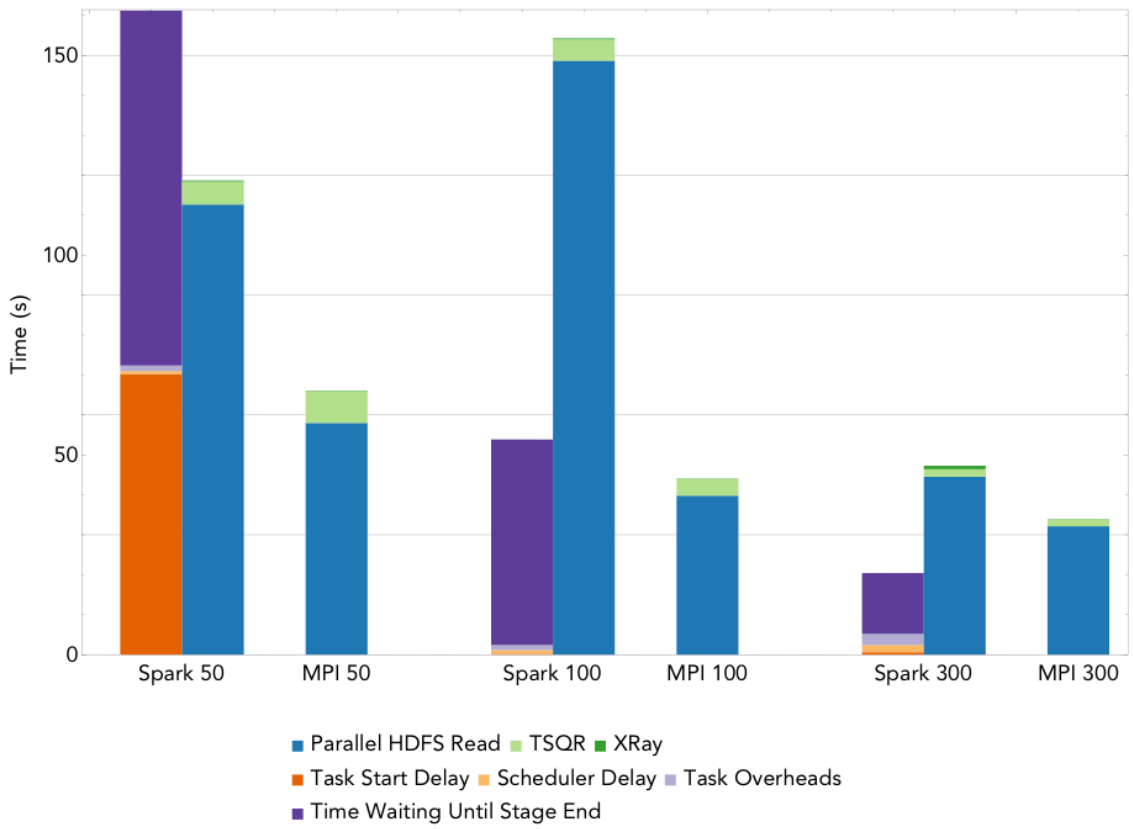


Figure 3: Running time breakdown when using NMF to compute a rank 10 approximation to the 1.6TB Daya Bay matrix at node counts of 50, 100, and 300. Each bin depicts the sum, over all stages, of the time spent in that bin by the average task within a stage.

is spent in I/O and Spark overheads, with a small amount of time spent in TSQR and XRay. Figure 3 shows that the Spark implementation exhibits good strong scaling behavior up to 300 nodes. Although the NMF algorithm used is entirely data parallel and suitable for Spark, we observed a 4×, 4.6×, and 2.3× performance gap on 50, 100, and 300 nodes, respectively, between Spark and MPI.

Figure 4 shows the parallel efficiencies of the MPI and Spark implementations of NMF, normalized to the 50 node running time of the respective parallel frameworks. MPI NMF is completely dominated by I/O and the results are primarily indicative of scaling issues in the I/O subsystem. Spark NMF displays good scaling with more nodes; this is reflected in the parallel efficiency. However, the scaling is due primarily to decreases in the Spark overhead.

Science Implications. We are currently investigating the results of the NMF decomposition. Preliminary analysis indicates that we will need to augment the input data with non-linear features to make the input signals invariant to rotations and translations. Our eventual goal is to learn event-specific classifiers from the loadings of the NMF basis vectors. The classification will enable us to accomplish the final goal of segmenting and classifying the timeseries of sensor measurements.

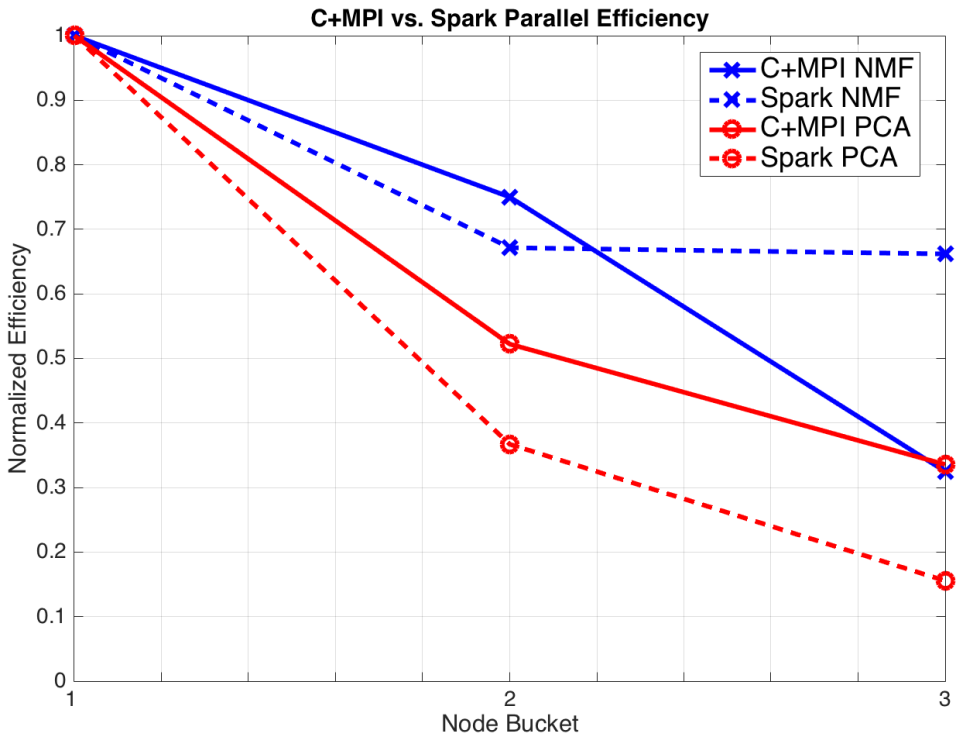


Figure 4: Comparison of parallel efficiency for C+MPI and Spark. The x-axis label "Node Bucket" refers to the node counts. For NMF these are 50, 100, and 300 nodes (left to right) and 100, 300, and 500 nodes for PCA. For both algorithms, efficiency is measured relatively to the performance at the smallest node count.

While implementing and verifying the scientific value of the entire pipeline is out of scope for this report, we have demonstrated the ability to apply our Spark NMF implementation to the TB-sized Daya Bay matrix. Together with feature augmentation, this will enable us to explore more advanced methods in the near future.

6.2 PCA applied to the climate matrices

We compute the PCA using an iterative algorithm whose main kernel is a distributed matrix-vector product. Since matrix-vector products are data parallel, this algorithm fits nicely into the Spark model. Because of the iterative nature of the algorithm, we cache the data matrix in memory to avoid I/O at each iteration.

C+MPI vs. Spark. Figure 5 shows the running time breakdown results for computing a rank-20 PCA decomposition of the Ocean matrix on 100, 300, and 500 nodes using the MPI implementation. Each bin depicts the sum, over all stages, of the time spent in that bin by the average task within a stage.

I/O is a significant bottleneck and does not exhibit the scaling observed for NMF in Figure 3. The I/O time is reduced going from 100 to 300 nodes, but not 300 to 500 nodes because the I/O

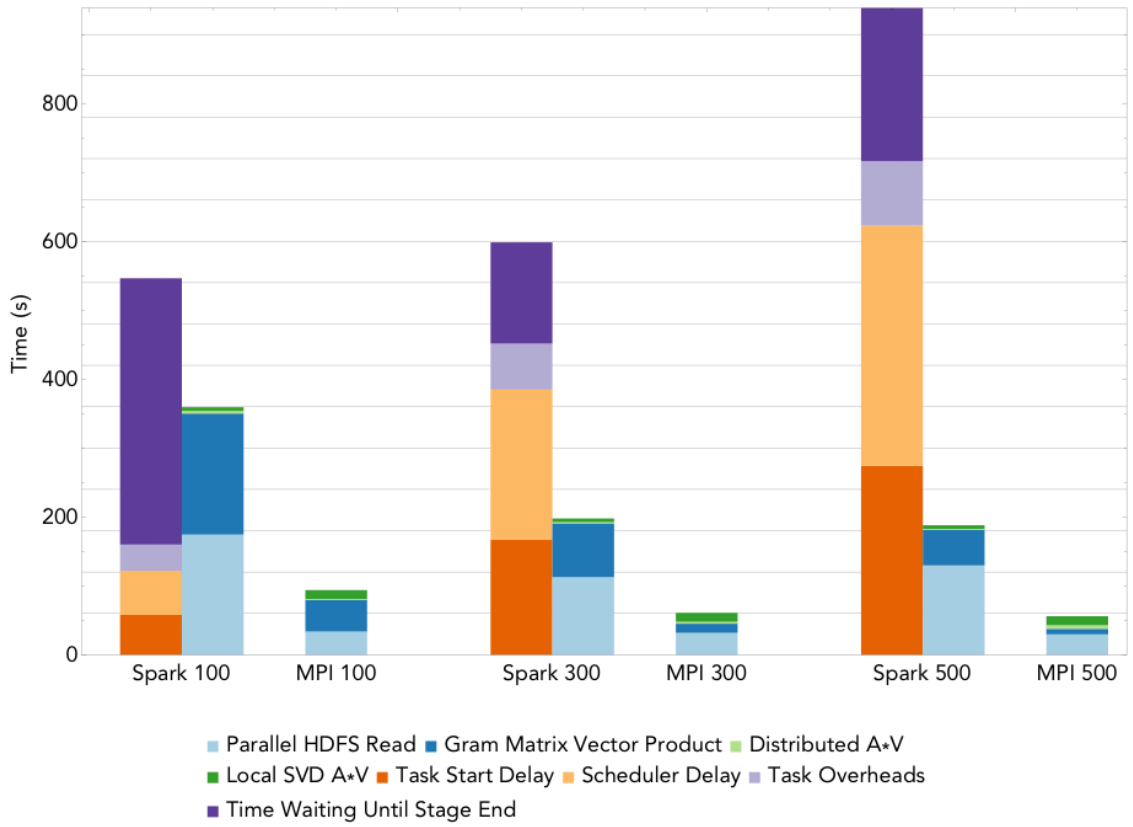


Figure 5: Running time breakdown of PCA on the 2.2TB Ocean matrix at node counts of 100, 300 and 500. Each bin depicts the sum, over all stages, of the time spent in that bin by the average task within a stage.

bandwidth is saturated for the stripe size and number of OSTs used for the Daya Bay and Ocean data sets. The Gram matrix-vector products are a significant portion of the running time but scale linearly with the number of nodes. The matrix-matrix product (AV) does not scale due to a communication bottleneck. The bottleneck is because we compute a rank-20 PCA which makes communicating V expensive. This cost grows with the number processors since it is entirely latency dominated. The final SVD of AV is a sequential bottleneck and does not scale. Unlike NMF the sequential bottleneck in PCA is significant; future implementations should perform this step in parallel.

Figure 5 also shows the scaling and running time breakdown of the Spark PCA implementation for 100, 300, and 500 nodes. The Gram matrix-vector products scale linearly with the number of nodes, however this is outweighed by inefficiencies in Spark. At this scale, Spark is dominated by bottlenecks due to scheduler delays, task overhead and straggler delay times. Task overhead consists of deserializing a task, serializing a result and writing and reading shuffle data. The Spark scheduling delay and task overhead times scale with the number of nodes, due to the centralized scheduler used in Spark. The iterative nature of the PCA algorithm stresses the Spark scheduler since many tasks are launched during each iteration. Under this workload we observed a 10.2×, 14.5×, and 22× performance gap on 100, 300, and 500 nodes, respectively, between Spark and MPI.

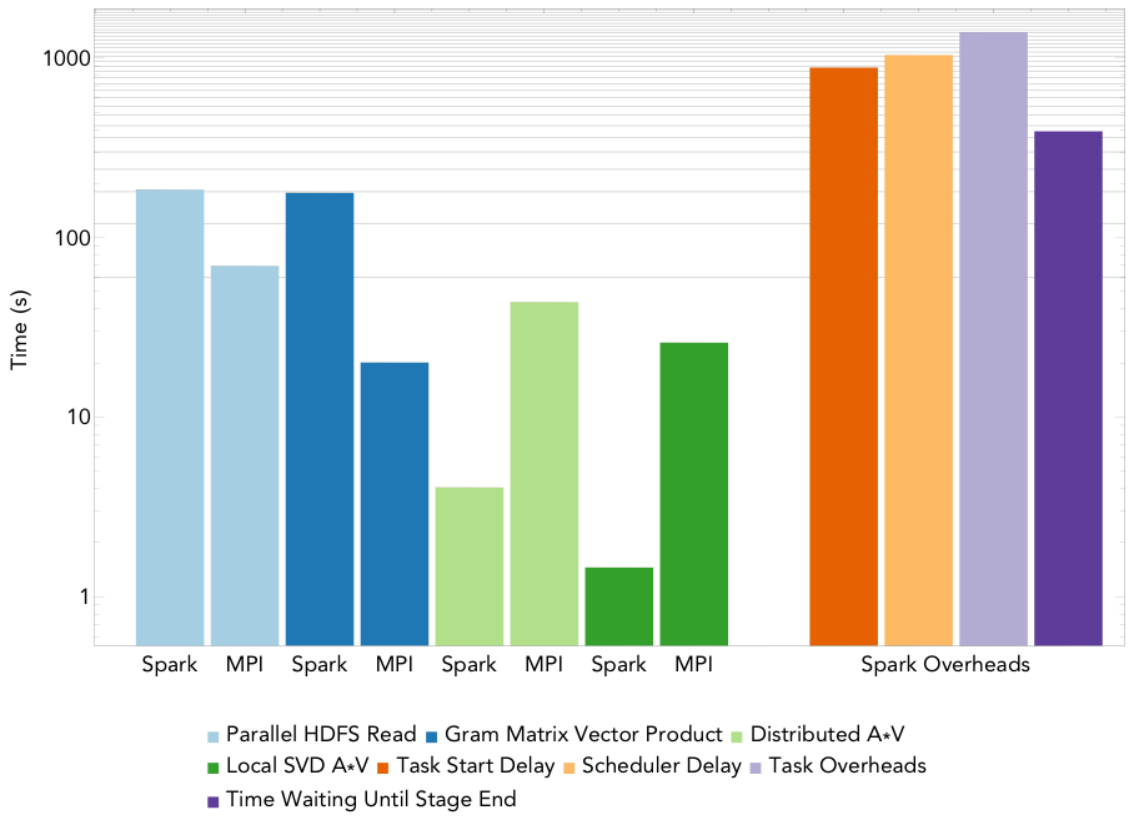


Figure 6: Running time comparison of the Spark and MPI implementations of PCA on the 16TB Atmosphere matrix. Each bin depicts the sum, over all stage, of the time spent in that bin by the average task within a stage.

The disparity between the costs of the AV products and sequential SVDs in MPI and Spark can be attributed to the use of single-threaded BLAS for MPI and multi-threaded BLAS for Spark.

Figure 4 shows the parallel efficiency of MPI PCA and Spark PCA. We observed that the MPI version hits an I/O bottleneck, a communication bottleneck in the AV product and a sequential bottleneck in SVD(AV). All of these are limiting factors and introduce inefficiencies to MPI PCA. Spark PCA is less efficient than MPI PCA due to scheduler delays, task overhead and straggler effects. The scheduler delays are more prominent in PCA than in NMF due to the larger number of tasks. NMF makes a single pass over the data whereas PCA makes many passes over the data and launches many tasks per iteration.

PCA Large-Scale Run. We used all 1600 Cori nodes to compute a rank-20 PCA decomposition of the 16TB Atmosphere matrix. In order to complete this computation in Spark in a reasonable amount of time, we fixed the number of iterations for the EVD of A^TA to 70 iterations. MPI PCA was able to complete this run in 160s. Unfortunately we were unsuccessful at launching Spark on 1600 nodes; after many attempts we reduced the number of nodes to 1522. At this node count, Spark PCA successfully completed the run in 4175s. Figure 6 shows the head-to-head running time comparison for this full-system run; each bin depicts the sum, over all stages, of the time spent within that bin by the average task within a stage. The Gram matrix-vector products are

Algo	Size	# Nodes	Spark Time (s)
CX	1.1 TB	60	1200
		100	784
		300	542

Table 2: Spark CX running times

an order of magnitude more costly in Spark. We noticed that the tree-aggregates were very slow at full-system scale and are the likely cause of the slow Gram matrix-vector products. The AV product and SVD are much faster in Spark than in MPI due to its use of multiple OpenBLAS threads. Finally, we observed that the Spark overheads were an order of magnitude larger than the communication and computation time.

Science Implications. For the 2.2TB Ocean data set, the first two temporal "empirical orthogonal functions" (EOFS)— corresponding to right singular vectors— fully capture the annual cycles. The remaining time series show abrupt changes due to the 1983 El Niño Southern Oscillation (ENSO), and more significantly, the record-breaking ENSO of 1997–98. The intermediate modes contain a complex interplay of various timescales, which is currently under investigation. The spatial EOFs, corresponding to the left singular vectors, show the relative dominance of the Indian Ocean Dipole and the classic warm pool–cold tongue patterns of ENSO at various depths below the ocean surface. Because of the 3D nature of the EOFs, we are able to see that the dynamic near the thermocline is most dominant, rather than that closer to the surface. Further, there are several smaller scale features that have a strong influence at different depths. Work is on-going to understand the nature of these different spatial patterns and the factors that influence their relative dominance.

6.3 CX on the Mass-spec matrix

Much like PCA, the CX decomposition requires a parallel Gramian multiply, a distributed matrix-matrix product and a randomized SVD in order to compute extremal columns of A. CX is applied to the sparse 1.1TB MSI matrix, which is stored in the Parquet format. Table 2 shows the running times and scaling behavior of Spark CX. We found that Spark exhibited good scaling for the range of nodes tested and attained speedups of 1.5× and 2.2× on 100 and 300 nodes, respectively. The corresponding parallel efficiencies are 90% for 100 nodes and 44% for 300 nodes. These results show that the behavior of CX is similar to that of PCA, which is due to the overlap in their linear algebra kernels.

Science Interpretation. The CX decomposition selected ions in three narrow regions of m/z. Among those ions identified as having significant leverage scores are ions at m/z values of 439.0819, 423.0832, and 471.1276, which correspond to neutral losses of CH_2, CH_2O, and a neutral "gain" of H_2O from the 453.0983 ion. These relationships indicate that this set of ions, all identified as having significant leverage scores, are chemically related. That fact indicates that these ions may share a common biological origin, despite having distinct spatial distributions in the plant tissue.

6.4 Summary of Spark vs. C+MPI performance comparison

We have demonstrated that matrix factorizations (which have traditionally been implemented using high-performance parallel libraries) can be implemented on Spark, and that Spark can run on large node counts on HPC platforms. By exploring the performance trade-offs of Spark matrix factorizations and comparing to traditional MPI implementations we have gained insights into the factors affecting Spark's scalability. Table 3 summarizes the wall-clock times of the MPI and Spark implementations of the considered factorizations, and Table 4 summarizes the performance gaps between Spark and MPI. These gaps range between $2 \times -25\times$ when I/O time is included in the comparison and $10 \times -40\times$ when I/O is not included. These gaps are large, but our experiments indicated that Spark I/O scaling is comparable to MPI I/O scaling, and that the computational time scales. The performance gaps are due primarily to scheduler delays, straggler effects, and task overhead times. If these bottlenecks can be alleviated, then Spark can close the performance gap and become a competitive, easy-to-use framework for data analytics on high-performance architectures.

Algo	Size	# Nodes	MPI Time (s)	Spark Time (s)
NMF	1.6 TB	50	66	278
		100	45	207
		300	30	70
PCA	2.2 TB	100	94	934
		300	60	827
		500	56	1160
	16 TB	MPI: 1600 Spark: 1522	160	4175

Table 3: Summary of Spark and MPI running times.

Algo	# Nodes	Gap with I/O	Gap without I/O
NMF	50	4×	21.2×
	100	4.6×	14.9×
	300	2.3×	15.7×
PCA	100	10.2×	12.6×
	300	14.5×	24.7×
	500	22×	39.3×
	MPI: 1600 Spark: 1522	26×	43.8×

Table 4: Summary of the performance gap between the MPI and Spark implementations.

7 Lessons Learned

Throughout the course of these experiments, we have learned a number of lessons pertaining to the behavior of Spark for linear algebra computations in large-scale HPC systems. In this section, we share some of these lessons and conjecture on likely causes.

Spark Scheduling Bottlenecks. The Spark driver creates and sends tasks serially, which can cause bottlenecks at high concurrency. This effect can be quantified by looking at two metrics: Task Start Delay and Scheduler Delay. Task Start Delay measures the time from the start of stage until the task is sent to an executor. Scheduler Delay measures the additional time until the driver receives confirmation that the task has been received and its execution has started. Figure 7 is a plot of a sample of the tasks from one stage of the 16TB Spark PCA run. Note that the ordering of the colored bars within each task line does not correspond to the order they occurred—Spark uses a pipelined execution model, where different portions of a task are interleaved at a fine grain, and reports the total time spent on each activity. We can see that the scheduling bottleneck causes a uniform distribution of start times, with tasks starting as late as 20 seconds after the earliest task. The scheduler delay grows with the start delay, indicating that confirmation messages are queuing up and waiting to be processed at the driver when it finishes sending new tasks.

Algo	Size	Nodes	Partitions	Time (s)	Measured Task Start Delay (s)	Predicted Delay (2000/sec)
PCA	2.2 TB	100	3200	924	411	112
		300	9600	827	332	336
		500	16000	1160	542	560
	16 TB	1522	51200	3718	1779	1792

Table 5: Spark scheduling delays.

Ousterhout et al. [46] showed that these factors limit the Spark scheduler to launching approximately 1500 tasks per second. Their measurements were based on an older version of Spark from 2013, but there have been no significant changes to the scheduler. Our results on Cori are consistent with a similar rate of about 2000 tasks per second. We show the impact of this bottleneck on PCA in Table 5. We expect that the largest negative impact on scaling is caused by the wait required to schedule the tasks in each iteration. The *Measured Task Start Delay* column shows the sum of the largest task start delays in each Spark stage. The *Predicted Delay* column shows the delay predicted by a scheduling rate of 2000 tasks per second over 70 iterations and the listed number of tasks/partitions. We observe that at 300, 500, and 1522 nodes, the task start delay is very close to the predicted value.

This bottleneck represents a limit on the scaling achievable by Spark for highly iterative algorithms. In particular, as the amount of parallelism increases, the minimum number of partitions and tasks also increases. This results in a linearly increasing overhead from the scheduler as we increase parallelism. This delay is further multiplied by the number of iterations. We see the impact of this in the PCA results in Table 5, where the final column represents this fixed overhead and is thus a lower bound on how fast we can execute at the given scale.

Other Significant Spark Overheads. Figures 3 and 5 illustrate that a large block of time is spent in Task Overheads. These overheads consist of the shuffle read and write time, the task deserialization time (executor deserialize time), and result serialization time. During our runs on Cori, most of these overheads are insignificant with the exception of the executor deserialize time, as can be seen in Figure 8. High executor deserialize times are usually attributable to large tasks

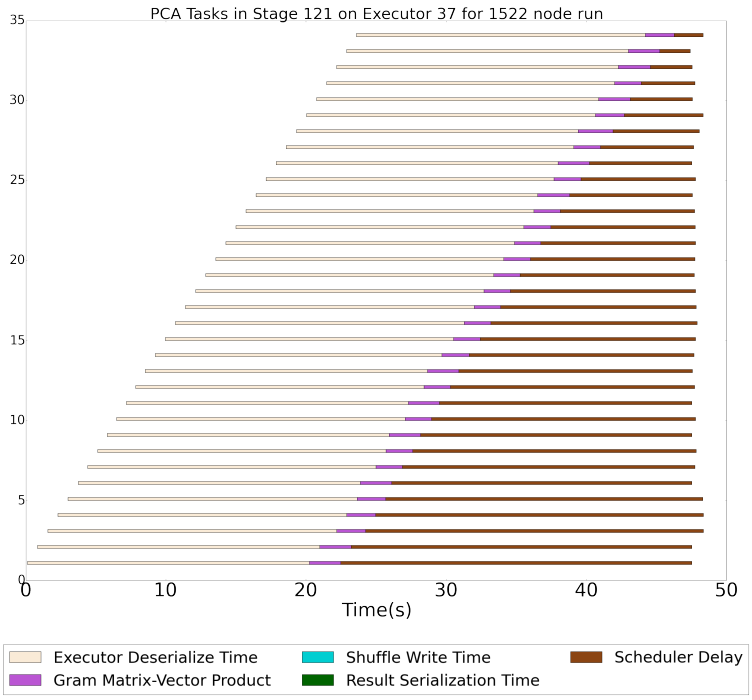


Figure 7: A timeline of tasks on a particular node for a multiply Gramian stage during the 1522 node Spark run.

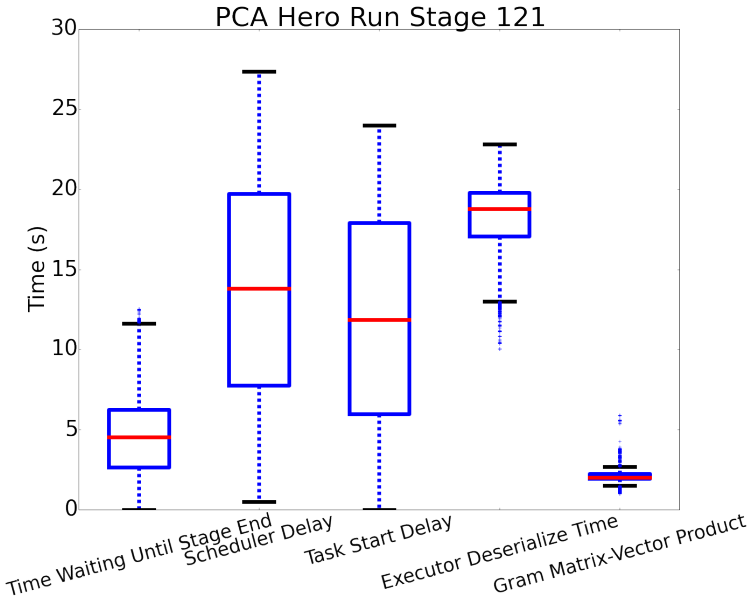


Figure 8: Distribution of various components of all tasks in a multiply Gramian stage in the Spark PCA hero run.

that take a long time to unpack. Also, any time spent in garbage collection during the deserialize phase counts toward the deserialize time.

Spark Variability and Stragglers. The time waiting for stage to end bucket describes the idle time in which a task has finished, but the next stage has not started, so new tasks have not yet been scheduled. The main cause of this idle time is "straggler" tasks, tasks that take a longer than average time to finish and thus hold up the next stage from starting. In Figure 8, we can see there is some variability in the multiply Gramian component of the tasks, but that this is insignificant compared to the Spark sheduler and task start delay, as well as the executor deserialization time. These overheads vary anywhere from less than a second to more than 20 seconds. This variation in task behavior leads to some task slots being left idle for up to 10 seconds.

The bulk-synchronous execution model of Spark creates scaling issues in the presence of stragglers. When a small number of tasks take much longer to complete, many cores waste cycles idling at synchronization barriers. At larger scales, we see increases in both the probability of at least one straggler, as well as the number of underutilized cores waiting at barriers.

During initial testing runs of the Spark PCA algorithm, variations in run time as large as 25% were observed (in our staging runs we had a median run time of 645 seconds, a minium run time of 489 seconds, and a maximum run time of 716 seconds). These variations could not be attributed to any particular spark stage. Sometimes the delay would occur in the multiply Gramian step, other times in the intial data collect stage. This variability is illustrated in the box and whiskers plot. Spark has a "speculation" functionality which aims to mitigate this variability by restarting straggling tasks on a new executor. However, we found that enabling speculation had no appreciable effect on improving the run time, because the overhead to fetch a portion of the RDD from another worker was sufficiently high. This is because requests for RDDs from other workers must wait until the worker finishes its running tasks. This can often result in delays that are as long as the run time of the straggling task.

8 Conclusion

We conclude our study of matrix factorizations at scale with the following take-away messages:

- A range of important matrix factorization algorithms can be implemented in Spark: we have successfully applied NMF, PCA and CX to TB-sized da tasets. We have scaled the codes on 50, 100, 300, 500, and 1600 XC40 nodes. To the best of our knowledge, these are some of the largest scale *scientific data analytics* workloads attempted with Spark.

- Spark and C+MPI head-to-head comparisons of these methods have revealed a number of opportunities for improving Spark performance. The current end-to-end performance gap for our workloads is $2 \times -25\times$; and $10 \times -40\times$ without I/O. At scale, Spark performance overheads associated with scheduling, stragglers, result serialization and task deserialization dominate the runtime by an order of magnitude.

- In order for Spark to leverage existing, high-performance linear algebra libraries, it may be worthwhile to investigate better mechanisms for integrating and interfacing with MPI-based runtimes with Spark. The cost associated with copying data between the runtimes may not be prohibitive.

- Finally, efficient, parallel I/O is critical for Data Analytics at scale. HPC system architectures will need to be balanced to support data-intensive workloads.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The authors thank Doug Jacobsen, Woo-Sun Yang, Tina Declerck and Rebecca Hartman-Baker for assistance with the large scale runs at NERSC. Edgar Solomonik, Penporn Koanantakool and Evangelos Georganas provided helpful comments and suggestions on tuning the MPI codes. We would like to acknowledge Craig Tull, Ben Bowen and Michael Wehner for providing scientific data sets used in the study. We are also grateful to the Daya Bay Collaboration for giving us access to their experimental data.

This research is partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung. This work is supported by Cray, Inc., the Defense Advanced Research Projects Agency XDATA program and DOE Office of Science grants DOE DE-SC0010200 DE-SC0008700, DE-SC0008699. AD is supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1106400. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] Docker hub. https://hub.docker.com/r/lgerhardt/spark-1.5.1-openblas/.

[2] Openblas. http://www.openblas.net/.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1999.

[5] F. Au et al. Measurement of the reactor antineutrino flux and spectrum at daya bay. *Physical Review Letters*, 116(061801), 2016.

[6] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)*, 36(3):13, 2009.

[7] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik. Reconstructing householder vectors from tall-skinny qr. In *Proceedings of the 2014 IEEE 28th*

International Parallel and Distributed Processing Symposium, IPDPS '14, pages 1159–1170, Washington, DC, USA, 2014. IEEE Computer Society.

[8] A. R. Benson, J. D. Lee, B. Rajwa, and D. F. Gleich. Scalable methods for nonnegative matrix factorizations of near-separable tall-and-skinny matrices. In *Advances in Neural Information Processing Systems*, pages 945–953, 2014.

[9] N. Chaimov, A. Malony, S. Canon, K. Ibrahim, C. Iancu, and J. Srinivasan. Scaling spark on hpc systems. In *The 25th International ACM Symposium on High-Performance Parallel and Distributed Computing*, 2016. in publication.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, pages 10–10, 2004.

[11] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[12] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. Relative-error CUR matrix decompositions. *SIAM J. Matrix Analysis Applications*, 30(2):844–881, 2008.

[13] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[14] N. Gillis. The why and how of nonnegative matrix factorization. In J. A. K. Suykens, M. Signoretto, and A. Argyriou, editors, *Regularization, Optimization, Kernels, and Support Vector Machines*, chapter 12. CRC Press, 2014.

[15] N. Gillis, D. Kuang, and H. Park. Hierarchical clustering of hyperspectral images using rank-two nonnegative matrix factorization. *Geoscience and Remote Sensing, IEEE Transactions on*, 53(4):2066–2078, 2015.

[16] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1996.

[17] M. Gu and S. C. Eisenstat. Efficient algorithms for computing a strong rank-revealing qr factorization. *SIAM J. ScI. COMPUT.*, 17(4):848–869, 1996.

[18] D. Guillamet and J. Vitria. Non-negative matrix factorization for face recognition. In *Topics in artificial intelligence*, pages 336–344. Springer, 2002.

[19] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2), 2011.

[20] V. Hernandez, J. Roman, A. Tomas, and V. Vidal. A survey of software for sparse eigenvalue problems. *Universidad Politecnica de Valencia, Valencia, Spain, SLEPc Technical Report STR-6, http://www. grycap. upv. es/slepc*, 2009.

[21] L. Hubert, J. Meulman, and W. Heiser. Two purposes for matrix factorization: A historical appraisal. *SIAM Review*, 42:68–82, 2000.

[22] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 2015.

[23] I. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[24] R. Kannan, G. Ballard, and H. Park. A high-performance parallel algorithm for nonnegative matrix factorization. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 9. ACM, 2016.

[25] J. Kim, Y. He, and H. Park. Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework. *Journal of Global Optimization*, 58(2):285–319, 2014.

[26] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov. Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc. *SIAM Journal on Scientific Computing*, 29(5):2224–2239, 2007.

[27] Y. Koren, R. Bell, C. Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[28] A. Kumar, V. Sindhwani, and P. Kambadur. Fast conical hull algorithms for near-separable non-negative matrix factorization. In *ICML*, 2013.

[29] R. M. Larsen. Lanczos bidiagonalization with partial reorthogonalization. *DAIMI Report Series*, 27(537), 1998.

[30] C. L. Lawson and R. J. Hanson. *Solving least squares problems*. Classics in applied mathematics. SIAM, Philadelphia (Pa.), 1995. SIAM : Society of industrial and applied mathematics.

[31] D. Lee and H. Seung. Algorithms for non-negative matrix factorization. In *NIPS*, 2001.

[32] J. S. Lee, D. D. Lee, S. Choi, K. S. Park, and D. S. Lee. Non-negative matrix factorization of dynamic images in nuclear medicine. In *Nuclear Science Symposium Conference Record, 2001 IEEE*, volume 4, pages 2027–2030 vol.4, 2001.

[33] R. B. Lehoucq and D. C. Sorensen. Deflation techniques for an implicitly restarted arnoldi iteration. *SIAM Journal on Matrix Analysis and Applications*, 17(4):789–821, 1996.

[34] R. B. Lehoucq, D. C. Sorensen, and C. Yang. Arpack users guide: Solution of large scale eigenvalue problems by implicitly restarted arnoldi methods., 1997.

[35] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 625–636, New York, NY, USA, 2013. ACM.

[36] R. Liao, Y. Zhang, J. Guan, and S. Zhou. CloudNMF: A MapReduce Implementation of Nonnegative Matrix Factorization for Large-scale Biological Datasets . *Genomics, Proteomics & Bioinformatics*, 12(1):48 – 51, 2014.

[37] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, pages 681–690. ACM, 2010.

[38] J. Liu, E. Racah, Q. Koziol, R. S. Canon, A. Gittens, L. Gerhardt, S. Byna, M. F. Ringenberg, and Prabhat. H5spark: Bridging the I/O gap between spark and scientific data formats on hpc systems. In *Cray User Group*, 2016.

[39] M. W. Mahoney. *Randomized algorithms for matrices and data*. Foundations and Trends in Machine Learning. NOW Publishers, Boston, 2011.

[40] M. W. Mahoney and P. Drineas. CUR matrix decompositions for improved data analysis. *Proc. Natl. Acad. Sci. USA*, 106:697–702, 2009.

[41] A. Mahout. Apache Mahout: Scalable machine learning and data mining. `http://mahout.apache.org/`. Accessed: 2016-04-10.

[42] P.-G. Martinsson, V. Rohklin, and M. Tygert. A Randomized Algorithm for the Approximation of Matrices. *Technical Report*, 2006.

[43] P.-G. Martinsson, V. Rohklin, and M. Tygert. A randomized algorithm for the decomposition of matrices. *Appl. Comput. Harmon. Anal.*, 30:47–68, 2011.

[44] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[45] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[46] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[47] P. Paatero. Least squares formulation of robust non-negative factor analysis. *Chemometrics and Intelligent Laboratory Systems*, 37(1):23 – 35, 1997.

[48] P. Paschou, E. Ziv, E. G. Burchard, S. Choudhry, W. Rodriguez-Cintron, M. W. Mahoney, and P. Drineas. PCA-correlated SNPs for structure identification in worldwide human populations. *PLoS Genetics*, 3:1672–1686, 2007.

[49] V. P. Pauca, J. Piper, and R. J. Plemmons. Nonnegative matrix factorization for spectral data analysis. *Linear algebra and its applications*, 416(1):29–47, 2006.

[50] S. Saha, S. Moorthi, H.-L. Pan, X. Wu, J. Wang, S. Nadiga, P. Tripp, R. Kistler, J. Woollen, D. Behringer, H. Liu, D. Stokes, R. Grumbine, G. Gayno, J. Wang, Y.-T. Hou, H.-Y. Chuang, H.-M. H. Juang, J. Sela, M. Iredell, R. Treadon, D. Kleist, P. V. Delst, D. Keyser, J. Derber, M. Ek, J. Meng, H. Wei, R. Yang, S. Lord, H. V. D. Dool, A. Kumar, W. Wang, C. Long,

M. Chelliah, Y. Xue, B. Huang, J.-K. Schemm, W. Ebisuzaki, R. Lin, P. Xie, M. Chen, S. Zhou, W. Higgins, C.-Z. Zou, Q. Liu, Y. Chen, Y. Han, L. Cucurull, R. W. Reynolds, G. Rutledge, and M. Goldberg. The ncep climate forecast system reanalysis. *Bulletin of the American Meteorological Society*, 91(8):1015–1057, 2010.

[51] E. Solomonik. *Provably efficient algorithms for numerical tensor algebra*. University of California, Berkeley, 2014.

[52] R. Thakur and W. D. Gropp. Improving the performance of collective operations in mpich. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 257–267. Springer, 2003.

[53] M. F. Wehner, K. A. Reed, F. Li, Prabhat, J. Bacmeister, C.-T. Chen, C. Paciorek, P. J. Gleckler, K. R. Sperber, W. D. Collins, A. Gettelman, and C. Jablonowski. The effect of horizontal resolution on simulation quality in the community atmospheric model, cam5.1. *Journal of Advances in Modeling Earth Systems*, 6(4):980–997, 2014.

[54] J. Yang, O. Rübel, Prabhat, M. W. Mahoney, and B. P. Bowen. Identifying important ions and positions in mass spectrometry imaging data using CUR matrix decompositions. *Analytical Chemistry*, 87(9):4658–4666, 2015.

[55] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon. The Magellan Report on Cloud Computing for Science. Technical report, U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), 2011.

[56] C.-W. Yip, M. W. Mahoney, A. S. Szalay, I. Csabai, T. Budavari, R. F. G. Wyse, and L. Dobos. Objective identification of informative wavelength regions in galaxy spectra. *The Astronomical Journal*, 147(110):15pp, 2014.

[57] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. *Proceedings of the VLDB Endowment*, 7(11):975–986, 2014.

[58] R. B. Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia. Matrix Computations and Optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016.

[59] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud'10: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.