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

Alex Gittens*, Aditya Devarakonda†, Evan Racah‡, Michael Ringenburg§, Lisa Gerhardt‡,
Jey Kottalam†, Jialin Liu‡, Kristyn Maschhoff§, Shane Canon‡, Jatin Chhugani¶, Pramod Sharma§,
Jiyan Yang‖, James Demmel**, Jim Harrell§, Venkat Krishnamurthy§, Michael W. Mahoney* and Prabhat‡
*ICSI and Department of Statistics, UC Berkeley
†EECS, UC Berkeley
‡NERSC, Lawrence Berkeley National Laboratory
§Cray, Inc.
¶Hiperform Consulting LLC
‖ICME, Stanford University
**EECS and Math, UC Berkeley

***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 1.6TB particle physics, 2.2TB and 16TB climate modeling and 1.1TB bioimaging data. 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.**

***Keywords*-matrix factorization; linear algebra; Apache Spark; PCA; NMF**

I. Introduction

Modern experimental devices and scientific simulations produce massive amounts of complex data: in high energy physics, the LHC project produces PBs of data; the climate science community relies upon access to the CMIP-5 archive, which is several PBs in size; the multi-modal imagers used in biosciences can acquire 100GBs-TBs of data. Several scientific domains are currently rate-limited by access to productive and performant data analytics tools that operate on data of these sizes.

We have seen recent substantial progress in the adoption of Big Data software frameworks such as Hadoop/MapReduce [1] and Spark [2]. Ideally, the scientific data analysis and high performance computing (HPC) communities would leverage the momentum behind Hadoop and Spark. Unfortunately, these frameworks have been developed for industrial applications and commodity hardware, and the performance of such frameworks at scale on conventional HPC hardware has not been investigated extensively. For matrix factorizations in particular, there is a gap between the performance of well-established libraries (ScaLAPACK, LAPACK, BLAS, PLASMA, MAGMA, etc. [3, 4]) and the tools available in Spark. Our work takes on the important task of testing nontrivial linear algebra and matrix factorization computations in Spark using 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:

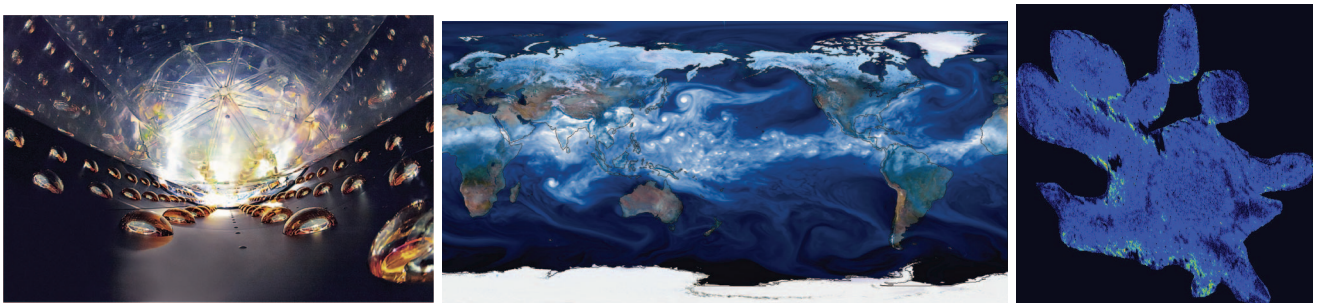
- We develop parallel versions of three leading matrix factorizations (PCA, NMF, CX) in Spark and C+MPI; and we apply them to several TB-sized scientific data sets. To ensure that our comparison of Spark to MPI is fair, we implement the same algorithms in Spark and MPI, drawing on a common set of numerical linear algebra libaries for which Spark bindings are readily available (Blas, Lapack, and Arpack).
- 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: by identifying the causes of the slow-downs in 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.
- We comment on opportunities for future work in Spark to better address large scale scientific data analytics on HPC platforms.

II. 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. These data sets are summarized in Table I.

The Daya Bay Neutrino Experiment: The Daya Bay Neutrino Experiment (Figure 1a) detects antineutrinos produced by the Ling Ao and Daya Bay nuclear power plants

978-1-4673-9005-7/16/$31.00 ©2016 IEEE



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

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

Table I: 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

and uses them to measure theta-13, a fundamental constant that helps describe the flavor oscillation of neutrinos. We computed an NMF factorization on a sparse 1.6TB matrix consisting of measurements from Daya Bay's photodetector arrays. The analytics problem that we hope to tackle with NMF is that of finding characteristic patterns or signatures corresponding to various particle types.

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. The most widely used tool for extracting important patterns from the measurements of atmospheric and oceanic variables is the Empirical Orthogonal Function (EOF) technique. Mathematically, EOFs are exactly PCA decompositions. Traditionally, the lack of scalable analytics methods and tools has prevented the community from analyzing full 3D fields; typical analysis is performed only on 2D spatial averages or slices. We compute the EOFs of a dense 2.2TB matrix comprising global ocean temperature data collected over 30 years [5], and of a dense 16TB matrix comprising atmospheric humidity measurements collected over 28 years [6] (Figure 1b). A better understanding of the dynamics of large-scale modes of variability in the ocean and atmosphere may be extracted from the 3D EOFs we compute.

Mass-Spectrometry Imaging: Mass spectrometry measures ionic spectra derived from the molecules present in a biological sample. We analyze one of the largest (1TB-sized) mass-spec imaging data sets in the field, obtained from a sample of a plant from the *Peltatum* species (Figure 1c). The MSI measurements are formed into a sparse matrix the sheer size of which has previously made complex analytics intractable. CX decompositions select a small numbers of columns (corresponding to ions) in the original data that reliably explain a large portion of the variation in the data.

III. Methods

Given an $m \times n$ data matrix A, low-rank matrix factorization methods aim to find two or more smaller matrices Y and Z such that

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

Depending on the particular application, various low-rank factorization techniques are of interest. Popular choices include the singular value decomposition [7], principal component analysis [8], rank-revealing QR factorization [9], nonnegative matrix factorization (NMF) [10], and CX/CUR decompositions [11]. 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. Throughout, we assume the data matrix A has size $m \times n$ and can be well approximated by a rank r approximation, with $r \ll n \ll m$; this "tall-skinny", highly rectangular setting is common in practice.

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 [12], BLOPEX [13], and ANASAZI [14], among others. We refer the interested reader to [15] 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 [16].

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

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 $(V_k, \_) = \text{IRAM}(\text{MULTIPLYGRAMIAN}(A, \cdot), k)$.
2: Let $Y = \text{MULTIPLY}(A, V_k)$.
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**

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 $(\mathcal{K}, H) = \text{XRAY}(R, k)$.
3: Let $W = A(:, \mathcal{K})$.

descent approach [17, 18, 19]. We mention a few of the high-visibility efforts in this space. The earlier work [20] developed and studied a distributed implementation of the NMF for general matrices under the Hadoop framework, while [21] 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 implementations of basic linear algebra routines [22]; 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 [23], but such factorizations are not appropriate for scientific applications which require high precision.

Our contribution is the provision of, for the first time, a detailed investigation of the scalability of three low-rank factorizations—PCA, NMF, and CX—using the linear algebra tools and bindings provided in Spark's baseline MLLIB [22] and ML-MATRIX [24] libraries.

Principal Components Analysis: Throughout, we use the term principal component analysis (PCA) of a centered matrix A (*i.e.*, one whose columns are zero-mean) to refer to the rank-k approximation given by $A_k = U_k \Sigma_k V_k^T$, where the columns of U_k and V_k are the top k *left and right singular vectors* of A, 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: a series of matrix-vector products against $A^T A$ (MULTIPLYGRAMIAN) are used to extract V_k by applying the implicitly restarted Arnoldi method (IRAM) [25], then the remaining factors U_k and Σ_k are computed by taking the SVD of AV_k. Here QR and SVD compute the "thin" versions of the QR and SVD decompositions [7]. (Algorithm 1 calls MULTIPLYGRAMIAN, which is summarized in Algorithm 2).

Nonnegative Matrix Factorization: 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 [26]. NMF has found applications, among other places, in medical imaging [27], facial recognition [28], chemometrics [29], hyperspectral imaging [30], and astronomy [31].

To find an NMF decomposition, we seek matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ such that the approximation error $\|A - WH\|_F$ is small and W and H are entrywise nonnegative. We adopt the one-pass algorithm of [21] to solve this problem. This approach assumes 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 QR factorization is used to compute the R factor of A; in step 2, the XRAY algorithm of [32] 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.

CX decompositions: CX decompositions are low-rank matrix decompositions that are expressed in terms of a small number of actual columns of A. They have been used in scientific applications where interpretability is paramount, including genetics [33], astronomy [34], and mass spectrometry imaging [35].

To find a CX decomposition, we seek matrices $C \in \mathbb{R}^{m \times k}$ and $X \in \mathbb{R}^{k \times n}$ such that the approximation error $\|A - CX\|_F$ is small and C contains k actual columns of A. The randomized algorithm of [36] generates a C with low approximation error. Details are given in Algorithm 4: the first nine steps of the algorithm approximate V_k, and the next step computes measures of the extent to which the columns of A influenced V_k; the remaining two steps uses these importance measures to sample from the columns of A to form C. The matrix X is implicitly determined, and for our purposes does not need to be computed.

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: $B \leftarrow \text{MULTIPLYGRAMIAN}(A, B)$
4: $(B, \_) \leftarrow \text{QR}(B)$
5: **end for**
6: Let Q be the first k columns of B.
7: Let $Y = \text{MULTIPLY}(A, Q)$.
8: Compute $(U, \Sigma, \tilde{V}^T) = \text{SVD}(Y)$.
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 ℓ columns from A in i.i.d. trials, using the importance sampling distribution $\{p_i\}_{i=1}^{n}$.

IV. 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.

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 stores the block of the rows of A denoted by $A_{(i)}$, then $A^T A B = \sum_{i=1}^{\ell} A_{(i)}^T A_{(i)} B$. 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 MLLIB 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.

V. 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.

We use Spark's Standalone Cluster Manager to run the Spark cluster in an encapsulated Shifter image. Shifter is a framework that delivers docker-like functionality to HPC [37]. 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.

H5Spark: Loading HDF5 data natively into Spark: The Daya Bay and climate data sets are stored in HDF5. We used the H5Spark [38] package to read this data into an RDD. 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.

Spark Tuning Parameters: 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) [39, 40] factorization implemented as part of the Communication-Avoiding Dense Matrix Computations (CANDMC) library [41] 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 [42] in the XRay algorithm. PCA requires EVD, SVD, matrix-vector products, and matrix-matrix products. We use arpack-ng [43] 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: When reporting the overheads due to Spark's communication and synchronization costs, we group them into the following bins:

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

VI. Results

A. 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.

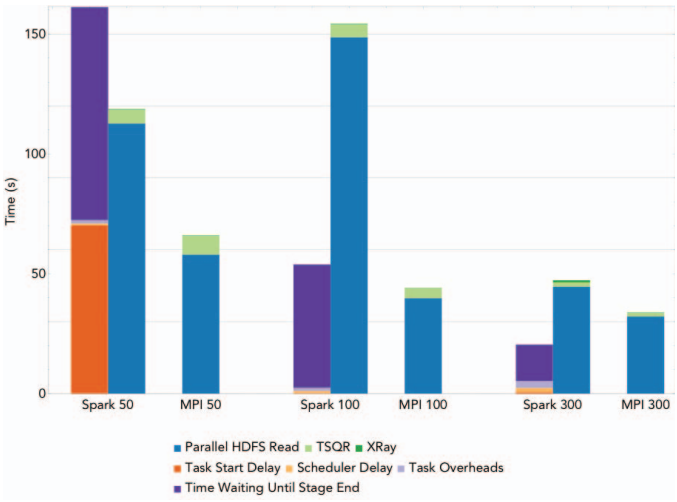


Figure 2: Running time breakdown when using NMF to compute a rank 10 approximation to the 1.6TB Daya Bay matrix at node counts of 50, 100, and 300. Each bin depicts the sum, over all stages, of the time spent in that bin by the average task within a stage. The 50 node run uses double the number of partitions as physical cores because due to out-of-memory errors using fewer partitions– this results in a large task start delay.

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 2 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 2 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 2 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 is spent in I/O and Spark overheads, with a small amount of time spent in TSQR and XRAY. Figure 2 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 $4\times$, $4.6\times$, and $2.3\times$ performance gap on 50, 100, and 300 nodes, respectively, between Spark and MPI. There is some disparity between the TSQR costs but this can be attributed to the lack of granularity in our Spark profiling, in particular the communication time due to Spark's lazy evaluation. Therefore, it is likely that the communication overhead is included in the other overhead costs whereas the MPI algorithm reports the combined communication and computation time.

Figure 3 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.

B. 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 4 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.

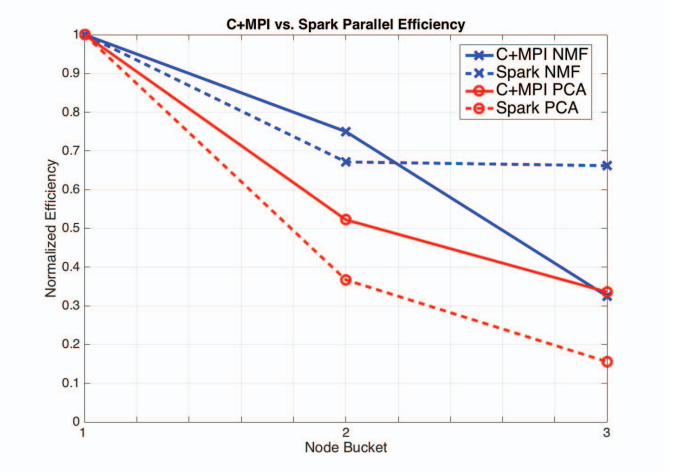


Figure 3: Comparison of parallel efficiency for C+MPI and Spark. The x-axis label "Node Bucket" refers to the node counts. For NMF these are 50, 100, and 300 nodes (left to right) and 100, 300, and 500 nodes for PCA. For both algorithms, efficiency is measured relatively to the performance at the smallest node count.

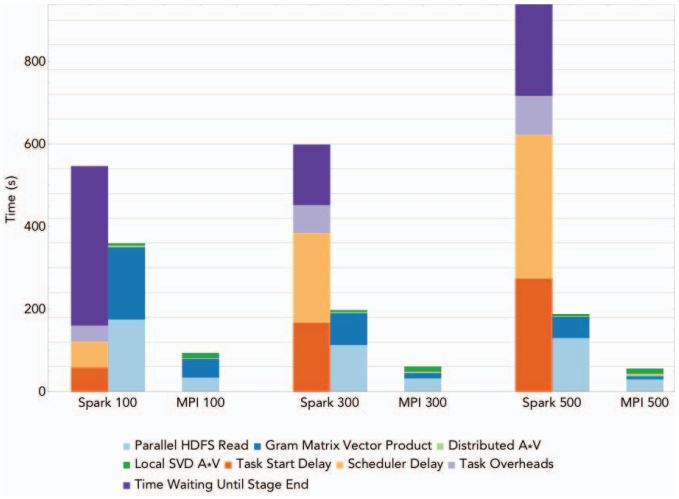


Figure 4: Running time breakdown of PCA on the 2.2TB Ocean matrix at node counts of 100, 300 and 500. Each bin depicts the sum, over all stages, of the time spent in that bin by the average task within a stage.

I/O is a significant bottleneck and does not exhibit the scaling observed for NMF in Figure 2. The I/O time is reduced going from 100 to 300 nodes, but not 300 to 500 nodes because the I/O 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

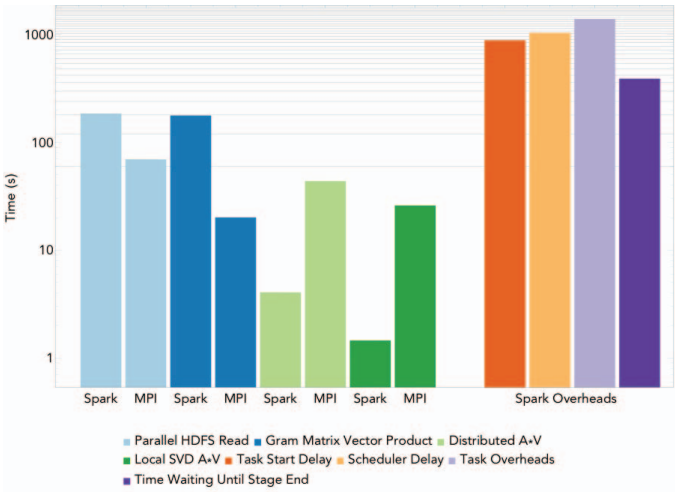


Figure 5: Running time comparison of the Spark and MPI implementations of PCA on the 16TB Atmosphere matrix. Each bin depicts the sum, over all stage, of the time spent in that bin by the average task within a stage.

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 4 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. The disparity between the costs of the AV products and sequential SVDs in MPI and Spark can be attributed to the lack of granularity in our Spark profiling, in particular the communication time due to Spark's lazy evaluation. Therefore, it is likely that the communication overhead is included in the other overhead costs whereas the MPI algorithm reports the combined communication and computation time.

Figure 3 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.

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

Table II: Spark CX running times

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 5 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 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 limited profiling granularity. Finally, we observed that the Spark overheads were an order of magnitude larger than the communication and computation time.

C. CX on the Mass-spec matrix

Much like PCA, the CX decomposition requires parallel Gramian multiplies, and distributed matrix-matrix products in order to compute extremal columns of A. CX was applied to the 1.1TB MSI matrix; Table II 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 $1.5\times$ and $2.2\times$ 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 expected due to the overlap in their linear algebra kernels.

D. Summary of Spark vs. C+MPI performance comparison

Table III summarizes the wall-clock times of the MPI and Spark implementations of the considered factorizations, and Table IV 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 III: 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 IV: Summary of the performance gap between the MPI and Spark implementations.

VII. 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.

Scheduling Bottlenecks: The Spark driver creates and sends tasks serially, which causes bottlenecks at high concurrency. This effect can be quantified by looking at Task Start Delay and Scheduler Delay. Figure 6 gives a breakdown of Spark's overheads for one stage of the 16TB climate PCA run. It is clear that the time spent waiting at scheduling bottlenecks is orders of magnitude higher than the time spent in actual computation. These significant per-stage cost limits the scaling achievable by Spark for highly iterative algorithms.

Spark Variability and Stragglers.: The time waiting for stage to end bucket in Figure 6 describes the idle time for a single stage in which a task has finished, but is waiting for other tasks to finish. The main cause of this idle time is what we call "straggler effect", where some tasks take a longer than average time to finish and thus hold up the next stage from starting. In Figure 6, we can see there is some variability in the MULTIPLYGRAMIAN component of the tasks, but this is insignificant compared

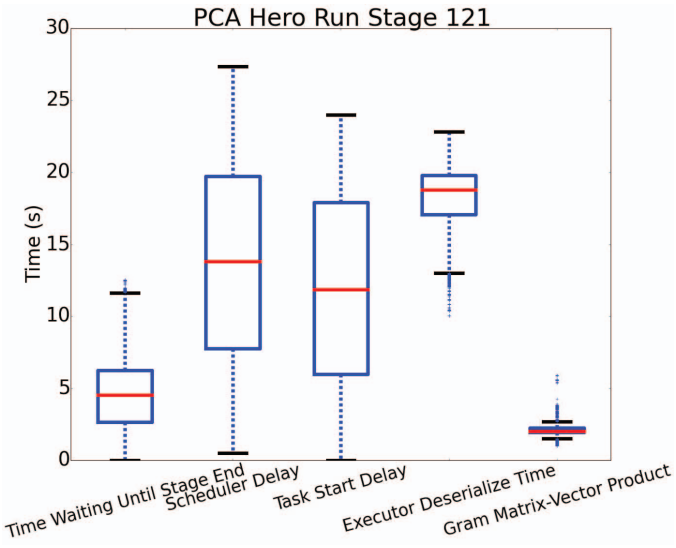


Figure 6: Distribution of various components of all tasks in a MULTIPLYGRAMIAN stage in the Spark PCA hero run.

to the remaining overheads. The straggler time may seem insignificant, however, Figure 6 shows the statistics for a single stage. When summed over all stages (*i.e.*, all PCA iterations) the straggler effect does a become significant overhead at O(100) seconds (see Figure 5).

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 minimum run time of 489 seconds, and a maximum run time of 716 seconds). Spark has a "speculation" functionality which aims to mitigate this variability by restarting straggling tasks on a new executor. 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.

VIII. Conclusion

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

- 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.

We would like to thank Doug Jacobsen, Woo-Sun Yang, Tina Declerck and Rebecca Hartman-Baker for assistance with the large scale runs at NERSC. We thank Edgar Solomonik, Penporn Koanantakool and Evangelos Georganas for helpful comments and suggestions on tuning the MPI codes. We would like to acknowledge Craig Tull, Ben Bowen and Michael Wehner for providing the scientific data sets used in the study. The authors gratefully acknowledge the Daya Bay Collaboration for 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] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. 6th Conf. Symp. on Operating Systems Design and Implementation*, 2004, pp. 10–10.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. 2nd USENIX Conf. on Hot Topics in Cloud Computing*, 2010.

[3] 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*. Philadelphia: SIAM, 1999.

[4] 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," *J. Phys. Conf. Series*, vol. 180, no. 1, 2009.

[5] S. Saha *et al.*, "The NCEP Climate Forecast System Reanalysis," *Bulletin of the American Meteorological Society*, vol. 91, no. 8, pp. 1015–1057, 2010.

[6] M. F. Wehner *et al.*, "The Effect of Horizontal Resolution on Simulation Quality in the Community Atmospheric Model, CAM5.1," *J. Adv. Model. Earth Syst.*, vol. 6, no. 4, pp. 980–997, 2014.

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

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

[9] M. Gu and S. C. Eisenstat, "Efficient Algorithms for Computing a Strong Rank-revealing QR Factorization," *SIAM J. Sci. Comput.*, vol. 17, no. 4, pp. 848–869, 1996.

[10] D. D. Lee and H. S. Seung, "Algorithms for Non-negative Matrix Factorization," in *Adv. Neural Inform. Process. Syst.*, 2001, pp. 556–562.

[11] M. W. Mahoney and P. Drineas, "CUR Matrix Decompositions for Improved Data Analysis," *Proc. Natl. Acad. Sci. USA*, vol. 106, pp. 697–702, 2009.

[12] R. M. Larsen, "Lanczos Bidiagonalization with Partial Reorthogonalization," *DAIMI Report Series*, vol. 27, no. 537, 1998.

[13] A. V. Knyazev, M. E. Argentati, I. Lashuk, and E. E. Ovtchinnikov, "Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) in Hypre and PETSc," *SIAM J. Sci. Comput.*, vol. 29, no. 5, pp. 2224–2239, 2007.

[14] 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 Trans. Math. Softw.*, vol. 36, no. 3, p. 13, 2009.

[15] V. Hernandez, J. Roman, A. Tomas, and V. Vidal, "A Survey of Software for Sparse Eigenvalue Problems," Universidad Politecnica de Valencia, Tech. Rep., 2009.

[16] R. Kannan, G. Ballard, and H. Park, "A High-performance Parallel Algorithm for Nonnegative Matrix Factorization," in *Proc. 21st ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. ACM, 2016, p. 9.

[17] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale Matrix Factorization with Distributed

Stochastic Gradient Descent," in *Proc. 17th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. ACM, 2011, pp. 69–77.

[18] 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," *Proc. VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.

[19] Y. Koren *et al.*, "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[20] 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 *Proc. 19th Int. Conf. on World Wide Web*. ACM, 2010, pp. 681–690.

[21] 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 *Adv. Neural Inform. Process. Syst.*, 2014, pp. 945–953.

[22] X. Meng *et al.*, "MLlib: Machine Learning in Apache Spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[23] B. Li, S. Tata, and Y. Sismanis, "Sparkler: Supporting large-scale matrix factorization," in *Proc. 16th Int. Conf. on Extending Database Technology*. New York, NY, USA: ACM, 2013, pp. 625–636.

[24] R. Zadeh *et al.*, "Matrix Computations and Optimization in Apache Spark," in *Proc. 22nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. ACM, 2016.

[25] R. B. Lehoucq and D. C. Sorensen, "Deflation Techniques for an Implicitly Restarted Arnoldi Iteration," *SIAM J. Mat. Anal. App.*, vol. 17, no. 4, pp. 789–821, 1996.

[26] N. Gillis, "The Why and How of Nonnegative Matrix Factorization," in *Regularization, Optimization, Kernels, and Support Vector Machines*. CRC Press, 2014, ch. 12.

[27] J. S. Lee *et al.*, "Non-negative Matrix Factorization of Dynamic Images in Nuclear Medicine," in *2001 IEEE Nucl. Sci. Symp. Conf. Rec.*, vol. 4, 2001, pp. 2027–2030.

[28] D. Guillamet and J. Vitria, "Non-negative Matrix Factorization for Face Recognition," in *Topics in Artificial Intelligence*. Springer, 2002, pp. 336–344.

[29] P. Paatero, "Least Squares Formulation of Robust Non-negative Factor Analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 37, no. 1, pp. 23 – 35, 1997.

[30] N. Gillis, D. Kuang, and H. Park, "Hierarchical Clustering of Hyperspectral Images Using Rank-two Nonnegative Matrix Factorization," *IEEE Trans. Geosci. Remote Sens.*, vol. 53, no. 4, pp. 2066–2078, 2015.

[31] V. P. Pauca, J. Piper, and R. J. Plemmons, "Nonnegative Matrix Factorization for Spectral Data Analysis," *Linear Algebra and its Applications*, vol. 416, no. 1, pp. 29–47, 2006.

[32] A. Kumar, V. Sindhwani, and P. Kambadur, "Fast Conical Hull Algorithms for Near-separable Non-negative Matrix Factorization," in *Proc. 30th Int. Conf. Mach. Learning*, 2013, pp. 231–239.

[33] P. Paschou and E. Ziv and E. G. Burchard and S. Choudhry and W. Rodriguez-Cintron and M. W. Mahoney and P. Drineas, "PCA-Correlated SNPs for Structure Identification in Worldwide Human Populations," *PLoS Genetics*, vol. 3, pp. 1672–1686, 2007.

[34] 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," *Ast. J.*, vol. 147, no. 110, p. 15pp, 2014.

[35] J. Yang *et al.*, "Identifying Important Ions and Positions in Mass Spectrometry Imaging Data using CUR Matrix Decompositions," *Anal. Chem.*, vol. 87, no. 9, pp. 4658–4666, 2015.

[36] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, "Relative-Error CUR Matrix Decompositions," *SIAM J. Mat. Anal. App.*, vol. 30, no. 2, pp. 844–881, 2008.

[37] D. Jacobsen and S. Canon, "Contain This, Unleashing Docker for HPC," *Proc. Cray User Group*, 2015.

[38] J. Liu *et al.*, "H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems," in *Cray User Group*, 2016.

[39] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik, "Reconstructing Householder Vectors from Tall-Skinny QR," in *Proc. IEEE 28th Int. Parallel Distrib. Proc. Symp.* IEEE, 2014, pp. 1159–1170.

[40] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal Parallel and Sequential QR and LU Factorizations," *SIAM J. Sci. Comput.*, vol. 34, no. 1, pp. 206–239, 2012.

[41] E. Solomonik, "Provably Efficient Algorithms for Numerical Tensor Algebra," University of California, Berkeley, Tech. Rep., 2014.

[42] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. SIAM, 1995.

[43] R. B. Lehoucq, D. C. Sorensen, and C. Yang, "ARPACK Users Guide: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods," 1997.