A multi-platform evaluation of low-rank matrix factorizations in Spark

Alex Gittens*, Jey Kottalam†, Jiyan Yang‡, Michael F. Ringenburg§, Jatin Chhugani¶, Evan Racah‖, Mohitdeep Singh**, Yushu Yao‖, Curt Fischer††, Oliver Ruebel‡‡, Benjamin Bowen††, Norman Lewis[x], Michael W. Mahoney*, Venkat Krishnamurthy§, Prabhat‖

*ICSI and Department of Statistics, UC Berkeley; Emails: gittens@icsi.berkeley.edu, mmahoney@stat.berkeley.edu

†Berkeley Institute for Data Science and EECS, UC Berkeley; Email: jey@berkeley.edu

‡ICME, Stanford University; Email: jiyan@stanford.edu

§Cray Inc.; Emails: mikeri@cray.com, venkat@cray.com

¶HiPerform Inc.; Email: jatinch@gmail.com

‖NERSC Division, Lawrence Berkeley National Laboratory; Emails: eracah@lbl.gov, yyao@lbl.gov, prabhat@lbl.gov

**Georgia Institute of Technology; Email: msingh84@gatech.edu

††Life Sciences Division, Lawrence Berkeley National Laboratory; Emails: crfischer@lbl.gov, bpbowen@lbl.gov

‡‡Computational Research Division, Lawrence Berkeley National Laboratory; Email: oruebel@lbl.gov

[x]Institute of Biological Chemistry, Washington State University; Email: lewisn@wsu.edu

***Abstract*—We investigate the performance, scalability, and applicability of low-rank matrix approximation algorithms, including randomized PCA and randomized CX/CUR low-rank matrix factorizations, on a 1 TB mass spectrometry imaging (MSI) dataset, using Apache Spark on an Amazon EC2 cluster, a Cray® XC40™ system, and an experimental Cray cluster. While these low-rank matrix computations are popular in small- to medium-scale machine learning and scientific data analysis, computing them provides a much more powerful "stress test" of linear algebra algorithms in large-scale distributed analytics frameworks than is provided by, e.g., low-precision PageRank computations. In addition, scientific applications such as MSI data analysis provides a very different use case than is provided by typical commercial workloads. We implemented these algorithms in Scala using the Apache Spark high-level cluster computing framework. We obtained competitive performance on all three platforms; we were able to process the 1TB size dataset in under 30 minutes with 960 cores on all systems, with the fastest times obtained on the experimental Cray cluster. We report these results, and conclude broader implications for hardware and software issues for supporting data-centric workloads in parallel and distributed environments.**

***Keywords*-matrix factorization; data analytics; high performance computing**

I. Introduction

Matrix algorithms are increasingly important in many large-scale data analysis applications. Essentially, the reason is that matrices (i.e., sets of vectors in Euclidean spaces) provide a convenient mathematical structure with which to model data arising in a broad range of applications

In particular, the low-rank approximation to a data matrix A that is provided by performing a truncated SVD (singular value decomposition)—or PCA (principal component analysis) or CX/CUR decompositions—is a very complicated object compared with what is conveniently supported by traditional database operations [1]. Recall that PCA is a popular method that finds the mutually orthogonal eigencomponents that maximize the variance captured by the factorization, and CX/CUR provides an interpretable low-rank factorization by selecting a small number of columns/rows from the original data matrix as its factors. Described in more detail in Section II, these low-rank approximation methods are popular in small- and medium-scale machine learning and scientific data analysis applications for exploratory and interactive data analysis and for providing compact and/or interpretable representations of complex matrix-based data, but their implementation at scale remains a challenge.

These matrix-based methods do not mesh well with much of the work that has been popular recently in large scale data analysis. For example, MapReduce/Hadoop does a certain amount of reading and writing at every step and thus iterative algorithms are prohibitive [2]. Apache Spark solves some of these problems by maintiaing some additional state, but even there systems are not designed for nontrivial matrix algorithms [3]. There is an associated question of the ideal hardware platform for running Big Data Analytics. Conventional EC2 class hardware utilizes loosely coupled nodes; whereas typical HPC system are much more tightly coupled with high performance interconnects. Frameworks such as Hadoop and Spark have been developed for EC2 class hardware, and their performance on HPC hardware, especially for complex analytics problems (such as linear algebra and matrix decompositions) is largely unexplored.

In this paper, we address the following research questions:

- Can we successfully apply low rank matrix factorization methods (such as CX) to a large scale scientific dataset?
- Can we implement CX in a contemporary data analytics framework such as Spark?
- What is the performance gap between a highly tuned C, and a Spark-based CX implementation?

- How well does Spark-based CX implementation scale on contemporary HPC and data-center hardware platforms?

We start with a description of matrix factorization algorithms in Section II, followed by single node and multi-node implementation details in Section III. We review the experimental setup for all of our performance tests in Section IV, followed by results and discussion in Section V.

II. Low-rank matrix factorization 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

$$\underset{m\times n}{A} \approx \underset{m\times k}{Y} \times \underset{k\times n}{Z}, \tag{1}$$

where $Y \times Z$ is a rank-k approximation to the original matrix A. Low-rank matrix factorization methods are an important topic in linear algebra and numerical analysis, and they find use in a variety of scientific fields and scientific computing as well as in machine learning and data analysis applications such as pattern recognition and personalized recommendation.

Depending on the application, various low-rank factorization techniques are of interest. Popular choices include the singular value decomposition [4], principal component analysis [5], rank-revealing QR factorization [6], nonnegative matrix factorization [7], and CUR/CX decompositions [8]. In this work, we consider using the SVD and CX decompositions for our scalable and interpretable data analysis; and, in the remainder of the section, we briefly describe these decompositions. For an arbitrary matrix A, denote by $\mathbf{a}_i$ its i-th row, $\mathbf{a}^j$ its j-th column and $\mathbf{a}_{ij}$ its (i,j)-th element. Hereby, we assume the data matrix A has size $m \times n$ and rank r.

A. SVD and PCA

The singular value decomposition (SVD) is the factorization of matrix $A \in \mathbb{R}^{m\times n}$ into the product of three matrices $U\Sigma V^T$ where $U \in \mathbb{R}^{m\times r}$ and $V \in \mathbb{R}^{n\times r}$ have orthonormal columns and $\Sigma \in \mathbb{R}^{r\times r}$ is a diagonal matrix with positive real entries. The columns of U and V are called left and right singular vectors and the diagonal entries of Σ are called singular values. For notation convenience, we assume the singular values are sorted such that $\sigma_1 \geq \cdots \geq \sigma_r \geq 0$, and this means that the columns of U and V are sorted by the order given by the singular values.

The SVD is of central interest 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

$$\min_{\text{rank}(\tilde{A})=k} \|A - \tilde{A}\|_F, \tag{2}$$

where the Frobenius norm $\|\cdot\|_F$ is defined as $\|X\|_F^2 = \sum_{i=1}^m \sum_{j=1}^n X_{ij}^2$. Specifically, the solution to (2) 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 singular vectors, i.e., the first k columns of U and V, respectively, and Σ_k is a diagonal matrix containing the top-k singular values.

Principal component analysis (PCA) and SVD are closely related. PCA aims to convert the original features into a set of linearly uncorrelated variables called *principal components*. To be more specific, the first principal component is defined to be the direction along which the highest variance possible among the data points is attained, and each succeeding principal component in turn has the largest variance possible subject to the constraint that it is orthogonal to the preceding principal components. When low-rank methods are appropriate, the number of principal components needed to preserve most of the information in A is far less than the number of original features, and thus the goal of dimension reduction is achieved. The PCA decomposition of A is defined as the SVD of the matrix formed by centering each column of A (i.e., removing the mean of each column).

B. Randomized SVD

The computation of the SVD (and thus of PCA for a data matrix A) is expensive [4]. For example, to compute the truncated SVD with rank k using traditional deterministic methods, the running time complexity is $\mathcal{O}(mnk)$, and $\mathcal{O}(k)$ passes over the dataset are needed. This becomes prohibitively expensive when dealing with datasets of even moderately-large size, e.g., $m = 10^6$, $n = 10^4$ and $k = 20$. To address these and related issues, recent work in Randomized Linear Algebra (RLA) has focused on using randomized approximation, e.g., random projection and random sampling methods, to perform scalable linear algebra computations for large-scale data problems. For an overview of the RLA area, see [9]; for a review of using RLA methods for low-rank matrix approximation, see [10]; and for a review of the theory underlying implementing RLA methods in parallel/distributed environments, see [11].

Here, we will use an algorithm described in Halko et al. [10] which uses a random projection to construct a rank-k approximation to A which approximates A nearly as well as A_k does. We refer the readers to [9], [10] for more details. Importantly, the algorithm runs in $\mathcal{O}(mn \log k)$ time, and the algorithm needs only a constant number of passes over the data matrix. These properties becomes extremely desirable in many large-scale data analytics. This algorithm, which we refer to as RandomizedSVD, is summarized in Algorithm 1. (Algorithm 1 calls MultiplyGramian, which is summarized in Algorithm 2, as well as three algorithms, Multiply, ThinQR, and ThinSVD, which are standard in numerical linear algebra [4].) The running time cost for RandomizedSVD is dominated by the matrix-matrix multiplication, which involve passing over the entire data

Algorithm 1 RANDOMIZEDSVD Algorithm

Input: $A \in \mathbb{R}^{m\times n}$, number of power iterations $q \geq 1$, target rank $r > 0$, slack $\ell \geq 0$, and let $k = r + \ell$.
Output: $U\Sigma V^T \approx \text{THINSVD}(A, r)$.

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

Algorithm 2 MULTIPLYGRAMIAN Algorithm

Input: $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times k}$.
Output: $X = A^T AB$.

1: Initialize $X = 0$.
2: **for** each row a in A **do**
3: $\quad X \leftarrow X + aa^T B$.
4: **end for**

matrix, appearing in Step 3 and Step 7 of Algorithm 1. These steps can be parallelized, and hence RANDOMIZEDSVD is well amenable to distributed computing.

C. CX/CUR decompositions

In addition to developing improved algorithms for PCA/SVD and related problems, work in RLA has also focused on so-called CX/CUR decompositions [8], [12]. As a motivation, observe that singular vectors are eigenvectors of the Gram matrix $A^T A$, and thus they are linear combinations of up to all of the original variables. A natural question arises: can we reconstruct the matrix using a small number of actual columns of A?

CX/CUR decompositions affirmatively answer this question. That is, these are low-rank matrix decompositions that are expressed in terms of a small number of columns/rows, i.e, actual data elements, and not a small number of eigencolumns/eigenrows (which form the principal components). As such, they have found applicability in scientific applications where coupling analytical techniques with domain knowledge is at a premium, including genetics [13], astronomy [14], and mass spectrometry imaging [15].

In more detail, CX decomposition factorizes an $m \times n$ matrix A into two matrices C and X, where C is an $m \times c$ matrix that consists of c actual columns of A, and X is a $c \times n$ matrix such that $A \approx CX$. (CUR decompositions further choose $X = UR$, where R is a small number of actual rows of A [8], [12].) For CX, using the same optimality criterion defined in (2), we seek matrices C and X such that the residual error $\|A - CX\|_F$ is small.

Algorithm 3 CXDECOMPOSITION

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

1: Compute an approximation of the top-k right singular vectors of A denoted by $\tilde{V}_k$, using RANDOMIZEDSVD with q power iterations.
2: Let $\ell_i = \sum_{j=1}^{k} \tilde{\mathbf{v}}_{ij}^2$, where $\tilde{\mathbf{v}}_{ij}^2$ is the (i, j)-th element of $\tilde{V}_k$, for $i = 1, \ldots, n$.
3: Define $p_i = \ell_i / \sum_{j=1}^{d} \ell_j$ for $i = 1, \ldots, n$.
4: Randomly sample c columns from A in i.i.d. trials, using the importance sampling distribution $\{p_i\}_{i=1}^{n}$.

The algorithm of [12] that computes a $1 \pm \epsilon$ relative-error low-rank CX matrix approximation consists of three basic steps: first, compute (exactly or approximately) the *statistical leverage scores* of the columns of A; and second, use those scores as a sampling distribution to select c columns from A and form C; finally once the matrix C is determined, the optimal matrix X with rank-k that minimizes $\|A - CX\|_F$ can be computed accordingly; see [12] for detailed construction. In this paper we focus on the first two steps.

The algorithm for approximating the leverage scores is provided in Algorithm 3. Let $A = U\Sigma V^T$ be the SVD of A. Given a target rank parameter k, for $i = 1, \ldots, n$, the i-th leverage score is defined as

$$\ell_i = \sum_{j=1}^{k} \mathbf{v}_{ij}^2. \tag{3}$$

These scores quantify the amount of "leverage" each column of A exerts on the best rank-k approximation to A. For each column of A, we have

$$\mathbf{a}_i = \sum_{j=1}^{r} (\sigma_j \mathbf{u}_j)\mathbf{v}_{ij} \approx \sum_{j=1}^{k} (\sigma_j \mathbf{u}_j)\mathbf{v}_{ij}.$$

That is, the i-th column of A can be expressed as a linear combination of the basis of the dominant k-dimensional left singular space with $\mathbf{v}_{ij}$ as the coefficients. If, for $i = 1, \ldots, n$, we define the *normalized leverage scores* as

$$p_i = \frac{\ell_i}{\sum_{j=1}^{n} \ell_j}, \tag{4}$$

where ℓ_i is defined in (3), and choose columns from A according to those normalized leverage scores, then (by [8], [12]) the selected columns are able to reconstruct the matrix A nearly as well as A_k does.

Observe that the running time for CXDECOMPOSITION is determined by the computation of the importance sampling distribution. To compute the leverage scores based on (3), i.e., exactly, one needs to compute the top k right-singular

vectors V_k. As pointed out above, this is prohibitive data with massive size. However, just as with our SVD computation, here we can use RANDOMIZEDSVD to compute *approximate* leverage scores. This approach, originally proposed by Drineas et al. [16], runs in "random projection time," i.e., it requires fewer FLOPS and fewer passes over the data matrix than the traditional deterministic algorithms that compute the leverage scores exactly.

III. HIGH PERFORMANCE IMPLEMENTATION

We undertake two classes of high performance implementations for the CX method. We start with a highly optimized, close-to-the-metal C implementation that focuses on obtaining peak efficiency from conventional multi-core CPU chipsets and extend it to multiple nodes. Secondly, we implement the CX method in Spark, an emerging standard for parallel data analytics frameworks.

A. Single Node Implementation/Optimizations

We now focus on optimizing the CX implementation on a single compute-node, aimed at exploiting the available SIMD units on each core, and the multiple cores across different sockets to speed up the performance. We began by profiling our initial scalar serial CX code and optimizing the steps in the order of execution times. The most time is spent in computing sparse-sparse-dense matrix multiplication (A^TAB, Step 3, 90.6%), followed by sparse-dense matrix multiplication (AQ, Step 7, 9.1%) and finally, QR decomposition (Step 4, 0.4%) for a representative dataset that fits in main memory. These three kernels account for more than 99.9% of the execution time. Recall that A is a sparse matrix with dimensions $m \times n$ and sparsity s, and B is a dense $n \times k$ matrix. We first discuss the case of A^TAB.

1) Optimizing $\mathrm{Res} = A^TAB$*:* Optimizing sparse matrix-matrix multiplication continues to be an active area of research [17], [18], with focus on reducing communication, and the state of the art implementations are bound by the memory bandwidth and heavily underutilize the compute resources. For our application, we exploit the following three observations: (1) One of the sparse matrices is the transpose of the other, (2) One of the matrices is a dense matrix, and (3) $n \gg k$ and $sm \gg k$.

Exploiting associativity of multiplication, we perform C = AB, followed by $\mathrm{Res} = A^TC$. This reduces the run-time complexity from O(*n*(nsm)*) to O(*k*(nsm)*). Furthermore, we do not explicitly compute (or store) the transpose of A. Consider the i^{th} row, A_i. By definition, $C_i = A_i \cdot B$. The $(j,l)^{th}$ element of Res, $\mathrm{Res}_{j,l} = \Sigma_p(A^T_{j,p}$ x $C_{p,l}) = \Sigma_p(A_{p,j}$ x $C_{p,l})$. For p = i, this reduces to incrementing $\mathrm{Res}_{j,l}$ by $A_{i,j}$ x $C_{i,l}$. Thus, for each row *i*, having computed C_i, we increment $\mathrm{Res}_{j,l}$ by $A_{i,j}$ x $C_{i,l}$ for *j*∈[1..*n*] and *l*∈[1..*k*]. We now describe how we parallelize to exploit data- and thread-level parallelism and other relevant optimizations.

1. Exploiting SIMD: Refer to Figure 1. Consider element $A_{i,j}$. To compute C_i, we need to scale each element of B_j by $A_{i,j}$ and add it to C_i (*j*∈[1..*n*]) (C_i += $A_{i,j}$ x B_j). Note that there are *k* elements in B_j, which are also stored consecutively (matrix stored in a row-major fashion). On modern computing platforms, SIMD width (number of simultaneous operations that can be performed) is growing [19]. SSE can perform 4 single-precision floating point operations in a single operation, while AVX (our SPARK platform) performs 8 ops. Let $\mathcal{S}$ denote the SIMD width (defined as number of double-precision floating point ops. per op – which is half of single-precision ops). The pseudo-code [1] for computing C_i ($\forall\ A_{i,j} \neq 0$)

```
  xmm_a = vec_load_and_splat(A_{i,j});
for (z = 0; z < k; z += S)
{
    xmm_c = vec_load (C_j + z);
    xmm_b = vec_load (B_j + z);
    xmm_ab = vec_mul (xmm_a, xmm_b);
    xmm_c = vec_add (xmm_ab, xmm_c);
    vec_store (xmm_C, C_j + z);
}
```

As highlighted in the code, for each $A_{i,j} \neq 0$ (number of nonzeros (*nnz*) in total in matrix A), we execute ($\lceil \frac{k}{\mathcal{S}} \rceil$) *add* (and same number of *mul*) operations – for a total of $\lceil \frac{2*nnz*k}{\mathcal{S}} \rceil$ operations, a potential speedup of $\mathcal{S}$.

We now describe the vectorization of $X = A^TC$. As explained above, this requires incrementing X_j by $A_{i,j}$ x C_i (both X_j and C_i have *k* elements each.) We execute a similar code to the previous step, to perform $\lceil \frac{2*nnz*k}{\mathcal{S}} \rceil$ operations, a potential speedup of $\mathcal{S}$.

2. Exploiting multiple cores: As explained above, we decompose the matrix multiplication into two steps, we *first* compute C_i (*k* elements), followed by updating Res_j for each row *j*. The number of executed flops is proportional to number of non-zeros in the specific row of A. Thus a straightforward way to divide work between the the cores ($\mathcal{C}$ in total) is to divide the rows such that each of them perform work on the same number of *nnz's*.

Thus each core (or thread) computes the starting and ending row index, and for each assigned row *i*, computes C_i. Now, the next step is to update Res. Two different possibilites exist. One option is for each thread to maintain a local copy of Res, and once all the threads are done executing, reduce the results to form the final answer. However, even for moderately sized datasets, (e.g. k = 32, n = 32K, and 8-bytes/element amounts to around 4MB/thread), *far exceeds* the cache per core. For each assigned row, would load and store the complete Res matrix. A more efficient

[1]Exact syntax varies with the ISA and compiler version.

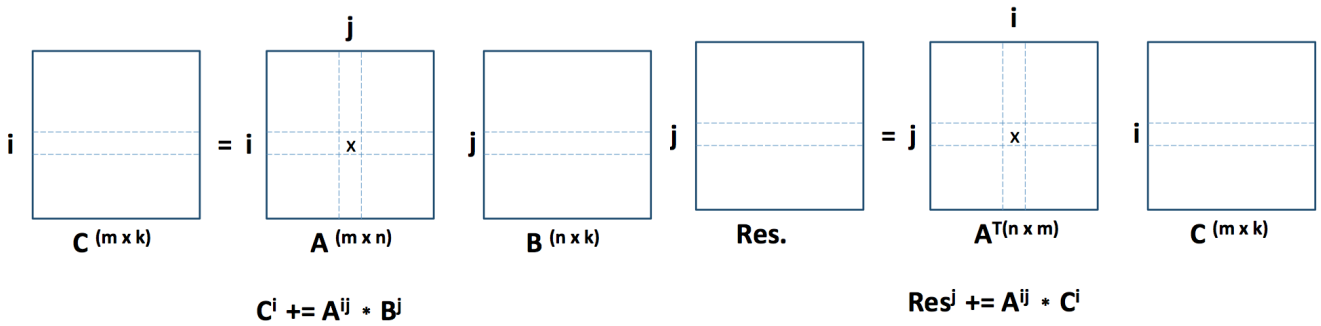

Figure 1: Data access pattern for computing C_i (left) and Res_j (right) respectively.

approach is to maintain a single copy of Res shared by all the threads and updated using locks, as described next.

We initialize n locks, one for each row of the output matrix (Res). Once an executing thread computes C_i, for each $A_{i,j} \neq 0$, it grabs the j^{th} lock, updates the row, and releases the lock. For realistic datasets, for sparsity(s) ($\sim$ 0.001 – 0.005), there is a very low probability of two threads blocking on a lock ($\sim$1% even with $\mathcal{C}$ = 128).

3. Cache Blocking: For smaller values of n, our thread-level parallelization scheme scales near-linearly with increasing number of cores. However, for $n > 64$K, we started noticing a drop in scaling. This is due to the working set growing larger than the size of the last-level cache, and thereby the computation becoming bound by the available memory bandwidth. In contrast, if most of the memory fetches can come from the caches, we can efficiently utilize the floating point compute units on the node. We now describe the computation of the working set, and our algorithm for performing cache-friendly updates.

During the execution of the algoritm, the matrix B is accessed, which is shared between all the cores. Matrix A is a streaming read from the memory, and does not contribute to the working set. Let's say each thread maintains its local copy of the Res matrix, thereby the total working set being 8kn*($\mathcal{C}$ + 1) bytes. For our system architecture, with $\mathcal{C}$ = 24, and matrix parameters of k = 32 and n = 128K, the total working set becomes around 1 GB, which is too large to fit in the caches [2]. Instead, maintaining a shared copy of the Res matrix reduces it to 8kn bytes, around 128 MB. Note that the total size is indepent of number of cores, and thus future proofs our implementation. However, it is still dependent on n, the number of columns in the input matrix A, and thus we devise the following scheme to reduce it further to a given cache size.

Instead of performing the computation for n columns, we divide it into chunks of n' columns, such that 2*8*k*n' $\sim$ $\mathcal{C}$. Hence, with $\mathcal{C}$ = 15 MB, $n'\sim$ 64K elements (we set n to be a multiple of n'). We thus perform the computation in $\lceil \frac{n}{n'} \rceil$ rounds, updating the corresponding rows ([$r\lceil \frac{n}{n'} \rceil$..($r$ + 1)$\lceil \frac{n}{n'} \rceil$] in round r). Recall from the previous subsection that the number of flops executed per nonzero element in A is $\lceil \frac{4k}{\mathcal{S}} \rceil$. Since the non zeros elements of A are stored consecutively, this may require loading each element $\lceil \frac{n}{n'} \rceil$ times. Hence, the flops/byte of the computation is around $\lceil \frac{4k}{\mathcal{S}} \rceil / \lceil \frac{n}{n'} \rceil$. Using our representative numbers, this is around 16 flops/byte, which is greater than the peak flops/byte of the platform (around 10 flops/byte), and hence our application is not bound by memory bandwidth. With large values of n, we might end up being bandwidth bound – in which case we need to modify the way A is stored, by storing it in chunks of columns that would be accessed in each round. This format of representing A helps exploit the complete computational power of the processor, and only incurs one-time cost of rearranging the elements of A. n' = 64K seems to be a resonable size for current architectures.

4. Multi-socket Optimization: Multi-socket architectures are increasing being used, wherein each socket has its own compute and memory resources. It is indeed possible for cores in any socket to access data present in the memory of the other sockets. However, all cross-socket traffic goes through a cross-socket link, which has lower bandwidth than access to local DRAM and caches. Hence, we need to optimize the amount of data transferred between sockets to ensure optimal performance.

We divide the allocaton of Res equally between the sockets. For e.g., for a CPU with 2 sockets, we divide the number of rows (n) by 2, and allocate the memory for each relevant part of the matrix on its individual socket. This ensures that (at an average), each socket has similar number of remote memory accesses. For our experiments, the prescribed style of memory allocation provided a boost of $\sim$5 – 10% to our performance, but we expect the optimizaton to be more beneficial with increasing number of sockets.

2) Optimizing AB: This step refers to Step 7 in the algorithm description in Algorithm 1. The data- and thread-level paralleization optimizations described in the previous

[2]In this discussion, caches refers to the last level cache

subsection (optimizing A^TAB) apply here, since there we explicitly compute $C = AB$. As far as cache blocking is concerned, since C does not have to be memory resident, we now have to ensure that B is completely cache resident (i.e. $8nk \leq \mathcal{C}$). With increasing n, we again peform the computation in multiple rounds, with each round operating on $n' = \frac{\mathcal{C}}{8k}$ rows of B. Finally, as far as multi-socket optimizations are concerned, we divide the allocation of C, the output matrix in this case, between the various sockets, to reduce the amount of cross-socket memory traffic.

B. Multi-Node Implementation

Consider A^TAB. Similar to the multi-core implementation, we achieve load balancing by dividing the rows such that each node operates on the same number of non zeros. Since only the total number of non zeros in A (and the number of rows) is known at the start of the CX computation, we perform this partitioning using a two step process. In the first step, we equally divide the number of rows, and each node reads in the corresponding part of the matrix, and computes the number of non-zeros read. This is followed by a redistribution step, where each node computes and distributes the relevant rows. The amount of data transferred between nodes is only a small fraction of the total input size (measured $< 0.01\%$), and this step is only performed once during the execution of the algorithm. Each node computes the local resultant matrix Res, which is then reduced globally to compute the final matrix. Note that Res consists of n X k elements, which occupies a few MBs even for our largest 1 TB datset (recall $m \gg n$).

As far as QR decompositon is concerned, given the small size of the matrix (n X k), it is performed on a single-node, but parallelized to exploit the multiple cores [**?**], with the resultant matrix being broadcast to all other nodes at the end of the computation. A similar work division scheme is used to compute AB (Step 7) in a distributed fashion. The final two steps (ThinSVD and small matrix multiplication) are performed on a single-node (given the small matrix sizes).

C. Spark

To support operating on datasets larger than can be stored and processed on a single node, we implement the algorithms using the Apache Spark cluster computing framework. Spark provides a high-level programming model and execution engine for fault-tolerant parallel and distributed computing, based on a core abstraction called *resilient distributed dataset (RDD)*. Each RDD may be thought of as a distributed collection of objects that is partitioned and stored across the Spark cluster. RDDs are immutable lazily materialized collections supporting functional programming operations such as `map`, `filter`, and `reduce`, each of which returns a new RDD. RDDs may be loaded from a distributed file system, computed from other RDDs, or created by parallelizing a collection created within the user's application. RDDs of key-value pairs may also be treated as associative arrays, supporting operations such as `reduceByKey`, `join`, and `cogroup`. Spark employs a lazy evaluation strategy for efficiency. Another major benefit of Spark over MapReduce is the use of in-memory caching and storage so that data structures can be reused.

D. Multi-node CX and PCA implementation on Spark

The main consideration when implementing CX and PCA are efficient implementations of operations involving the data matrix A. All access of A by the CX and PCA algorithms occurs through the RANDOMIZEDSVD routine shared in common. RANDOMIZEDSVD in turn accesses A only through the MULTIPLYGRAMIAN and MULTIPLY routines, with repeated invocations of MULTIPLYGRAMIAN accounting for the majority of the execution time.

The matrix A is stored as an RDD containing one `IndexedRow` per row of the input matrix, where each `IndexedRow` consists of the row's index and corresponding data vector. This is a natural storage format for many datasets stored on a distributed or shared file system, where each row of the matrix is formed from one record of the input dataset, thereby preserving locality by not requiring data shuffling during construction of A.

We then express MULTIPLYGRAMIAN in a form amenable to efficient distributed implementation by exploiting the fact that the matrix product A^TAB can be written as a sum of outer products, as shown in Algorithm 2. This allows for full parallelism across the rows of the matrix with each row's contribution computed independently, followed by a summation step to accumulate the result. This approach may be implemented in Spark as a `map` to form the outer products followed by a `reduce` to accumulate the results:

```
def multiplyGramian(A: RowMatrix, B: LocalMatrix) =
  A.rows.map(row => row * row.t * B).reduce(_ + _)
```

However, this approach forms $2m$ unnecessary temporary matrices of same dimension as the output matrix $n \times k$, with one per row as the result of the `map` expression, and the `reduce` is not done in-place so it too allocates a new matrix per row. This results in high Garbage Collection (GC) pressure and makes poor use of the CPU cache, so we instead remedy this by accumulating the results in-place by replacing the `map` and `reduce` with a single `treeAggregate`. The `treeAggregate` operation is equivalent to a map-reduce that executes in-place to accumulate the contribution of a single worker node, followed by a tree-structured reduction that efficiently aggregates the results from each worker. The reduction is performed in multiple stages using a tree topology to avoid creating a single bottleneck at the driver node to accumulate the results from each worker node. Each worker emits a relatively large result with dimension $n \times k$, so the communication latency savings of having multiple reducer tasks is significant.

```
def multiplyGramian(A: RowMatrix, B: LocalMatrix) = {
```

```
  A.rows.treeAggregate(LocalMatrix.zeros(n, k))(
    seqOp = (X, row) => X += row * row.t * B,
    combOp = (X, Y) => X += Y
  )
}
```

IV. Experimental Setup

A. MSI Dataset

Mass spectrometry imaging with ion-mobility: Mass spectrometry measures ions that are derived from the molecules present in a complex biological sample. These spectra can be acquired at each location (pixel) of a heterogeneous sample, allowing for 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 capabilities available to many scientists. The addition of IMS exacerbates these problems.

Utility of CX/PCA in MSI: Dimensionality reduction techniques can help reduce MSI datasets to more amenable sizes. Typical approaches for dimensionality reduction include PCA and NMF, but interpretation of the results is difficult because the components extracted via these methods are typically combinations of many hundreds or thousands of features in the original data. CX decompositions circumvent this problem by identifying small numbers of columns in the original data that reliably explain a large portion of the variation in the data. This facilitates rapidly pinpointing important ions and locations in MSI applications.

In this paper, we analyze one of the largest (1TB sized) mass-spec imaging datasets in the field. The sheer size of this dataset has previously made complex analytics intractable. This paper presents first-time science results from the successful application of CX to TB-sized data.

B. Platforms

In order to assess the relative performance of CX matrix factorization on various hardware, we choose the following contemporary platforms:

- a Cray® XC40™ system [20], [21],
- an experimental Cray cluster, and
- an Amazon EC2 `r3.8xlarge` cluster.

For all platforms, we sized the Spark job to use 960 executor cores (except as otherwise noted). Table I shows the full specifications of the three platforms. Note that these are state-of-the-art configurations in datacenters and high performance computing centers.

V. Results

A. CX Performance on Single Node

In Table II, we show the benefits of various optimizations described in Sec. III-A to the performance of MULTIPLY-GRAMIAN and MULTIPLY on each compute node. The test matrix $\mathcal{A}$ has m = 1.95M, n = 128K, s = 0.004, and nnz = 10^9. The parameter k = 32. We started with a parallelized implementation, without any of the described optimizations, and measured the performance (in terms of time taken). We first implemented the multi-core synchronization scheme, wherein a single copy of the output matrix is maintained across all the threads (for the matrix multiplication). This resulted in a speedup of around 6.5X, primarily due to the reduction in the amount of data traffic between the last-level cache and main memory (there was around 19X measured reduction in traffic). We then implemented our cache blocking scheme, primarily targeted towards ensuring that the output of the matrix multiplication resides in the caches (since it is accessed and updated frequently). This led to a further 2.4X reduction in run-time, for an overall speedup of around 15.6X.

Once the memory traffic was optimized for, we implemented our SIMD code by vectorizing the element-row multiplication-add operations (described in detail in Sec. III-A). The resultant code sped up by a further 2.6X, for an overall speedup of 39.7X. Although the effective SIMD width ($\mathcal{S}$ = 4), there are overheads of address computation, stores, and not all computations were vectorized (QR code is still scalar).

We did a head-to-head comparison of C code with the Scala node implementation on a single node, and measured a performance gap of around 20X. This performance gap can be attributed to the careful cache optimizations of maintaining single copy of the output matrix shared across threads, bandwidth friendly access of matrices and vector computation using SIMD units.

Some of these optimizations can be implemented in Spark, such as arranging the order of memory accesses to make efficient use of the memory bus. However, other optimizations such as sharing the output matrix between threads and use of SIMD intrinsics fall outside the Spark programming model, and would require piercing the abstractions provided by Spark and the JVM to more directly access and manipulate the hardware. We thus find that there is a tradeoff between optimizing absolute performance on each machine, versus the ease of implementation and efficient global scheduling available by expressing programs in the Spark programming model.

B. CX Performance across Multiple Nodes

1) CX Spark Phases: Our implementations of CX and PCA share the RANDOMIZEDSVD subroutine, which accounts for the bulk of the runtime and all of the distributed

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

Table I: Specifications of the three hardware platforms used in these performance experiments.

Single Node Optimization	Overall Speedup
Original Implementation	1.0
Multi-Core Synchronization	6.5
Cache Blocking	15.6
SIMD	39.7

Table II: Single node optimizations to the CX C implementation and the subsequent speedup each additional optimization provides.

computations. The execution of RANDOMIZEDSVD proceeds in four distributed phases listed below, along with a small amount of additional local computation.

1) **Load Matrix Metadata** The dimensions of the matrix are read from the distributed filesystem to the driver.
2) **Load Matrix** A distributed read is performed to load the matrix entries into an in-memory cached RDD containing one entry per row of the matrix.
3) **Power Iterations** The MULTIPLYGRAMIAN loop on lines 2-5 of RANDOMIZEDSVD is run to compute an approximation Q of the dominant right singular subspace.
4) **Finalization (Post-Processing)** Right multiplication by Q on line 7 of RANDOMIZEDSVD to compute C.

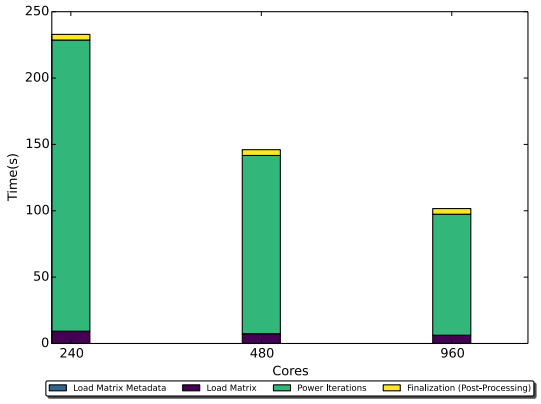

Figure 2: Strong scaling for the 4 phases of CX on an XC40 for 100GB dataset at $k = 32$ and default partitioning as concurrency is increased.

2) Empirical Results: Figure 2 shows how the distributed Spark portion of our code scales as we add additional processors. We considered 240, 480, and 960 cores. An additional doubling (to 1920 cores) would be ineffective as there are only 1654 partitions, so many cores would remain unused. In addition, with fewer partitions per core there are fewer opportunities for load balancing and speculative reexecution of slow tasks.

When we go from 240 to 480 cores, we achieve a speedup of approximately 1.6x from doubling the cores: 233 seconds versus 146 seconds. However, as the number of partitions per core drops below two, and the amount of computation-per-core relative to communication overhead drops, the scaling slows down (as expected). This results in a lower speedup of approximately 1.4x (146 seconds versus 102 seconds) when we again double the core count to 960.

C. CX Performance across Multiple Platforms

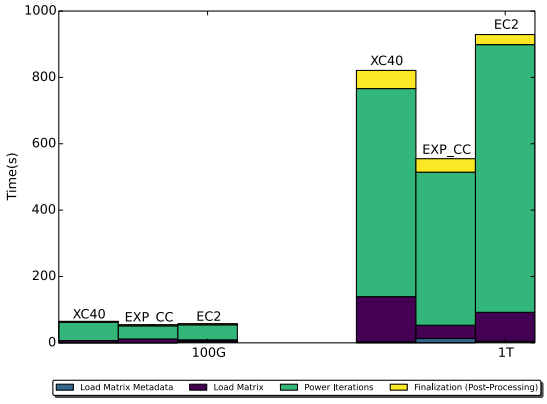

Figure 3: Run times for the various stages of computation for CX for two different dataset sizes for the three platforms using $k = 16$ and default partitioning for the given platform

Table III shows the total runtime of CX for the 1 TB dataset on our three platforms. The distributed Spark portion of the computation is also depicted visually in Figure 3 for $k = 16$ (similar results were obtained for $k = 32$, but have been omitted due to length considerations). All three platforms were able to successfully process the 1 TB dataset in under 25 minutes. As the table and figure illustrates, most of the variation between the platforms occurred during the `MultiplyGramian` iterations. Table I shows the specifications of the three platforms. In this section, we explore

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

Table III: Total runtime for the 1 TB dataset ($k = 16$), broken down into load time and per-iteration time. The per-iteration time is further broken down into the average time for each task of the local stage and each task of the aggregation stage. We also show the average amount of time spent waiting for a network fetch, to illustrate the impact of the interconnect.

how these difference relate to the performance of the matrix iterations.

Spark divides each iteration into two stages. The first *local* stage computes each row's contribution, sums the local results (the rows computed by the same worker node), and records these locally-aggregated results. The second *aggregation* stage combines all of the workers' locally-aggregated results using a tree-structured reduction. Most of the variation between platforms occurs during the aggregation phase, where data from remote worker nodes is fetched and combined. In Spark, all inter-node data exchange occurs via *shuffle operations*. In a shuffle, workers with data to send write the data to their local scratch space. Once all data has been written, workers with data to retrieve from remote nodes request that data from the sender's block manager, which in turns retrieves if from the senders local scratch space, and sends it over the interconnect to the receiving node.

Examining our three platforms (Table I), we notice two key hardware differences that impact shuffle operations:

- First, both the EC2 nodes and the experimental Cray cluster nodes have fast SSD storage local to the compute nodes that they can use to store Spark's shuffle data. The Cray® XC40™ system's [20], [21] nodes, on the other hand, have no local persistent storage devices. Thus we must emulate local storage with a remote Lustre filesystem. The impacts of this can be somewhat mitigated, however, by leaving sufficient memory to store some of the data in a local RAM disk, and/or to locally cache some of the remote writes to Lustre.[3]
- Second, the Cray XC40 and the experimental Cray cluster both communicate over the HPC-optimized Cray Aries interconnect [20], [21], while the EC2 nodes use 10 Gigabit Ethernet.

[3]This is an ideal usage of caching, since Spark assumes the scratch space is only locally accessible; thus we are guaranteed that the only node that reads a scratch file will be the same node that wrote it.

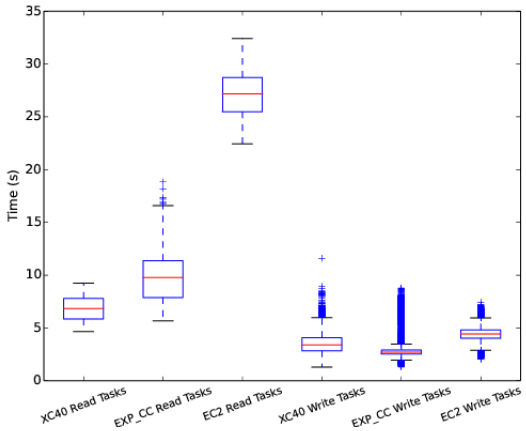

Figure 4: A box and whisker plot of the distribution of local (write) and aggregation (read) task times on our three platforms for the 1TB dataset with $k = 16$. The boxes represent the 25th through 75th percentiles, and the lines in the middle of the boxes represent the medians. The whiskers are set at 1.5 box widths outside the boxes, and the crosses are outliers (results outside the whiskers). Note that each iteration has 4800 write tasks and just 68 read tasks.

We can see the impact of differing interconnect capabilities in the Average Network Wait column in Table III. These lower average network wait times explain why the two Cray platforms outperform the EC2 instance (with the experimental cluster achieving a speedup of roughly 1.5x over EC2).

The XC40 is still slightly slower than the experimental Cray cluster, however. Part of this difference is due to the slower matrix load phase on the XC40. On EC2 and the experimental Cray cluster, the input matrix is stored in SSDs on the nodes running the Spark executors. Spark is aware of the location of the HDFS blocks, and attempts to schedule tasks on the same nodes as their input. The XC40, however, lacks SSDs on its compute nodes, so the input matrix is instead stored on a parallel Lustre file system. The increased IO latency slows the input tasks. The rest of the difference in performance can be understood by looking at the distribution of local (write) task times in the box and whiskers plot in

Figure 4. The local/write tasks are much more numerous than the aggregation/read tasks (4800 vs 68 per iteration), thus they have a more significant impact on performance. We see that the XC40 write tasks had a similar median time to the experimental cluster's write tasks, but a much wider distribution. The large tail of slower "straggler" tasks is the result of some shuffle data going to the remote Lustre file system rather than being cached locally. We enabled Spark's optional speculative re-execution (`spark.speculation`) for the XC40 runs, and saw that some of these tasks were successfully speculatively executed on alternate nodes with more available OS cache, and in some case finished earlier. This eliminated many of the straggler tasks and brought our performance closer to the experimental Cray cluster, but still did not match it (the results in Figure 3 and Table III include this configuration optimization). We discuss future directions for improving the performance on Spark on HPC systems in Section V-E.

D. Science Results

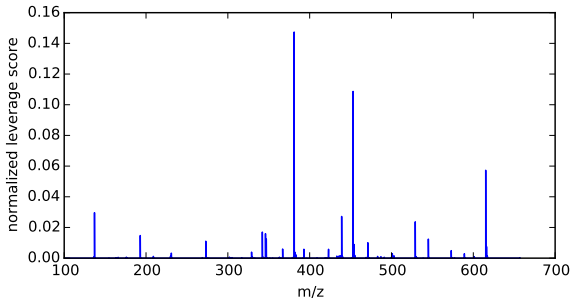

Figure 5: Normalized leverage scores (sampling probabilities) for m/z marginalized over τ. Three narrow regions of m/z account for 59.3% of the total probability mass.

1) CX: Here, we examine the results of the CX decomposition of the MSI dataset. The rows and columns of our data matrix A correspond to pixels and $(\tau, m/z)$ values of ions, respectively. We compute the CX decompositions of both A and A^T in order to identify important ions in addition to important pixels.

In Figure 5, we present the distribution of the normalized ion leverage scores marginalized over τ. That is, each score corresponds to an ion with m/z value shown in the x-axis. Leverage scores of ions in three narrow regions have significantly larger magnitude than the rest. This indicates that these ions are more informative and should be kept as basis for reconstruction. Encouragingly, several other ions with significant leverage scores are chemically related to the ions with highest leverage scores. For example, the ion with an m/z value of 453.0983 has the second highest leverage score among the CX results. Also 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 by CX 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 sample.

E. Improving Spark on HPC Systems

The differences in performance between the Cray® XC40™ system [20], [21] and the experimental Cray cluster point to optimizations to Spark that could improve its performance on HPC-style architectures. The two platforms have very similar configurations, with the primary difference being the lack of local persistent storage on the XC40 nodes. As described in Section V-C, this forces some of Spark's local scratch space to be allocated on the remote Lustre file system, rather than in local storage. To mitigate this, and keep more of the scratch data local, we propose the following future work:

- Spark is currently inefficient in cleaning up its local scratch space. In particular, shuffle data is not immediately cleaned up after a shuffle completes. This makes fault recovery more efficient, but results in higher storage requirements for scratch space. If clean up was more efficient, it would be more feasible to fit all of the scratch data in a local RAM disk and not rely on Lustre at all.
- Spark does not currently allow you to configure primary and backup scratch directories. Instead you list all scratch directories in a single list, and it distributes data in a round round fashion between them as long as space is available. You can bias it towards one storage device (e.g., RAM disk vs. Lustre) by listing multiple directories on the preferred device. Ideally, though, we would like to use a RAM disk (or other local storage) exclusively unless and until it fills, and only switch to Lustre directories if necessary.
- Spark does not allow you to specify that a scratch directory is globally accessible. Thus non-cached data is stored to the remote Lustre directory by the sender, and then later retrieved by the sender and sent to the receiver. This wastes a step, since the receiver could easily fetch the data directly from Lustre (or any other global file system).
- Alternatively, a push model of communication (as opposed to the current pull model) might be possible - however this would have implications for reliability and handling of very large data sets.[4]

[4]Storing the shuffle data to a large persistent block storage device, and only sending it as needed, allows Spark to easily shuffle more data than could fit in the remote buffers. In a push-based model, extra logic and synchronization would be necessary to ensure that the remote buffers do not overflow.

VI. Conclusions

Matrix factorizations are an important class of linear algebra computations with broad applications in Big Data analytics. In this work, we have successfully developed highly optimized versions of the CX algorithm, both on a single node, as well as distributed multi-node implementations. We utilize Spark as a contemporary data analytics framework for developing and deploying the CX algorithm, and successfully demonstrate the implementation scaling up to 960 cores. We evaluate our parallel implementation on HPC and EC2 class hardware; we find that faster interconnects enable the numerically intensive computations to run more efficiently on HPC systems. Finally, this scalable implementation was used to analyze a large TB-sized mass-spec imaging dataset; the resulting ion and spatial patterns obtained from the analysis are providing biologists with novel insights on complex molecular mechanisms in cells.

References

[1] D. Skillicorn, *Understanding complex datasets: data mining with matrix decompositions*. Boca Raton, FL: Hall/CRC Press, 2007.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Zaharia, M. Chowdhury *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

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

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

[6] M. GU and S. C. EISENSTA, "Efficient algorithms for computing a strong rank-revealing qr factorization," *SIAM J. ScI. COMPUT.*, vol. 17, no. 4, pp. 848–869, 1996.

[7] D. Lee and H. Seung, "Algorithms for non-negative matrix factorization," in *NIPS*, 2001.

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

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

[10] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, no. 2, 2011.

[11] J. Yang, X. Meng, and M. W. Mahoney, "Implementing randomized matrix algorithms in parallel and distributed environments," Tech. Rep., 2015, preprint: arXiv:1502.03032.

[12] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, "Relative-error CUR matrix decompositions." *SIAM J. Matrix Analysis Applications*, vol. 30, no. 2, pp. 844–881, 2008.

[13] 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*, vol. 3, pp. 1672–1686, 2007.

[14] 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*, vol. 147, no. 110, p. 15pp, 2014.

[15] 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*, vol. 87, no. 9, pp. 4658–4666, 2015.

[16] P. Drineas, M. Magdon-Ismail, M. W. Mahoney, and D. P. Woodruff, "Fast approximation of matrix coherence and statistical leverage," *Journal of Machine Learning Research*, vol. 13, 2012.

[17] G. Ballard, A. Buluç *et al.*, "Communication optimal parallel multiplication of sparse random matrices," in *SPAA*, 2013.

[18] M. M. A. Patwary *et al.*, "Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms," in *ISC*, 2015, pp. 48–57.

[19] Intel, "Intel Advanced Vector Extensions Programming Reference," *White paper*, June 2011.

[20] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," *Cray Inc. White Paper WP-Aries01-1112*, 2012.

[21] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: A scalable HPC system based on a Dragonfly network," ser. SC '12, 2012, pp. 103:1–103:9.