Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures

Austin R. Benson
Institute for Computational and Mathematical Engineering
Stanford University
Stanford, CA
arbenson@stanford.edu

David F. Gleich
Department of Computer Science
Purdue University
West Lafayette, IN
dgleich@purdue.edu

James Demmel
Computer Sciences Division and Department of Mathematics
University of California, Berkeley
Berkeley, CA
demmel@cs.berkeley.edu

***Abstract*—The QR factorization and the SVD are two fundamental matrix decompositions with applications throughout scientific computing and data analysis. For matrices with many more rows than columns, so-called "tall-and-skinny matrices," there is a numerically stable, efficient, communication-avoiding algorithm for computing the QR factorization. It has been used in traditional high performance computing and grid computing environments. For MapReduce environments, existing methods to compute the QR decomposition use a numerically unstable approach that relies on indirectly computing the Q factor. In the best case, these methods require only two passes over the data. In this paper, we describe how to compute a stable tall-and-skinny QR factorization on a MapReduce architecture in only slightly more than 2 passes over the data. We can compute the SVD with only a small change and no difference in performance. We present a performance comparison between our new direct TSQR method, indirect TSQR methods that use the communication-avoiding TSQR algorithm, and a standard unstable implementation for MapReduce (Cholesky QR). We find that our new stable method is competitive with unstable methods for matrices with a modest number of columns. This holds both in a theoretical performance model as well as in an actual implementation.**

***Keywords*-matrix factorization, QR, SVD, TSQR, MapReduce, Hadoop**

I. Introduction

The QR factorization of an $m \times n$ real-valued matrix A is:

$$A = QR$$

where Q is an $m \times n$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. It is a fundamental subroutine in many advanced data analysis procedures including principal components analysis, linear regression, and general linear models. We call a matrix tall-and-skinny if it has many more rows than columns ($m \gg n$). This case is common in big data applications with billions of data points with only a few hundred descriptors. In practice, this means that it is cheap to distribute $\mathcal{O}(n^2)$ data to all processors and cheap to perform $\mathcal{O}(n^3)$ floating point operations in serial. In this paper, we study algorithms to compute a QR factorization of a tall-and-skinny matrix for nearly-terabyte sized matrices on MapReduce architectures [7].

Previous work by one of the authors gave a fast MapReduce method to compute *only* R [5]. (The details of these are described further in Sec. II.) In order to compute the matrix Q, an indirect formulation is used:

$$Q = AR^{-1}.$$

For R to be invertible, A must be full-rank, and we assume A is full-rank throughout this paper. The indirect formulation is known to be numerically unstable. Numerically stable algorithms are important because they ensure that the algorithm behaves predictably regardless of the properties of the input. This feature is vital for implementing algorithmic libraries that handle the diversity of data input in real-world applications. We refer readers to the text by Higham for more on the numerical properties of various algorithms [11].

The simple process of repeating the algorithm, which is called iterative refinement, can sometimes be used to produce a Q factor with acceptable accuracy [14]. However, if a matrix is sufficiently ill-conditioned, iterative refinement will still result in a computed matrix Q that is not orthogonal, and hence, is not nearly the QR factorization from any matrix. We shall describe a numerically stable method (Sec. III) that computes Q and R directly in approximately the same time as performing the repetition of the indirect computation for some matrices.

In Sec. IV-A, we present a performance model for our algorithms on a MapReduce cluster, which allows us to compute lower bounds on running times. Real world performance is almost always within a factor of two of the lower bounds (Sec. IV-B).

A. MapReduce motivation

The data in a MapReduce computation is defined by a collection of key-value pairs. When we use MapReduce to analyze tall-and-skinny matrix data, a key represents the identity of a row and a value represents the elements in that row. Thus, the matrix is a collection of key-value pairs. We assume that each row has a distinct key for

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

simplicity; although we note that our methods also handle cases where each key represents a set of rows.

There are a growing number of MapReduce frameworks that implement the same computational engine: first, *map* applies a function to each key-value pair which outputs a transformed key-value pair; second, *shuffle* rearranges the data to ensure that all values with the same key are together; finally, *reduce* applies a function to all values with the same key. The most popular MapReduce implementation – Hadoop [19] – stores all data and intermediate computations on disk. Thus, we do not expect numerical linear algebra algorithms for MapReduce to be faster than state-of-the-art in-memory MPI implementations running on clusters with high-performance interconnects. However, the MapReduce model offers several advantages that make the platform attractive for large-scale, large-data computations (see also [20] for information on tradeoffs). First, many large datasets are already warehoused in MapReduce clusters. With the availability of algorithms, such as QR, on a MapReduce cluster, these data do not need to be transferred to another cluster for analysis. In fact, a simple corollary of our analysis is the performance of the algorithms is largely bounded by simply reading and writing the data in the MapReduce cluster, indicating that even using an MPI cluster for the computation would not greatly reduce running time. Second, MapReduce systems like Hadoop provide transparent fault-tolerance, which is a major benefit over standard MPI systems. Other MapReduce implementations, such as Phoenix++ [18], LEMOMR [9], and MRMPI [15], often store data in memory and may be a great deal faster; although, they usually lack the automatic fault tolerance. Third, the Hadoop computation engine handles all details of the distributed input-output routines, which greatly simplifies the resulting programs.

For the majority of our implementations, we use Hadoop streaming and the Python-based Dumbo MapReduce interface [2]. These programs are concise, straightforward, and easy-to-adapt to new applications. We have also investigated C++ and Java implementations, but these programs offered only mild speedups (around 2-fold), if any. The Python implementation uses about 70 lines of code, while the C++ implementation uses about 600 lines of code.

B. Success metrics

Our two success metrics are speed and stability. The differences in speed are examined in Sec. IV-B. To analyze the performance, we construct a performance model for the MapReduce cluster. After fitting two parameters to the performance of the cluster, it predicts the runtime to within a factor of two. We study stability in an expanded online version of this manuscript.[1] These results show that only our new direct TSQR method produces a matrix Q that is numerically orthogonal.

[1] Available from http://arxiv.org/abs/1301.1071.

II. Indirect QR factorizations in MapReduce

One of the first papers to explicitly discuss the QR factorization on MapReduce architectures was written by Constantine and Gleich [5]; however many had studied methods for *linear regression* and *principal components analysis* in MapReduce [3]. These methods all bear a close resemblance to the Cholesky QR algorithm we describe next.

A. Cholesky QR

The Cholesky factorization of an $n \times n$ symmetric positive definite real-valued matrix A is:

$$A = LL^T$$

where L is an $n \times n$ lower triangular matrix. Note that, for any A that is full rank, A^TA is symmetric positive definite. The Cholesky factor L for the matrix A^TA is exactly the matrix R^T in the QR factorization as the following derivation shows. Let $A = QR$. Then

$$A^TA = (QR)^TQR = R^TQ^TQR = R^TR.$$

Since R is upper triangular and L is unique, $R^TR = LL^T$. The method of computing R via the Cholesky decomposition of A^TA matrix is called *Cholesky QR*.

Thus, the problem of finding R becomes the problem of computing A^TA. This task is straightforward in MapReduce. In the map stage, each task collects rows – recall that these are key-values pairs – to form a local matrix A_i and then computes $A_i^TA_i$. These matrices are small, $n \times n$, and are output by row. In fact, $A_i^TA_i$ is symmetric, and there are ways to reduce the computation by utilizing this symmetry. We do not exploit them because disk access time dominates the computation; a more detailed performance discussion is in Sec. IV. In the reduce stage, each individual reduce function takes in multiple instances of each row of A^TA from the mappers. These rows are summed to produce a row of A^TA. Formally, this method computes:

$$A^TA = \sum_{i=1}^{\#(\text{map tasks})} A_i^TA_i$$

where A_i is the input to each map-task.

Extending the A^TA computation to Cholesky QR simply consists of gathering all rows of A^TA on one processor and serially computing the Cholesky factorization $A^TA = LL^T$. The serial Cholesky factorization is fast since A^TA is small, $n \times n$. The Cholesky QR MapReduce algorithm is illustrated in Fig. 1.

It is important to note the architecture limitation due to the number of columns, n. The number of keys emitted by each map task is exactly n: 0, 1, ... $n-1$ (one for each row of $A_i^TA_i$), and the total number of unique keys passed to the reduction stage is n. Thus, the row sum reduction stage can use at most n tasks.

Alternatively, the reduce function can emit a key-value pair where the key represents the row and column index

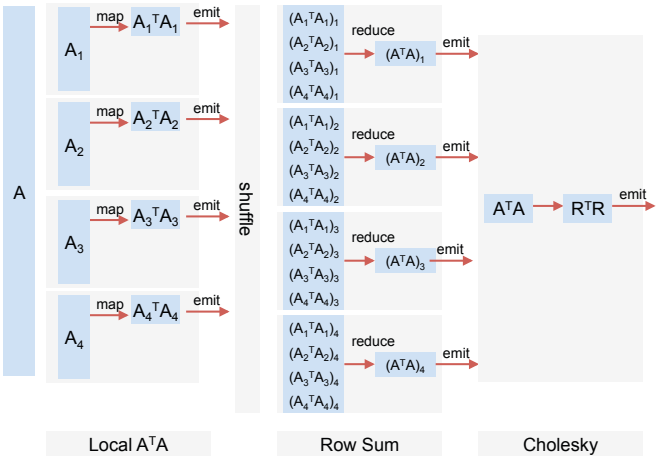

Fig. 1. MapReduce Cholesky QR computation for a matrix A with 4 columns.

of a given entry of $A_i^T A_i$, and the value is the given entry. This increases the number of unique keys to n^2 (or, by taking symmetry into account, $n(n-1)$). It is also valid to use more general reduction trees where partial row sums are computed on all the processors, and a reduction to n processors accumulates the partial row sums. The cost of this more general tree is the startup time for another map and reduce iteration. Typically, the extra startup time outweighs the benefit of additional parallelism.

Each of these variations of Cholesky QR can be described by our performance model in Sec. IV-A. For experiments, we use a small cluster (where at most 40 reduce tasks are available), and these design choices have little effect on the running times. We use the implementation where the reduce function takes in rows of $A^T A$ as it is the simplest.

B. Indirect TSQR

One of the problems with Cholesky QR is that the matrix $A^T A$ has the *square* of the condition number of the matrix A. This suggests that finite precision computations with $A^T A$ will not always produce an accurate R matrix. For this reason, Constantine and Gleich studied a succinct MapReduce implementation [5] of the communication-avoiding TSQR algorithm by Demmel et al. [8], where map and reduce tasks both compute local QR computations. This method is known to be numerically stable [8] and was recently shown to have superior stability to many standard algorithms [13]. Constantine and Gleich's initial implementation is only designed to compute R. We will refer to this method as "Indirect TSQR", because Q may be computed indirectly with $Q = AR^{-1}$. In Sec. III, we extend this method to also compute Q in a stable manner.

We will now briefly review the Indirect TSQR algorithm and its implementation to facilitate the explanation of the more intricate direct version. Let A be a matrix that – for simplicity of explanation – has $8n$ rows and n columns, which is partitioned across four map tasks as:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix}.$$

Each map task computes a local QR factorization:

$$A = \underbrace{\begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}}_{4n \times n}.$$

The matrix of stacked upper triangular matrices on the right is then passed to a reduce task and factored into $\tilde{Q}\tilde{R}$. At this point, we have the QR factorization of A in product form:

$$A = \overbrace{\underbrace{\begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\tilde{Q}}_{4n \times n}}^{=Q} \overbrace{\underbrace{\tilde{R}}_{n \times n}}^{=R}.$$

The above construction generalizes to the case that A is not partitioned evenly. If A is $m \times n$, in the first step, a QR decomposition is computed on the block of A that is streamed to a map task. The Indirect TSQR method ignores the intermediate Q factors and simply outputs the $n \times n$ factors R_i in the intermediate stage and $\tilde{R}$ in the final stage. Fig. 2 illustrates each map and reduce output. We do not need to gather all R factors onto a single task to compute $\tilde{R}$. Any reduction tree computes $\tilde{R}$ correctly. Constantine and Gleich found that using an additional MapReduce iteration to form a more parallel reduction tree could greatly accelerate the method. This finding differs from the Cholesky QR method, where additional iterations rarely helped. In the Sec. III, we show how to save the Q factors to reconstruct Q directly.

C. Computing AR^{-1}

Given the matrix R, the simplest method for computing Q is computing the inverse of R and multiplying by A, that is, computing AR^{-1}. Since R is $n \times n$ and upper-triangular, we can compute its inverse quickly. Fig. 3 illustrates how the matrix multiplication and iterative refinement step cleanly translate to MapReduce. This "indirect" method of the inverse computation is not backwards stable (for example, see [17]). Thus, a step of iterative refinement may be used to get Q within desired accuracy. However, the indirect methods may still have large errors after iterative refinement if A is ill-conditioned enough. This further motivates the use of a direct method.

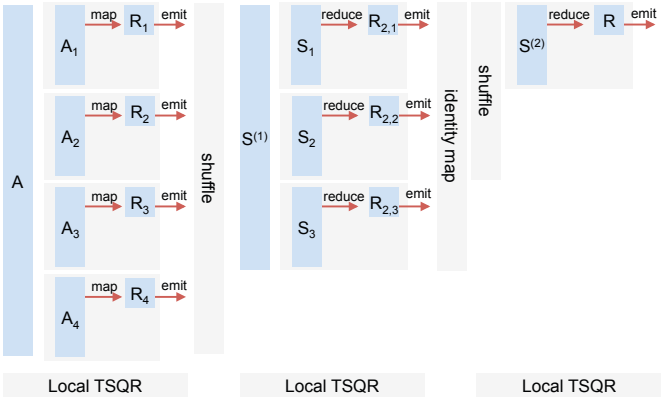

Fig. 2. MapReduce TSQR computation with a 2-stage reduction tree. $S^{(1)}$ is the matrix consisting of the rows of the R_i factors stacked on top of each other, $i = 1, 2, 3, 4$. Similarly, $S^{(2)}$ is the matrix consisting of the rows of the $R_{2,j}$ factors stacked on top of each other, $j = 1, 2, 3$.

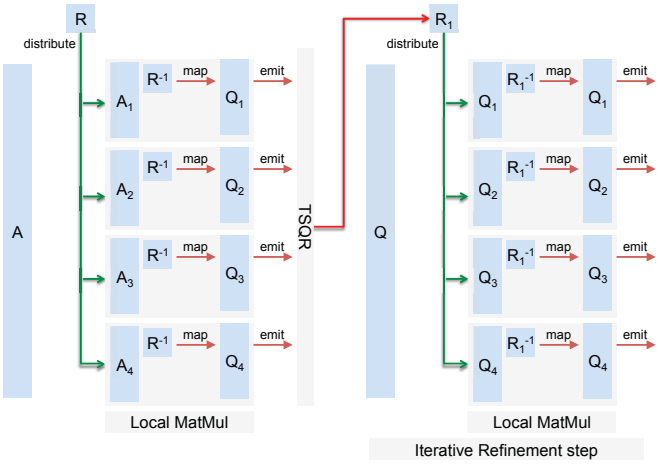

Fig. 3. Indirect MapReduce computation of Q with iterative refinement.

D. Pseudo-Iterative Refinement

A variety of fast, randomized algorithms have recently been developed for least squares problems [1], [12], [16]. A key idea from this work is that the R factor from the QR factorization of a small, random subset of the rows of A is a first-order approximation to the R factor of the entire matrix. For tall-and-skinny matrices, the number of rows sampled grows as approximately $100n \log n$.

The standard iterative refinement procedure will compute the R factor of two $m \times n$ matrices: the original matrix A and the approximate Q factor AR^{-1}. We take a sampling approach to compute the R factor of only one $m \times n$ matrix and one smaller matrix. To approximate random sampling, we read a single block of the matrix from Hadoop Distributed File System (HDFS). Given a sample of rows, A_s, we compute its R factor, R_s. We then compute the approximate Q factor via $Q_1 = AR_s^{-1}$. Next the R factor of Q_1, R_1, is computed. Finally, the refined Q factor is given by $Q_1 R_1^{-1}$. In the implementation, the computations of Q_1 and R_1 are performed simultaneously to avoid writing the disposable factor Q_1 to disk. The refined Q factor is computed by $(AR_s^{-1})R_1^{-1}$, and the R factor is given by $R_1 R_s$. We call this method *Pseudo-Iterative Refinement*. The standard iterative refinement procedure is then the special case of Pseudo-Iterative Refinement where $A_s = A$.

III. Direct QR Factorizations in MapReduce

One of the textbook algorithms to compute a stable QR factorization is the Householder QR method [10]. This method always produces a matrix Q where $\|Q^T Q - I\|_2$ is on the order of machine error. We have implemented the algorithm in MapReduce and discuss it in the online version (see previous footnote). However, the Householder method involves changing the entire matrix once for each column. Writing the updated matrix to disk is prohibitively expensive in MapReduce and our performance data showed that Householder QR is an order of magnitude slower. Thus, we begin our discussion with our new, stable algorithm, Direct TSQR.

A. Direct TSQR

We finally arrive at our proposed method. Here, we directly compute the QR decomposition of A in three steps using two map functions and one reduce function, as illustrated in Fig. 4. This avoids the iterative nature of the Householder methods but maintains the stability properties [8], [10], [13]. For an example, consider again a matrix A with $8n$ rows and n columns, which is partitioned across four map tasks for the first step:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix}.$$

The first step uses only map tasks. Each task collects data as a local matrix, computes a single QR decomposition, and emits Q and R to separate files. The factorization of A then looks as follows, with $Q_{i,1}R_i$ the computed factorization on the i-th task:

$$A = \underbrace{\begin{bmatrix} Q_{1,1} & & & \\ & Q_{2,1} & & \\ & & Q_{3,1} & \\ & & & Q_{4,1} \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}}_{4n \times n}.$$

The second step is a single reduce task. The input is the set of R factors from the first step. The R factors are collected as a matrix and a single QR decomposition is performed. The sections of Q corresponding to each R factor are emitted as values. In the following figure, $\tilde{R}$ is the final upper triangular factor in our QR decomposition

of A:

$$\underbrace{\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}}_{4n \times n} = \underbrace{\begin{bmatrix} Q_{1,2} \\ Q_{2,2} \\ Q_{3,2} \\ Q_{4,2} \end{bmatrix}}_{4n \times n} \underbrace{\tilde{R}}_{n \times n}.$$

The third step also uses only map tasks. The input is the set of Q factors from the first step. The Q factors from the second step are small enough that we distribute the data in a file to all map tasks. The corresponding Q factors are multiplied together to emit the final Q:

$$\underbrace{Q}_{8n \times n} = \underbrace{\begin{bmatrix} Q_{1,1} & & & \\ & Q_{2,1} & & \\ & & Q_{3,1} & \\ & & & Q_{4,1} \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} Q_{1,2} \\ Q_{2,2} \\ Q_{3,2} \\ Q_{4,2} \end{bmatrix}}_{4n \times n} = \underbrace{\begin{bmatrix} Q_{1,1}Q_{1,2} \\ Q_{2,1}Q_{2,2} \\ Q_{3,1}Q_{3,2} \\ Q_{4,1}Q_{4,2} \end{bmatrix}}_{8n \times n}$$

$$A = Q\tilde{R}.$$

One implementation challenge is matching the Q and R factors to the tasks on which they are computed. In the first step, the key-value pairs emitted use a unique map task identifier (e.g., via the `uuid` package in Python) as the key and the Q or R factor as the value. The reduce task in the second step maintains an ordered list of the keys read. The k-th key in the list corresponds to rows $(k-1)n+1$ to kn of the locally computed Q factor. The map tasks in the third step parse a data file containing the Q factors from the second step, and this redundant parsing allows us to skip the *shuffle* and *reduce*. Another implementation challenge is that the map tasks in the first step and the reduce task in the second step must emit the Q and R factors to separate files. For this functionality, we use the `feathers` extension of Dumbo.

The thin singular value decomposition (SVD) of an $m \times n$ real-valued matrix is A is:

$$A = U\Sigma V^T$$

where U is an $m \times n$ orthogonal matrix, Σ is a diagonal matrix with decreasing, non-negative entries on the diagonal, and V is an $n \times n$ orthogonal matrix. To compute the SVD of A, we modify the second step and add a fourth step. In the second step, we also compute $R = U\Sigma V^T$. Then $A = (QU)\Sigma V^T$ is the SVD of A. Since R is $n \times n$, computing its SVD is cheap. The fourth step computes QU. If Q is not needed, i.e., only the singular vectors are desired, then we can pass U to the third step and compute QU directly without writing Q to disk. In this case, the SVD uses the same number of passes over the data as the QR factorization. If only the singular values are needed, then only the first two steps of the algorithm are needed along with the SVD of R. However, in this case, it would be favorable to use the TSQR implementation from Sec. II-B to compute R only.

B. Extending Direct TSQR to a recursive algorithm

A central limitation to the Direct TSQR method is the necessity of gathering all R factors from the first step onto one reduce task in the second step. As the matrix becomes fatter, this serial bottleneck becomes limiting. We can cope by recursively extending the method and repeating the computation on the output R from the first step. The algorithm is outlined in Alg. 1, and the performance benefits of the algorithms are empirically analyzed in Sec. IV-C.

Algorithm 1 Recursive extension of Direct TSQR

```
function DirectTSQR(matrix A)
    Q1, R1 = FirstStep(A)
    if R1 is too big then
        Assign keys to rows of R1
        Q2 = DirectTSQR(R1)
    else
        Q2 = SecondStep(R1)
    end if
    Q = ThirdStep(Q1, Q2)
    return Q
end function
```

IV. Performance Experiments

We evaluate performance in two ways. First, we build a performance model for our methods based on how much data is read and written by the MapReduce cluster. Second, we evaluate the implementations on a 10-node, 40-core MapReduce cluster at Stanford's Institute for Computational and Mathematical Engineering (ICME). Each node has 6 2-TB disks, 24 GB of RAM, and a single Intel Core i7-960 3.2 GHz processor. They are connected via Gigabit ethernet. After fitting only two parameters – the read and write bandwidth – the performance model predicts the actual runtime within a factor of two.

Although the cluster is small, we emphasize that the algorithms scale with the number of map tasks launched, *not* the number of nodes. Therefore, these algorithms scale to larger clusters. This is covered in detail in our performance model, and the numbers of map tasks used by the algorithms are listed in Table III.

All matrices used in the experiments are synthetic. The matrix dimensions are chosen to reflect problems in model reduction [6] and fast robust linear regression [4].

We do not perform standard parallel scaling studies due to how the Hadoop framework integrates the computational engine with the distributed filesystem. This combination makes these measurements difficult without rebuilding the cluster for each experiment.

A. Performance model

Since the QR decomposition algorithms have more output data (Q and R factors) than input data (the matrix A), we choose a performance model that emphasizes read

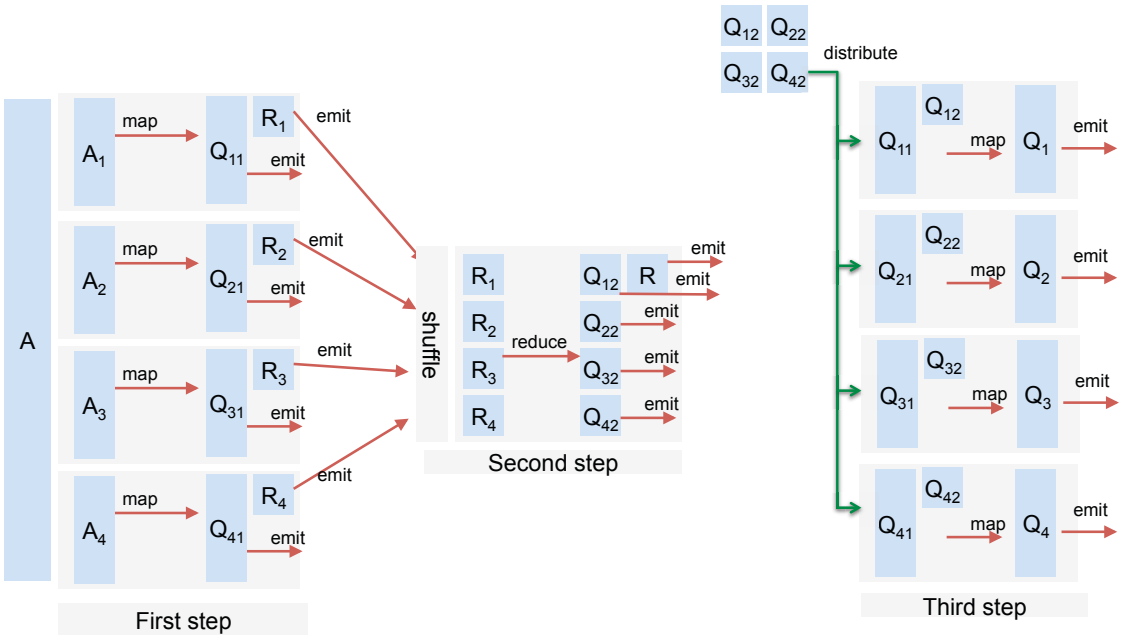

Fig. 4. Direct MapReduce computation of Q and R.

and write volume. Our performance model targets Hadoop so that we can more accurately evaluate the performance of the algorithms for a popular MapReduce framework.

Let $\mathcal{M}_j$ and $\mathcal{R}_j$ be the number of map and reduce tasks for step j, respectively. Let $\mathcal{M}_{\max}$ and $\mathcal{R}_{\max}$ be the maximum number of map and reduce tasks that can run concurrently on the cluster. Both $\mathcal{M}_{\max}$ and $\mathcal{R}_{\max}$ are fixed in the Hadoop configuration, and $\mathcal{M}_{\max} + \mathcal{R}_{\max}$ is usually at least the total number of cores. Let k_j be the number of distinct input keys passed to the reduce tasks for step j. We define the map parallelism for step j as $p_j^m = \min\{\mathcal{M}_{max}, \mathcal{M}_j\}$ and the reduce parallelism for step j as $p_j^r = \min\{\mathcal{R}_{max}, \mathcal{R}_j, k_j\}$. Let R_j^m, W_j^m be the amount of data read and written in the j-th map step, by all map tasks, respectively. We have analogous definitions for R_j^r and W_j^r for the j-th reduce step. Finally, let β_r and β_w be the inverse read and write bandwidth, respectively. After computing β_r and β_w, we can provide a lower bound for the algorithm by counting disk reads and writes. The lower bound for a job with N iterations is:

$$T_{lb} = \sum_{j=1}^{N} \frac{R_j^m \beta_r + W_j^m \beta_w}{p_j^m} + \frac{R_j^r \beta_r + W_j^r \beta_w}{p_j^r}.$$

We use streaming benchmarks to estimate β_r and β_w for the 40-core ICME cluster, and the results are in Table I. On this cluster, $\mathcal{M}_{max} = \mathcal{R}_{max} = 40$. Table II provides the number of reads and writes for our algorithms, and Tables III and IV provide the information for computing p_j^m and p_j^r. The keys for the matrix row identifiers are 32-byte strings. The computed lower bounds for our algorithms are in Table V. In Sec. IV-B, we examine how close the implementations are to the lower bounds.

TABLE III
VALUES OF $\mathcal{M}_j$ AND $\mathcal{R}_j$ NEEDED TO COMPUTE p_j^m AND p_j^r. $\mathcal{M}_1$, $\mathcal{M}_3$, AND $\mathcal{M}_5$ ARE DEPENDENT ON THE MATRIX SIZE. OTHER LISTED DATA ARE NOT. THE VALUES $\mathcal{M}_j$ AND $\mathcal{R}_j$ FOR CHOLESKY AND INDIRECT TSQR ARE THE SAME.

Matrix Dimensions		Cholesky, Indir. TSQR	+PIR	+IR	Direct
$4.0B \times 4$	$\mathcal{M}_1$	1200	3	1200	2000
$2.5B \times 10$		1680	7	1680	2640
$600M \times 25$		1200	3	1200	1600
$500M \times 50$		1920	3	1920	2560
$150M \times 100$		880	44	880	880
	$\mathcal{M}_2$	$\mathcal{M}_{\max}$	1	$\mathcal{M}_{\max}$	$\mathcal{M}_{\max}$
$4.0B \times 4$	$\mathcal{M}_3$	1200	1200	1200	2000
$2.5B \times 10$		1680	1680	1680	2640
$600M \times 25$		1200	1200	1200	1600
$500M \times 50$		1920	1920	1920	2560
$150M \times 100$		880	880	880	880
	$\mathcal{M}_4$	–	$\mathcal{M}_{\max}$	$\mathcal{M}_{\max}$	$\mathcal{M}_{\max}$
$4.0B \times 4$	$\mathcal{M}_5$	–	1200	1200	–
$2.5B \times 10$		–	1680	1680	–
$600M \times 25$		–	1200	1200	–
$500M \times 50$		–	1920	1920	–
$150M \times 100$		–	880	880	–
	$\mathcal{R}_1$	$\mathcal{R}_{\max}$	$\mathcal{R}_{\max}$	$\mathcal{R}_{\max}$	$\mathcal{R}_{\max}$
	$\mathcal{R}_2$	1	1	1	1
	$\mathcal{R}_3$	–	$\mathcal{R}_{\max}$	$\mathcal{R}_{\max}$	–
	$\mathcal{R}_4$	–	1	1	–

B. Algorithmic comparison

Using one step of iterative refinement yields numerical errors that are acceptable in a vast majority of cases. In these cases, performance is our motivator for algorithm choice. Tables VI and VII show performance results of

TABLE I

STREAMING TIME TO READ FROM AND WRITE TO DISK. PERFORMANCE IS IN INVERSE BANDWIDTH, SO LARGER β_r AND β_w MEANS SLOWER STREAMING. THE STREAMING BENCHMARKS ARE PERFORMED WITH $\mathcal{M}_{\max}$ MAP TASKS.

Rows	Cols.	HDFS Size (GB)	read+write (secs.)	read (secs.)	$\beta_r/\mathcal{M}_{max}$ (s/GB)	$\beta_w/\mathcal{M}_{max}$ (s/GB)
4,000,000,000	4	134.6	713	305	2.2660	3.0312
2,500,000,000	10	193.1	909	309	1.6002	3.1072
600,000,000	25	112.0	526	169	1.5089	3.1875
500,000,000	50	183.6	848	253	1.3780	3.2407
150,000,000	100	109.4	529	151	1.3803	3.4552

TABLE II

NUMBER OF READS AND WRITES AT EACH STEP (IN BYTES). WE ASSUME A DOUBLE IS 8 BYTES AND K IS THE NUMBER OF BYTES FOR A ROW KEY ($K = 32$ IN OUR EXPERIMENTS). THE AMOUNT OF KEY DATA IS SEPARATED FROM THE AMOUNT OF VALUE DATA. FOR EXAMPLE, $8mn + Km$ IS Km BYTES IN KEY DATA AND $8mn$ BYTES IN VALUE DATA. FOR ITERATIVE REFINEMENT, p_{samp} IS THE PROBABILITY OF SAMPLING A ROW. $p_{samp} = 1$ FOR STANDARD ITERATIVE REFINEMENT.

	Cholesky	Cholesky + I.R	Indirect TSQR	Indirect TSQR + I.R.	Direct TSQR
R_1^m	$8mn + Km$	$p_{samp}(8mn + Km)$	$8mn + Km$	$p_{samp}(8mn + Km)$	$8mn + Km$
W_1^m	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8mn + 8\mathcal{M}_1n^2 + Km + 64\mathcal{M}_1$
R_1^r	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	$8\mathcal{M}_1n^2 + 8\mathcal{M}_1n$	0
W_1^r	$8n^2 + 8n$	$8n^2 + 8n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	0
R_2^m	$8n^2 + 8n$	$8n^2 + 8n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{M}_1n^2 + K\mathcal{M}_1$
W_2^m	$8n^2 + 8n$	$8n^2 + 8n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{M}_1n^2 + K\mathcal{M}_1$
R_2^r	$8n^2 + 8n$	$8n^2 + 8n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{R}_1n^2 + 8\mathcal{R}_1n$	$8\mathcal{M}_1n^2 + K\mathcal{M}_1$
W_2^r	$8n^2 + 8n$	$8n^2 + 8n$	$8n^2 + 8n$	$8n^2 + 8n$	$8\mathcal{M}_1n^2 + 32\mathcal{M}_1 + 8n^2 + 8n$
R_3^m	$8mn + Km$ $+ \mathcal{M}_3(8n^2 + 8n)$	$8mn + Km$ $+ \mathcal{M}_3(8n^2 + 8n)$	$8mn + Km$ $+ \mathcal{M}_3(8n^2 + 8n)$	$8mn + Km$ $+ \mathcal{M}_3(8n^2 + 8n)$	$8mn + Km$ $+ \mathcal{M}_3(8\mathcal{M}_1n^2 + 64\mathcal{M}_1)$
W_3^m	$8mn + Km$	$8\mathcal{M}_3n^2 + 8\mathcal{M}_3n$	$8mn + Km$	$8\mathcal{M}_3n^2 + 8\mathcal{M}_3n$	$8mn + Km$
R_3^r	0	$8\mathcal{M}_3n^2 + 8\mathcal{M}_3n$	0	$8\mathcal{M}_3n^2 + 8\mathcal{M}_3n$	0
W_3^r	0	$8n^2 + 8n$	0	$8\mathcal{R}_3n^2 + 8\mathcal{R}_3n$	0
R_4^m	–	$8n^2 + 8n$	–	$8\mathcal{R}_3n^2 + 8\mathcal{R}_3n$	–
W_4^m	–	$8n^2 + 8n$	–	$8\mathcal{R}_3n^2 + 8\mathcal{R}_3n$	–
R_4^r	–	$8n^2 + 8n$	–	$8\mathcal{R}_3n^2 + 8\mathcal{R}_3n$	–
W_4^r	–	$8n^2 + 8n$	–	$8n^2 + 8n$	–
R_5^m	–	$8mn + Km$ $+2\mathcal{M}_5(8n^2 + 8n)$	–	$8mn + Km$ $2\mathcal{M}_5(8n^2 + 8n)$	–
W_5^m	–	$8mn + Km$	–	$8mn + Km$	–
R_5^r	–	0	–	0	–
W_5^r	–	0	–	0	–

TABLE V

COMPUTED LOWER BOUNDS FOR EACH ALGORITHM. p_{samp} IS THE SAMPLING PROBABILITY FOR PSEUDO-ITERATIVE REFINEMENT.

Rows	Cols.	p_{samp}	Cholesky	Indirect TSQR	Cholesky +PIR	Indirect TSQR+PIR	Cholesky +IR	Indirect TSQR+IR	Direct TSQR
			T_{lb} (secs.)						
4,000,000,000	4	0.0025	1803	1803	1821	1821	2343	2343	2528
2,500,000,000	10	0.0042	1645	1645	1655	1655	2062	2062	2464
600,000,000	25	0.0025	804	804	812	812	1000	1000	1237
500,000,000	50	0.0016	1240	1240	1250	1250	1517	1517	2103
150,000,000	100	0.0500	723	723	735	735	884	884	1217

TABLE VI

TIMES TO COMPUTE QR ON A VARIETY OF MATRICES WITH SEVEN MAPREDUCE ALGORITHMS; ONLY THE DIRECTTSQR METHOD IS GUARANTEED TO BE NUMERICALLY STABLE.

Rows	Cols.	HDFS Size (GB)	Cholesky	Indirect TSQR	Cholesky +PIR	Indirect TSQR+PIR	Cholesky +IR	Indirect TSQR+IR	Direct TSQR
			job time (secs.)						
4,000,000,000	4	134.6	2931	3460	3276	3620	4365	4741	6128
2,500,000,000	10	193.1	2508	2509	2887	3354	3778	4034	4035
600,000,000	25	112.0	1098	1104	1275	1476	1645	2006	1910
500,000,000	50	183.6	1563	1618	1772	1960	2216	2655	3090
150,000,000	100	109.6	1023	1127	1146	1304	1400	1652	2076

TABLE IV
VALUES OF k_j NEEDED TO COMPUTE p_j^m AND p_j^r.

	Chol.	Chol. +PIR,+IR	Indir. TSQR	Indir. TSQR +PIR,+IR	Direct
k_1	n	n	$\mathcal{M}_1 n$	$\mathcal{M}_1 n$	$\mathcal{M}_1$
k_2	n	n	$\mathcal{M}_2 n$	$\mathcal{M}_2 n$	$\mathcal{M}_1$
k_3	0	n	0	$\mathcal{M}_3 n$	0
k_4	–	n	–	$\mathcal{M}_4 n$	–
k_5	–	0	–	0	–

TABLE VIII
FRACTION OF TIME SPENT IN EACH STEP OF THE DIRECT TSQR ALGORITHM (FRACTIONS MAY NOT SUM TO 1 DUE TO ROUNDING).

Rows	Cols.	Step 1	Step 2	Step 3
4,000,000,000	4	0.72	0.02	0.26
2,500,000,000	10	0.61	0.04	0.34
600,000,000	25	0.56	0.06	0.38
500,000,000	50	0.55	0.07	0.39
150,000,000	100	0.47	0.15	0.38

Cholesky QR and the the Indirect and Direct TSQR methods for a variety of matrices.

In our experiments, we see that Indirect TSQR and Cholesky QR provide the fastest ways of computing the Q and R factors, albeit $\|Q^T Q - I\|_2$ may be large. For all matrices with greater than four columns, these two methods have similar running times. For such matrices, the majority of the running time is the AR^{-1} step, and this step is identical between the two methods. This is precisely because the write bandwidth is less than the read bandwidth.

For the matrices with 10, 25, and 50 columns, Direct TSQR is competitive with the indirect methods with iterative refinement, albeit slightly slower. The performance is the most similar for smaller number of columns (e.g., with ten columns). However, when the matrix becomes too skinny (e.g., with four columns), Cholesky QR with iterative refinement is a better choice. When the matrix becomes too fat (e.g., with 100 columns), the local gather in Step 2 becomes expensive. Table VIII shows the amount of time spent in each step of the Direct TSQR computation. Indeed, Step 2 consumes a larger fraction of the running time as the number of columns increases.

Table IX shows how each algorithm performs compared to its lower bound from Table V. We see that Direct TSQR diverges from this bound when the number of columns is too small. To explain this difference, we note that Direct TSQR must gather all the keys and values in the first step before performing any computation. When the number of key-value pairs is large, e.g., with the 4,000,000,000 $\times$ 4 matrix, then this step becomes limiting and this is not accounted for by our performance model. Thus, the model predicts the runtime of Cholesky QR and Indirect TSQR with iterative refinement more accurately than Direct TSQR. Although their lower bounds are greater, the empirical performance makes these algorithms more attractive as the number of columns increases. If guaranteed stability is required, Direct TSQR is the best method, and the performance cost of stability is quite small for a modest number of columns.

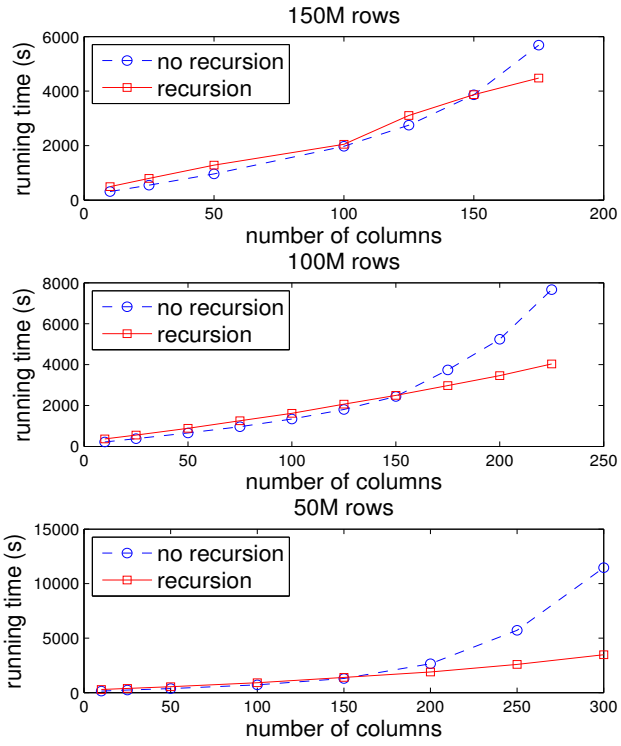

Fig. 5. Running time of Direct TSQR with and without recursion. The recursive version takes only one recursive step.

C. Recursive Direct TSQR

In the preceding performance analysis, we used Direct TSQR without the recursive extension described in Sec. III-B. Fig. 5 shows the performance benefits for the recursive extension as the number of columns increases for matrices with 50, 100, and 150 million rows. In these experiments, a single recursive step is taken. For these matrices, the recursive version of the algorithm is faster once the number of columns is approximately 150.

V. CONCLUSION

If numerical stability is required, the Direct TSQR method discussed in this paper is the best choice of algorithm. It is guaranteed to produce a numerically orthogonal matrix. It usually takes no more than twice the time of the fastest, but unstable method, and it is often competitive with conceptually simpler methods.

Our code for this paper is openly available, see:

https://github.com/arbenson/mrtsqr

This software runs on any system with Hadoop streaming.

In the future we plan to investigate mixed MPI and Hadoop code. The idea is that once all the local mappers

TABLE VII
FLOATING POINT OPERATIONS PER SECOND ON A VARIETY OF MATRICES WITH FOUR MAPREDUCE ALGORITHMS.

Rows	Cols.	$2*rows*cols^2$	Cholesky	Indirect TSQR	Cholesky +PIR	Indirect TSQR+PIR	Cholesky +IR	Indirect TSQR+IR	Direct TSQR
			$2*rows*cols^2$/sec						
4,000,000,000	4	1.280e+11	4.367e+07	3.140e+07	3.907e+07	3.536e+07	2.932e+07	2.700e+07	2.089e+07
2,500,000,000	10	5.000e+11	1.994e+08	1.993e+08	1.732e+08	1.491e+08	1.323e+08	1.239e+08	1.239e+08
600,000,000	25	7.500e+11	6.831e+08	6.793e+08	5.882e+08	5.081e+08	4.559e+08	3.739e+08	3.927e+08
500,000,000	50	2.500e+12	1.599e+09	1.545e+09	1.411e+09	1.276e+09	1.128e+09	9.416e+08	8.091e+08
150,000,000	100	3.000e+12	2.933e+09	2.662e+09	2.643e+09	2.338e+09	2.143e+09	1.836e+09	1.393e+09

TABLE IX
PERFORMANCE OF ALGORITHMS AS A MULTIPLE OF THE LOWER BOUNDS FROM TABLE V.

Rows	Cols.	Cholesky	Indirect TSQR	Cholesky +PIR	Indirect TSQR+PIR	Cholesky +IR	Indirect TSQR+IR	Direct TSQR
		multiple of T_{lb}						
4,000,000,000	4	1.626	2.261	1.799	1.988	1.863	2.023	2.424
2,500,000,000	10	1.525	1.525	1.744	2.027	1.832	1.956	1.638
600,000,000	25	1.366	1.373	1.570	1.818	1.645	2.006	1.544
500,000,000	50	1.260	1.305	1.418	1.568	1.461	1.750	1.469
150,000,000	100	1.415	1.559	1.544	1.746	1.584	1.848	1.770

have run in the first step of the Direct TSQR method, the resulting R_i matrices constitute a much smaller input. If we run a standard, in-memory MPI implementation to compute the QR factorization of this smaller matrix, then we could remove two iterations from the direct TSQR method. Also, we would remove much of the disk IO associated with saving the Q_i matrices. These changes could reduce runtime by at most a factor of 4.

ACKNOWLEDGMENT

Austin R. Benson is supported by an Office of Technology Licensing Stanford Graduate Fellowship.

David F. Gleich is supported by a DOE CSAR grant.

We acknowledge funding from Microsoft (award 024263) and Intel (award 024894), and matching funding by UC Discovery (award DIG07-10227), with additional support from ParLab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung, and support from MathWorks. We also acknowledge the support of the US DOE (grants DE-SC0003959, DE-SC0004938, DE-SC0005136, DE-SC0008700, DE-AC02-05CH11231) and DARPA (award HR0011-12-2-0016).

We are grateful to ICME for letting us use their MapReduce cluster for these computations.

We are grateful to Paul Constantine for working on the initial TSQR method and for continual discussions about using these routines in simulation data analysis problems.

REFERENCES

[1] H. Avron, P. Maymounkov, and S. Toledo. Blendenpik: Supercharging LAPACK's least-squares solver. *SIAM J. Sci. Comput.*, 32(3):1217–1236, Apr. 2010.

[2] K. Bosteels. Dumbo. http://klbostee.github.io/dumbo/, 2012.

[3] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2006.

[4] K. L. Clarkson, P. Drineas, M. Magdon-Ismail, M. W. Mahoney, X. Meng, and D. P. Woodruff. The fast Cauchy transform and faster robust linear regression. In *SODA*, pages 466–477, 2013.

[5] P. Constantine and D. F. Gleich. Tall and skinny QR factorizations in MapReduce architectures. In *MAPREDUCE2012*, page 43.50, 2011.

[6] P. G. Constantine, D. F. Gleich, Y. Hou, and J. Templeton. Model Reduction with MapReduce-enabled Tall and Skinny Singular Value Decomposition. *arXiv*, math.NA:1306.4690, June 2013.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI2004)*, pages 137–150, 2004.

[8] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *EECS-2008-89*, Aug. 2008.

[9] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Benchmarking MapReduce implementations for application usage scenarios. GRID '11, pages 90–97, 2011.

[10] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, October 1996.

[11] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[12] M. W. Mahoney. Randomized algorithms for matrices and data. *arXiv*, cs.DS, 2011.

[13] D. Mori, Y. Yamamoto, and S.-L. Zhang. Backward error analysis of the AllReduce algorithm for Householder QR decomposition. *Jpn. J. Ind. Appl. Math.*, 29(1):111–130, Feb. 2012.

[14] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA, USA, 1998.

[15] S. J. Plimpton and K. D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, 2011.

[16] V. Rokhlin and M. Tygert. A fast randomized algorithm for overdetermined linear least-squares regression. *Proceedings of the National Academy of Sciences*, 105(36):13212–13217, 2008.

[17] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23:2165–2182, June 2001.

[18] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. MAPREDUCE 2011, pages 9–16, New York, NY, USA, 2011. ACM.

[19] Various. Hadoop version 0.21. http://hadoop.apache.org, 2012.

[20] J. Zhao and J. Pjesivac-Grbovic. MapReduce: The programming model and practice. http://research.google.com/archive/papers/mapreduce-sigmetrics09-tutorial.pdf, 2009. Tutorial.