

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

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

David F. Gleich  
Department of Computer Science  
Purdue University  
dgleich@purdue.edu

James Demmel  
Computer Sciences Division and  
Department of Mathematics  
University of California, Berkeley  
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, a standard unstable implementation for MapReduce (Cholesky QR), and the classic stable algorithm implemented for MapReduce (Householder QR). We find that our new stable method has a large performance advantage over the Householder QR method. 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. We call a matrix tall-and-skinny if it has many more rows than columns ( $m \gg n$ ). 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 [6]. Current tall-and-skinny QR methods for MapReduce provide *only* a fast way to compute  $R$  [5]. (The details of these are described further in Sec. II.) In order to

compute the matrix  $Q$ , they use the indirect formulation:

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

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, although, a step of iterative refinement can sometimes be used to produce a  $Q$  factor with acceptable accuracy [15]. (Iterative refinement is the process of repeating the QR decomposition on the computed  $Q$  factor.) However, if a matrix is sufficiently ill-conditioned, iterative refinement will still result in a large error measured by  $\|Q^T Q - I\|_2$  (see Sec. IV). We shall describe a numerically stable method (Sec. III) that computes  $Q$  and  $R$  directly and faster than performing the refinement of the indirect computation for some matrices.

Sec. V-A describes a performance model for our algorithms, which allows us to compute lower bounds on running times. The algorithms are almost always within a factor of two of the lower bounds (Sec. V-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 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 [20] – 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

Table I

THE PERFORMANCE IMPROVEMENT OF C++ OVER PYTHON FOR OUR DIRECT TSQR ON A 10-NODE MAPREDUCE CLUSTER IS ONLY MILD.

Rows	Cols.	Job time (secs.)	Speedup
4,000,000,000	4	2217	2.76
2,500,000,000	10	3137	1.29
600,000,000	25	1482	1.29
500,000,000	50	1477	2.09
150,000,000	100	1503	1.43

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 [21] 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. Second, MapReduce systems like Hadoop provide transparent fault-tolerance, which is a major benefit over standard MPI systems. Other MapReduce implementations, such as Twister [9], Phoenix++ [18], LEMOMR [10], and MRMPI [16], 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. See Table I for a comparison against C++. 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. V-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. For stability, we use the metric  $\|A - QR\|_2 / \|R\|_2$  to measure the accuracy of the decomposition and  $\|Q^T Q - I\|_2$  to measure the orthogonality of the computed  $Q$  factor. Small scale simulations of the MapReduce algorithms show that, regardless of the algorithm,  $\|A - QR\|_2 / \|R\|_2$  is  $O(\epsilon)$  where  $\epsilon$  is the machine precision. However,  $\|Q^T Q - I\|_2$  varies dramatically based on the algorithm, but is always

$O(\epsilon)$  for our new direct TSQR method. We examine these differences in Sec. IV.

## 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 [4]. 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,  $A^T A$  is symmetric positive definite. The Cholesky factor  $L$  for the matrix  $A^T A$  is exactly the matrix  $R$  in the QR factorization as the following derivation shows. Let  $A = QR$ . Then

$$A^T A = (QR)^T QR = R^T Q^T QR = R^T R.$$

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

Thus, the problem of finding  $R$  becomes the problem of computing  $A^T A$ . 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  $A_p$  and then computes  $A_p^T A_p$ . These matrices are small,  $n \times n$ , and are output by row. In fact,  $A_p^T A_p$  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. V. In the reduce stage, each individual reduce function takes in multiple instances of each row of  $A^T A$  from the mappers. These rows are summed to produce a row of  $A^T A$ . Formally, this method computes:

$$A^T A = \sum_{p=1}^P A_p^T A_p$$

where  $A_i$  is the input to each map-task. Alg. 1 explicitly shows how this is done with key-value pairs in a MapReduce architecture.

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

---

**Algorithm 1** Compute  $A^T A$  in MapReduce

---

```
function MAP(key  $k$ , val  $a$ )  
  for  $i$ , row in enumerate( $a^T a$ ) do  
    emit( $i$ , row)  
  end for  
end function  
  
function REDUCE(key  $k$ ,  $\langle$  vals  $v_j^k$   $\rangle$ )  
  emit( $k$ , sum( $\langle v_j^k \rangle$ ))  
end function
```

---

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  $n$ :  $0, 1, \dots, n-1$  (one for each row of  $A_p^T A_p$ ), 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 of a given entry of  $A_p^T A_p$ , 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 is more expensive than the performance penalty of having less parallelism.

Each of these variations of Cholesky QR can be described by our performance model in Sec. V-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 described in Alg. 1 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 TSQR algorithm by Demmel et al. [7], where map and reduce tasks both compute local QR computations. This method is known to be numerically stable [7] and was recently shown to have superior stability to many standard algorithms [14]. 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 the following section, 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 with  $8n$  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:

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

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 next section, 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.

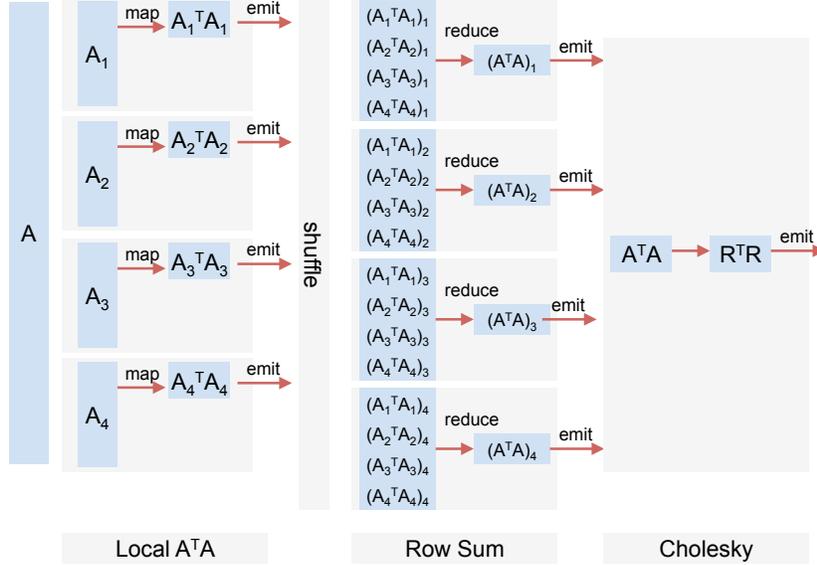


Figure 1. MapReduce Cholesky QR computation for a matrix  $A$  with 4 columns.

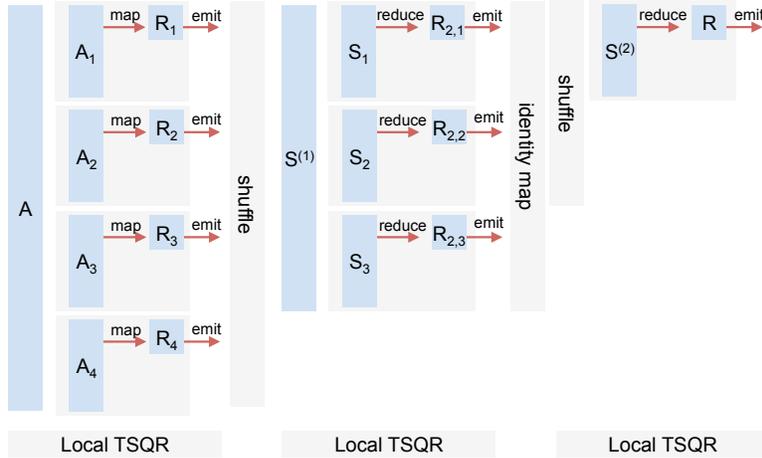


Figure 2. MapReduce TSQR computation.  $S^{(1)}$  is the matrix consisting of the rows of the  $R_i$  factors stacked on top of each other,  $i = 1, 2, 3, 4$ . Similarly,  $S^{(2)}$  is the matrix consisting of the rows of the  $R_{2,j}$  factors stacked on top of each other,  $j = 1, 2, 3$ .

### III. DIRECT QR FACTORIZATIONS IN MAPREDUCE

One of the textbook algorithms to compute a stable QR factorization is the Householder QR method [11]. This method always produces a matrix  $Q$  where  $\|Q^T Q - I\|_2$  is on the order of machine error. We begin our discussion by explaining how to implement this method in MapReduce.

#### A. Householder QR

The Householder QR algorithm [19] is not as friendly to MapReduce as either Cholesky QR or Indirect TSQR. One reason for this phenomena is the iterative nature of the algorithm. At each step of the algorithm, the matrix  $A$  is completely updated. In MapReduce, this

means we must constantly rewrite the matrix on disk. Conceptually, each step of the Householder QR method corresponds to three MapReduce calls. These are illustrated in Fig. 4. The first step of the algorithm computes the norm of a column of  $A$  to help form the Householder reflector. The second and third steps of the algorithm update the matrix with  $A \leftarrow A - 2v(A^T v)^T$ , where  $v$  is the Householder reflector. However, in the actual implementation, the first and third steps are combined because we can compute the norm for the next step immediately after updating the matrix in the third step.

Thus, the MapReduce Householder QR algorithm uses  $2n$  passes over the data for a matrix  $A$  with  $n$  columns. Every other pass requires rewriting the matrix on disk.

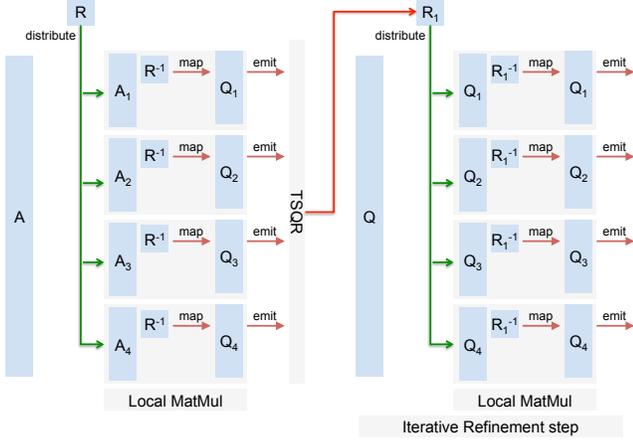


Figure 3. Indirect MapReduce computation of  $Q$  with iterative refinement.

As  $n$  grows, the performance of this algorithm becomes significantly worse than our other algorithms.

This MapReduce implementation of Householder QR is a BLAS 2 algorithm, whereas standard Sca/LAPACK uses a BLAS 3 algorithm [1], [3]. The central reason for this is the row-wise layout of the matrix in the Hadoop Distributed File System (HDFS). For tall-and-skinny matrices, the canonical key-value pair stored in HDFS uses a row as the matrix as the value and a unique row identifier for the key. Thus, reading the leading columns of the matrix has the same cost as reading the entire matrix. The stock BLAS 3 algorithm for LAPACK is a much better choice for their column-wise matrix layout.

### B. 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. 5. This avoids the iterative nature of the Householder methods. 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  $Q_j R_j$  the

computed factorization on the  $j$ th task:

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

The 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  $Q$ :

$$\underbrace{Q}_{8n \times n} = \underbrace{\begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} Q_1^2 \\ Q_2^2 \\ Q_3^2 \\ Q_4^2 \end{bmatrix}}_{4n \times n} \\ = \underbrace{\begin{bmatrix} Q_1 Q_1^2 \\ Q_2 Q_2^2 \\ Q_3 Q_3^2 \\ Q_4 Q_4^2 \end{bmatrix}}_{8n \times n} \\ A = Q \tilde{R}$$

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 of  $QU$  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  $R$ .

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

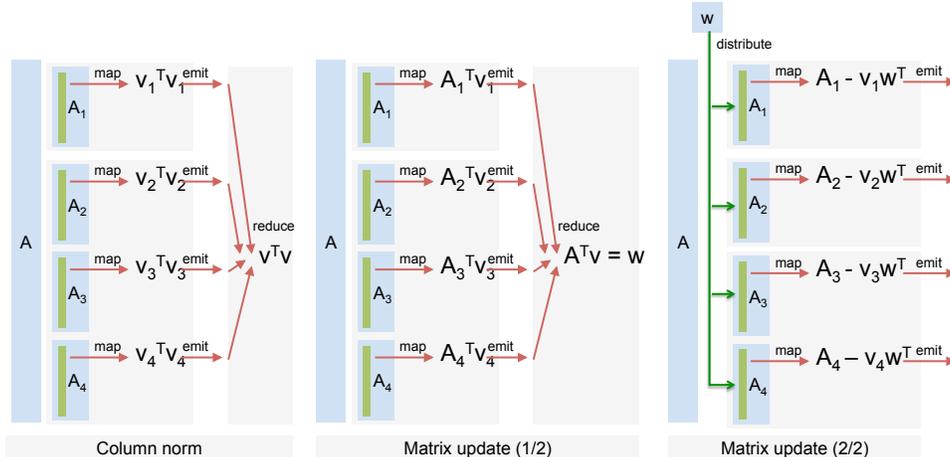


Figure 4. Outline of MapReduce Householder QR.

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.

### C. 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 processor in the second step. As the matrix becomes fatter, this serial bottleneck becomes limiting. We can cope with this issue by recursively extending the method with a recursive step following the first step. The algorithm is outlined in Alg. 2.

---

#### Algorithm 2 Recursive extension of direct method

---

```

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. STABILITY EXPERIMENTS

A major motivation for using the Direct TSQR method is numerical stability. Based on prior work, we know that the Direct TSQR method should produce a matrix  $Q$  with columns that are orthogonal to machine precision [8], [14], and Indirect TSQR and Cholesky QR should fail if the matrix is sufficiently ill-conditioned. Fig. 6 shows results from a numerical stability experiment which measures the loss in orthogonality in  $Q$  for Cholesky QR (with and without iterative refinement), Indirect TSQR (with and without iterative refinement), and Direct TSQR. We use  $\|Q^T Q - I\|_2$  to measure the accuracy of  $Q$ . As expected, using the inverse results in error that scales with the condition number. One step of iterative refinement and the direct TSQR method both yield errors consistently around  $10^{-15}$ . Cholesky QR fails when the condition number of the matrix is  $10^8$  or greater, and Indirect TSQR with iterative refinement has a large error when the condition number reaches  $10^{16}$ . Previous work by Langou shows consistent results for similar experiments [13].

## V. PERFORMANCE EXPERIMENTS

We evaluate performance in three 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. Finally, we explore the fault-tolerance of the MapReduce system by artificially introducing faults

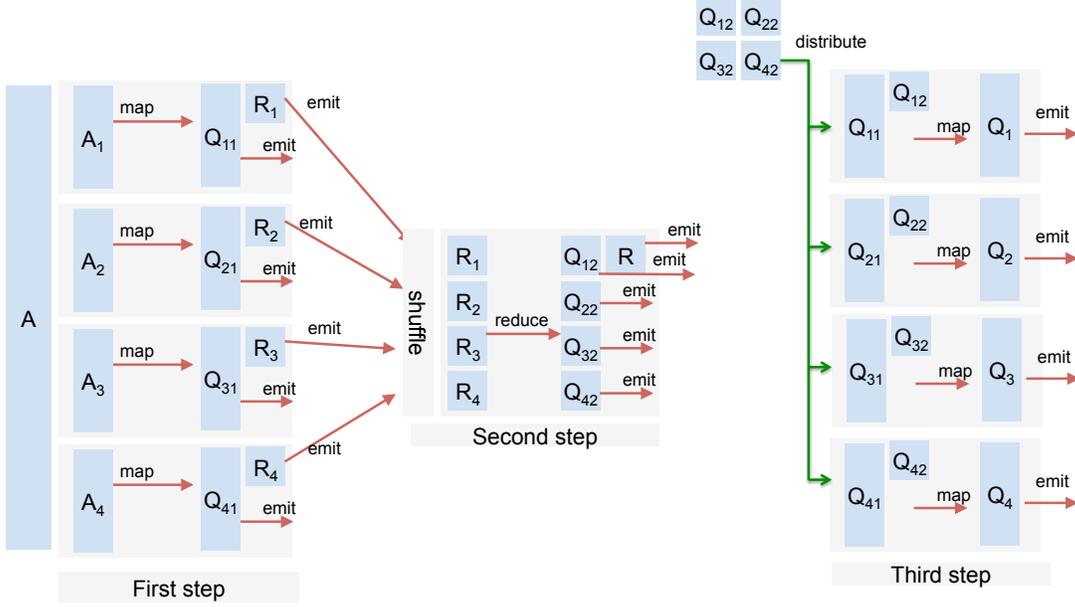


Figure 5. Direct MapReduce computation of  $Q$  and  $R$ .

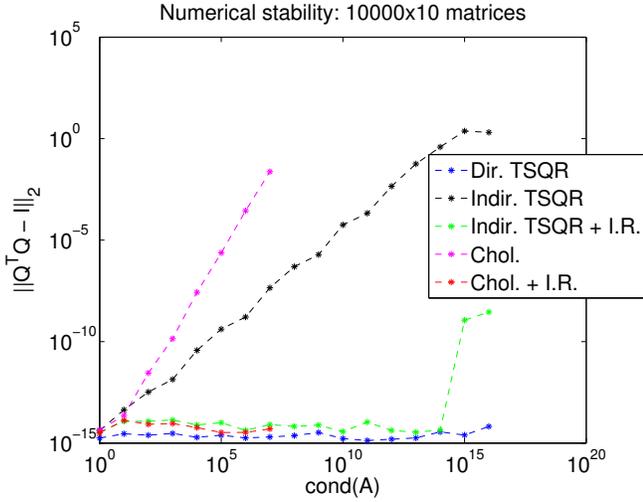


Figure 6. Stability measurements for each algorithm for matrices of varying condition number

into each task. Even when the frequency of faults is  $1/8$ , the runtime only grows by about 23.2%.

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

Let  $m_j$  and  $r_j$  be the number of map and reduce tasks for step  $j$ , respectively. Let  $m_{max}$  be the maximum number of map tasks and  $r_{max}$  be the maximum number

of reduce tasks for the cluster. Both  $m_{max}$  and  $r_{max}$  are fixed in the Hadoop configuration, and  $m_{max} + r_{max}$  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  $p_j^m = \min\{m_{max}, m_j\}$  and the reduce parallelism for step  $j$  as  $p_j^r = \min\{r_{max}, r_j, k_j\}$ . Let  $R_j^m$ ,  $W_j^m$  be the amount of data read and written in the  $j$ th map task, respectively. We have analogous definitions for  $R_j^r$  and  $W_j^r$  for the  $j$ th reduce task. Finally, let  $\beta_r$  and  $\beta_w$  be the inverse read and write bandwidth, respectively. After computing  $\beta_r$  and  $\beta_w$ , 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  $\beta_r$  and  $\beta_w$  for the 40-core ICME cluster, and the results are in Table II. On this cluster,  $m_{max} = r_{max} = 40$ . Table III provides the number of reads and writes for our algorithms, and Table IV provides 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. V-B, we examine how close the implementations are to the lower bounds.

#### B. Algorithmic comparison

Using one step of iterative refinement yields numerical errors that are acceptable in a vast majority of cases. In

Table II

STREAMING TIME TO READ FROM AND WRITE TO DISK. PERFORMANCE IS IN INVERSE BANDWIDTH, SO LARGER  $\beta_r$  AND  $\beta_w$  MEANS SLOWER STREAMING. THE STREAMING BENCHMARKS ARE PERFORMED WITH  $m_{max}$  MAP TASKS.

Rows	Cols.	HDFS Size (GB)	read+write (secs.)	read (secs.)	$\beta_r/m_{max}$ (s/GB)	$\beta_w/m_{max}$ (s/GB)
4,000,000,000	4	134.6	713	305	2.266	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	1.378	3.2407
150,000,000	100	109.6	504	152	1.3869	3.2117

Table III

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). ONLY ONE ITERATION OF HOUSEHOLDER QR IS SHOWN: THE LOWER BOUND REPEATS THIS ITERATION  $n$  TIMES. 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.

	Cholesky	Indirect TSQR	Direct TSQR	House. (1 step)
$R_1^m$	$8mn + Km$	$8mn + Km$	$8mn + Km$	$8mn + Km$
$W_1^m$	$8m_1n^2 + 8m_1n$	$8m_1n^2 + 8m_1n$	$8mn + 8m_1n^2 + Km + 64m_1$	$8mn + Km$
$R_1^r$	$8m_1n^2 + 8m_1n$	$8m_1n^2 + 8m_1n$	0	0
$W_1^r$	$8n^2 + 8n$	$8r_1n^2 + 8r_1n$	0	0
$R_2^m$	$8n^2 + 8n$	$8r_1n^2 + 8r_1n$	$8m_1n^2 + Km_1$	$8mn + Km$
$W_2^m$	$8n^2 + 8n$	$8r_1n^2 + 8r_1n$	$8m_1n^2 + Km_1$	$16m_1$
$R_2^r$	$8n^2 + 8n$	$8r_1n^2 + 8r_1n$	$8m_1n^2 + Km_1$	0
$W_2^r$	$8n^2 + 8n$	$8n^2 + 8n$	$8m_1n^2 + 32m_1 + 8n^2 + 8n$	0
$R_3^m$	$8mn + Km + m_3(8n^2 + 8n)$	$8mn + Km + m_3(8n^2 + 8n)$	$8mn + Km + m_3(8m_1n^2 + 64m_1)$	—
$W_3^m$	$8mn + Km$	$8mn + Km$	$8mn + Km$	—
$R_3^r$	0	0	0	—
$W_3^r$	0	0	0	—

Table V

COMPUTED LOWER BOUNDS FOR EACH ALGORITHM.

Rows	Cols.	Cholesky	Indirect TSQR	Cholesky +I.R.	Indirect TSQR+I.R.	Direct TSQR	House.
$T_{lb}$ (secs.)							
4,000,000,000	4	1803	1803	3606	3606	2528	7213
2,500,000,000	10	1645	1645	3290	3290	2464	16448
600,000,000	25	804	804	1609	1609	1236	20111
500,000,000	50	1240	1240	2480	2480	2095	61989
150,000,000	100	696	696	1392	1392	1335	69569

these cases, performance is our motivator for algorithm choice. Tabs. VI and VII show performance results of the Indirect and Direct TSQR methods, Cholesky QR, and Householder QR for a variety of matrices. The running time of Householder QR is long enough that we extrapolate the performance data from the first four steps of the algorithm.

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 outperforms the indirect methods with iterative refinement. The performance gain for this method is the greatest for smaller numbers of 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.

For every matrix, Householder QR is by far the slowest method. As the number of columns grows, the algorithm

Table VI

TIMES TO COMPUTE  $QR$  ON A VARIETY OF MATRICES WITH FOUR MAPREDUCE ALGORITHMS. \*HOUSEHOLDER QR DATA EXTRAPOLATED FROM THE FIRST FOUR STEPS OF THE ALGORITHM.

Rows	Cols.	HDFS Size (GB)	Cholesky	Indirect TSQR	Cholesky +I.R.	Indirect TSQR+I.R.	Direct TSQR	House.*
			job time (secs.)					
4,000,000,000	4	134.6	2931	4076	5832	7431	6128	15021
2,500,000,000	10	193.1	2508	2509	5011	5052	4035	32950
600,000,000	25	112.0	1098	1104	2221	2235	1910	37388
500,000,000	50	183.6	1563	1618	3204	3298	3090	117775
150,000,000	100	109.6	921	954	1878	1960	2154	133025

Table VII

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

Rows	Cols.	2*rows*cols <sup>2</sup>	Cholesky	Indirect TSQR	Cholesky +I.R.	Indirect TSQR+I.R.	Direct TSQR	House.*
			2*rows*cols <sup>2</sup> /sec					
4,000,000,000	4	1.28e+11	4.37e+07	3.14e+07	2.19e+07	1.72e+07	2.09e+07	8.52e+06
2,500,000,000	10	5.00e+11	1.99e+08	1.99e+08	9.98e+07	9.90e+07	1.24e+08	1.52e+07
600,000,000	25	7.50e+11	6.83e+08	6.79e+08	3.38e+08	3.36e+08	3.93e+08	2.01e+07
500,000,000	50	2.50e+12	1.60e+09	1.55e+09	7.80e+08	7.58e+08	8.09e+08	2.12e+07
150,000,000	100	3.00e+12	3.26e+09	3.14e+09	1.60e+09	1.53e+09	1.39e+09	2.26e+07

Table IV

VALUES NEEDED TO COMPUTE  $p_j^m$  AND  $p_j^r$ . FOR HOUSEHOLDER QR, ONLY THE DATA FOR ONE STEP IS SHOWN. EACH STEP OF HOUSEHOLDER QR HAS IDENTICAL DATA. BOTH  $m_1$  AND  $m_3$  ARE DEPENDENT ON THE MATRIX SIZE. OTHER LISTED DATA ARE NOT.

		Cholesky	Indirect TSQR	Direct TSQR	House. (1 step)
$4.0B \times 4$	$m_1$	1200	1200	2000	1200
$2.5B \times 10$		1680	1680	2640	1680
$600M \times 25$		1200	1200	1600	1920
$500M \times 50$		1920	1920	2560	1920
$150M \times 100$		1200	1200	1600	1200
	$m_2$	$m_{max}$	$m_{max}$	$m_{max}$	—
$4.0B \times 4$	$m_3$	1200	1200	2000	—
$2.5B \times 10$		1680	1680	2640	—
$600M \times 25$		1200	1200	1600	—
$500M \times 50$		1920	1920	2560	—
$150M \times 100$		1200	1200	1600	—
	$r_1$	$r_{max}$	$r_{max}$	$r_{max}$	—
	$r_2$	1	1	1	—
	$k_1$	$n$	$m_1 n$	$m_1$	—
	$k_2$	$n$	$m_1 n$	$m_1$	—
	$k_3$	0	0	0	—

becomes continuously less competitive.

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., the 4,000,000,000  $\times$  4 matrix, then this

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

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. The enormous lower bound of Householder QR makes the algorithm entirely unattractive, which renders Direct TSQR the best algorithm if guaranteed stability is required.

### C. Fault tolerance

One motivation for using a MapReduce architecture is fault tolerance. We measure the effects of faults on performance by crashing tasks with a certain probability of fault. Fig. 7 shows how the performance changes as we vary the probability of failure for tasks while running the Direct TSQR method on a matrix with 800 million rows and 10 columns. This matrix occupies 62.9 GB on HDFS.

In total, 800 map tasks are launched for each map stage of the Direct TSQR method. With no injected faults, the running time is 1220 seconds. When the

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

Rows	Cols.	Cholesky	Indirect TSQR	Cholesky +I.R.	Indirect TSQR+I.R.	Direct TSQR	House.
multiple of $T_{lb}$							
4,000,000,000	4	1.6256	2.2607	1.6173	2.0607	2.4241	2.0825
2,500,000,000	10	1.5246	1.5252	1.5231	1.5356	1.6376	2.0033
600,000,000	25	1.3657	1.3731	1.3804	1.3891	1.5453	1.8591
500,000,000	50	1.2605	1.3048	1.2919	1.3298	1.4749	1.8999
150,000,000	100	1.3233	1.3707	1.3491	1.4080	1.6135	1.9121

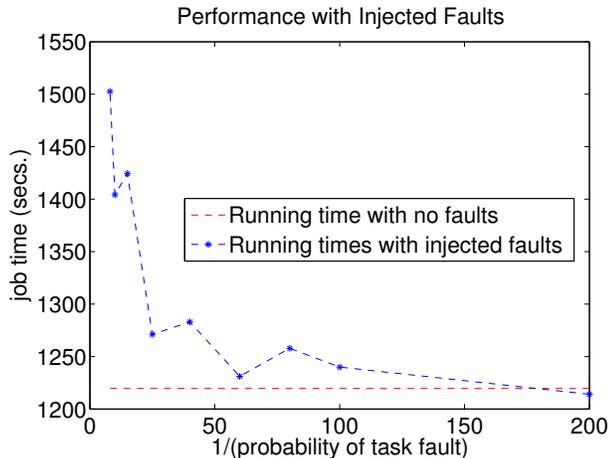


Figure 7. Running time of Direct TSQR on an  $800,000,000 \times 10$  matrix with injected faults

probability of a fault is  $1/8$ , the running time is 1503 seconds, only a 23.2 % performance penalty.

## VI. 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 often outperforms conceptually simpler methods. It is also orders of magnitude faster than the Householder QR method implemented in MapReduce.

All of the code used for this paper is openly available online, see:

<https://github.com/arbenson/mrtsqr>

This software runs on any system supporting Hadoop streaming, including cluster management systems like Mesos [12].

In the future we plan to investigate mixed MPI and Hadoop code. The idea is that once all the local mappers 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. We believe these changes would make our MapReduce codes significantly faster.

## ACKNOWLEDGMENT

Austin Benson is supported by an Office of Technology Licensing Stanford Graduate Fellowship. Many implementation optimizations were done by Austin Benson for the CS 267 (instructed by James Demmel and Kathy Yelick) and Math 221 (instructed by James Demmel) courses at UC-Berkeley. Thanks to the team at NERSC, including Lavanya Ramakrishnan and Shane Canon, for help with MapReduce codes on the Magellan cluster.

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

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Research is also supported by DOE grants DESC0003959 and de-sc0004938.

We are grateful to Stanford’s Institute for Computational and Mathematical Engineering 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] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [2] K. Bosteels. Dumbo. <http://projects.dumbotics.com/dumbo/>, 2012.
- [3] J. Choi, J. Demmel, I. S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *PARA*, pages 95–106, 1995.

- [4] 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.
- [5] P. Constantine and D. Gleich. Tall and skinny QR factorizations in mapreduce architectures. *Proceedings of the second international workshop on MapReduce and its applications*, page 43.50, 2011.
- [6] 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.
- [7] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *EECS-2008-89*, Aug. 2008.
- [8] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comp.*, 34, Feb. 2012.
- [9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10*, pages 810–818, New York, NY, USA, 2010. ACM.
- [10] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Benchmarking mapreduce implementations for application usage scenarios. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing, GRID '11*, pages 90–97, Washington, DC, USA, 2011. IEEE Computer Society.
- [11] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, October 1996.
- [12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI 2011*, 2011.
- [13] J. Langou. *Solving large linear systems with multiple right hand sides*. PhD thesis, INSA Toulouse, June 2003.
- [14] D. Mori, Y. Yamamoto, and S.-L. Zhang. Backward error analysis of the allreduce algorithm for householder qr decomposition. *Japan Journal of Industrial and Applied Mathematics*, 29(1):111–130, February 2012.
- [15] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA, USA, 1998.
- [16] S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.
- [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. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.
- [19] L. N. Trefethen and D. I. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.
- [20] Various. Hadoop version 0.21. <http://hadoop.apache.org>, 2012.
- [21] J. Zhao and J. Pjesivac-Grbovic. Mapreduce: The programming model and practice. <http://research.google.com/archive/papers/mapreduce-sigmetrics09-tutorial.pdf>, 2009. Tutorial.