Tall and Skinny QR factorizations in MapReduce architectures

Paul G. Constantine
Sandia National Laboratories[*]
Albuquerque, NM
pconsta@sandia.gov

David F. Gleich
Sandia National Laboratories[*]
Livermore, CA
dfgleic@sandia.gov

ABSTRACT

The QR factorization is one of the most important and useful matrix factorizations in scientific computing. A recent communication-avoiding version of the QR factorization trades flops for messages and is ideal for MapReduce, where computationally intensive processes operate locally on subsets of the data. We present an implementation of the tall and skinny QR (TSQR) factorization in the MapReduce framework, and we provide computational results for nearly terabyte-sized datasets. These tasks run in just a few minutes under a variety of parameter choices.

Categories and Subject Descriptors

G.1.3 [**Numerical analysis**]: Numerical Linear Algebra—*Sparse, structured, and very large systems (direct and iterative methods)*; H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Algorithms, Performance

Keywords

matrix factorization, QR factorization, TSQR, linear regression, Hadoop

1. INTRODUCTION

The thin QR factorization of an $m \times n$ matrix $\boldsymbol{A}$ with $m > n$ computes an $m \times n$ matrix $\boldsymbol{Q}$ with orthogonal columns and an $n \times n$ upper triangular matrix $\boldsymbol{R}$ such that

$$\boldsymbol{A} = \boldsymbol{QR}. \qquad (1)$$

Standard methods for computing the QR factorization are numerically stable [12], which has made it one of the most useful tools in scientific computing. In particular, it yields stable methods for least squares data fitting [2], eigenvalue computations [10], and null-space methods for optimization [11], to name a few. As the scale and scope of scientific computations increase, the technology of QR factorizations must be updated to meet the requirements of modern supercomputing environments, grid computations, and MapReduce environments.

However, the projection operations originally formulated to stably compute the factors $\boldsymbol{Q}$ and $\boldsymbol{R}$ assume a serial computing paradigm. Efforts to parallelize these operations have followed advances in parallel architectures [13]. For example, modern solvers work on blocks of a matrix to reduce the number of serial steps. But the serial approach has remained fundamentally unchanged. Thus, the resulting implementations have many synchronized steps; see, for example, the panel-QR routine in the state-of-the-art dense linear algebra library ScaLAPACK [4]. The standard approach is particularly ill-suited to the MapReduce environment because each step of the algorithm involves changing the entire matrix.

By examining trends in supercomputing hardware, Demmel et al. [8] showed that advances in processor performance – as measured by floating point operations per second – have greatly surpassed advances in communication performance. Consequently, they propose a *communication-avoiding* paradigm to revise dense linear algebra operations – such as the QR factorization – for modern supercomputer architectures. They present and analyze a communication-avoiding QR (CAQR) that essentially trades flops for messages, yielding a stable algorithm with optimal locality. The key component of CAQR is a modification of the thin QR factorization for matrices with many more rows than columns – the so-called Tall and Skinny QR (TSQR). The essence of TSQR is to perform smaller QR factorizations on row blocks of the tall matrix and to combine groups of $\boldsymbol{R}$ factors in such a way as to avoid communication. Then another round of smaller QR factorizations is performed; this process repeats until a final $\boldsymbol{R}$ factor is computed. The matrix $\boldsymbol{Q}$ of orthogonal columns is never explicitly formed, but premultiplying a vector by $\boldsymbol{Q}^T$ can be done efficiently; this is sufficient for many applications, e.g. least squares data-fitting. Beyond CAQR, the TSQR method also serves as an essential component of block iterative algorithms – such as block Krylov solvers for linear systems of equations – where the operation of a vector norm is replaced by a QR factorization on a matrix with many more rows than columns.

The communication-avoiding paradigm is appealing for a grid computing environment, where communication overhead is dramatically more expensive than computing power

[*] Sandia National Laboratories is a multi-program laboratorymanaged and operated by Sandia Corporation, a wholly ownedsubsidiary of Lockheed Martin Corporation, for the U.S.Department of Energy's National Nuclear Security Administrationunder contract DE-AC04-94AL85000.

MapReduce'11, June 8, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0700-0/11/06 ...$10.00.

on individual processors. By trading expensive synchronization points for additional local computation, communication-avoiding algorithms often fit into the MapReduce environment [7]. In MapReduce, these algorithms take the computation to the data in-situ, instead of communicating the data between computers. Here, we present the TSQR algorithm and its implementation in a MapReduce computing environment. By treating the matrix as an unordered collection of rows, we formulate both the mapping and reducing step as streaming TSQR factorizations. These approaches update a local QR factorization in response to a new block of rows. With this implementation, we can compute the QR factorization on terabyte scale data sets stored redundantly on the hard disks of a MapReduce cluster, and the bulk of the computation is done with data-local operations.

The related work falls into two broad categories: statistical algorithms in MapReduce, and tall-and-skinny QR factorizations in high-performance computing architectures. A recent proposal in the Apache Mahout package is to implement linear regression – a natural application of the tall-and-skinny QR factorization – using a stochastic gradient descent method [21]. In contrast, our approach allows these problems to be solved exactly when the matrix has a particular shape; see the conclusion for some ideas about future work related to Mahout. Another frequently used MapReduce computation is principal components analysis (PCA). The standard technique for implementing this algorithm is to compute the covariance matrix directly. See Section 2.4 for more about some issues with this approach.

The success and flexibility of the TSQR procedure developed by Demmel et al. [8] has sparked a small revolution in applications of this technique. It has been implemented *directly* inside the MPI framework for parallel codes as a native reduce operation [14]. This construction allows codes to use the TSQR procedure optimally on each architecture. Others have successfully used the TSQR procedure to compute the QR factorization in a grid environment [1] and on Amazon's EC2 platform [6].

2. THE TSQR FACTORIZATION

We now describe the idea behind a TSQR factorization and its implementation in a MapReduce system. As previously mentioned, this algorithm is a natural fit when only the $\boldsymbol{R}$ factor is required. The following derivation of the idea closely parallels ref. [8].

Consider a matrix $\boldsymbol{A}$ with $8n$ rows and n columns, which is partitioned as

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \boldsymbol{A}_3 \\ \boldsymbol{A}_4 \end{bmatrix}.$$

In this example, each $\boldsymbol{A}_i$ is a $2n \times n$ matrix, but in general, the idea is that each $\boldsymbol{A}_i$ is a small local matrix. We assume that $\boldsymbol{A}_i$ is small enough that we can efficiently compute its QR factorization with the standard algorithm on a single processor. These independent QR factorizations provide a factorization of $\boldsymbol{A}$:

$$\boldsymbol{A} = \underbrace{\begin{bmatrix} \boldsymbol{Q}_1 & & & \\ & \boldsymbol{Q}_2 & & \\ & & \boldsymbol{Q}_3 & \\ & & & \boldsymbol{Q}_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} \boldsymbol{R}_1 \\ \boldsymbol{R}_2 \\ \boldsymbol{R}_3 \\ \boldsymbol{R}_4 \end{bmatrix}}_{4n \times n}.$$

This is not yet a QR factorization for $\boldsymbol{A}$. Note, however, that a QR factorization of the $4n \times n$ matrix on the right will produce a QR factorization of $\boldsymbol{A}$. (Recall that the product of orthogonal matrices is orthogonal.) That is,

$$\boldsymbol{A} = \overbrace{\underbrace{\begin{bmatrix} \boldsymbol{Q}_1 & & & \\ & \boldsymbol{Q}_2 & & \\ & & \boldsymbol{Q}_3 & \\ & & & \boldsymbol{Q}_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\tilde{\boldsymbol{Q}}}_{4n \times n}}^{=\boldsymbol{Q}} \underbrace{\tilde{\boldsymbol{R}}}_{n \times n}.$$

But, suppose for the sake of the example that a $4n \times n$ QR factorization is itself too large for any single processor. Consequently, we repeat the small QR factorizations for both $2n \times n$ pairs of matrices:

$$\underbrace{\begin{bmatrix} \boldsymbol{R}_1 \\ \boldsymbol{R}_2 \\ \hline \boldsymbol{R}_3 \\ \boldsymbol{R}_4 \end{bmatrix}}_{4n \times n} = \underbrace{\begin{bmatrix} \boldsymbol{Q}_5 & \\ \hline & \boldsymbol{Q}_6 \end{bmatrix}}_{4n \times 2n} \underbrace{\begin{bmatrix} \boldsymbol{R}_5 \\ \hline \boldsymbol{R}_6 \end{bmatrix}}_{2n \times n}.$$

One final QR factorization of the $2n \times n$ matrix on the right remains. After that operation,

$$\boldsymbol{A} = \overbrace{\underbrace{\begin{bmatrix} \boldsymbol{Q}_1 & & & \\ & \boldsymbol{Q}_2 & & \\ & & \boldsymbol{Q}_3 & \\ & & & \boldsymbol{Q}_4 \end{bmatrix}}_{8n \times 4n} \underbrace{\begin{bmatrix} \boldsymbol{Q}_5 & \\ \hline & \boldsymbol{Q}_6 \end{bmatrix}}_{4n \times 2n} \underbrace{\boldsymbol{Q}_7}_{2n \times n}}^{=\boldsymbol{Q}} \underbrace{\boldsymbol{R}_7}_{n \times n}$$

is a QR factorization. What this examples shows is that to produce the matrix $\boldsymbol{R}$, the only operation required is the QR factorization of a $2n \times n$ matrix. As stated, $\boldsymbol{Q}$ is the product of a sequence of three orthogonal matrices. We return to the issue of computing $\boldsymbol{Q}$ in Sections 2.2 and 2.3

As noted in ref. [8], *any* sequence or tree of QR factorizations of this form will work. The above example shows how to compute a QR factorization with maximum parallelism. (The factorizations of $\boldsymbol{A}_i$ are independent, as are the factorizations of $\begin{bmatrix} \boldsymbol{R}_1^T & \boldsymbol{R}_2^T \end{bmatrix}^T$ and $\begin{bmatrix} \boldsymbol{R}_3^T & \boldsymbol{R}_4^T \end{bmatrix}^T$.) In particular, suppose that the matrix $\boldsymbol{A}$ is presented one row at a time. Further, suppose that we have the ability to store $3n$ rows of $\boldsymbol{A}$ and compute a QR factorization for these stored rows. Then the following sequence of four QR factorizations will produce the $\boldsymbol{R}$ factor:

$$\boldsymbol{A}_1 = \boldsymbol{Q}_1\boldsymbol{R}_1; \begin{bmatrix} \boldsymbol{R}_1 \\ \boldsymbol{A}_2 \end{bmatrix} = \boldsymbol{Q}_2\boldsymbol{R}_2; \begin{bmatrix} \boldsymbol{R}_2 \\ \boldsymbol{A}_3 \end{bmatrix} = \boldsymbol{Q}_3\boldsymbol{R}_3; \begin{bmatrix} \boldsymbol{R}_3 \\ \boldsymbol{A}_4 \end{bmatrix} = \boldsymbol{Q}_4\boldsymbol{R}_4.$$

As an algorithm, this procedure first reads and stores the $2n$ rows for $\boldsymbol{A}_1$. It then computes a QR factorization of this block, yielding an $n \times n$ matrix $\boldsymbol{R}_1$. Next, it reads in another $2n$ rows for $\boldsymbol{A}_2$. At this point, its buffer of rows is full, and consequently, it "compresses" that to a new matrix $\boldsymbol{R}_2$ via a second QR factorization. This process continues in blocks of $2n$ rows until there are no more rows of $\boldsymbol{A}$ left. In a single equation, it computes:

$$\boldsymbol{A} = \overbrace{\underbrace{\begin{bmatrix} \boldsymbol{Q}_1 & & & \\ & \boldsymbol{I}_{2n} & & \\ & & \boldsymbol{I}_{2n} & \\ & & & \boldsymbol{I}_{2n} \end{bmatrix}}_{8n \times 7n} \underbrace{\begin{bmatrix} \boldsymbol{Q}_2 & & \\ & \boldsymbol{I}_{2n} & \\ & & \boldsymbol{I}_{2n} \end{bmatrix}}_{7n \times 5n} \underbrace{\begin{bmatrix} \boldsymbol{Q}_3 & \\ & \boldsymbol{I}_{2n} \end{bmatrix}}_{5n \times 3n} \underbrace{\boldsymbol{Q}_4}_{3n \times n}}^{=\boldsymbol{Q}} \underbrace{\boldsymbol{R}}_{n \times n}.$$

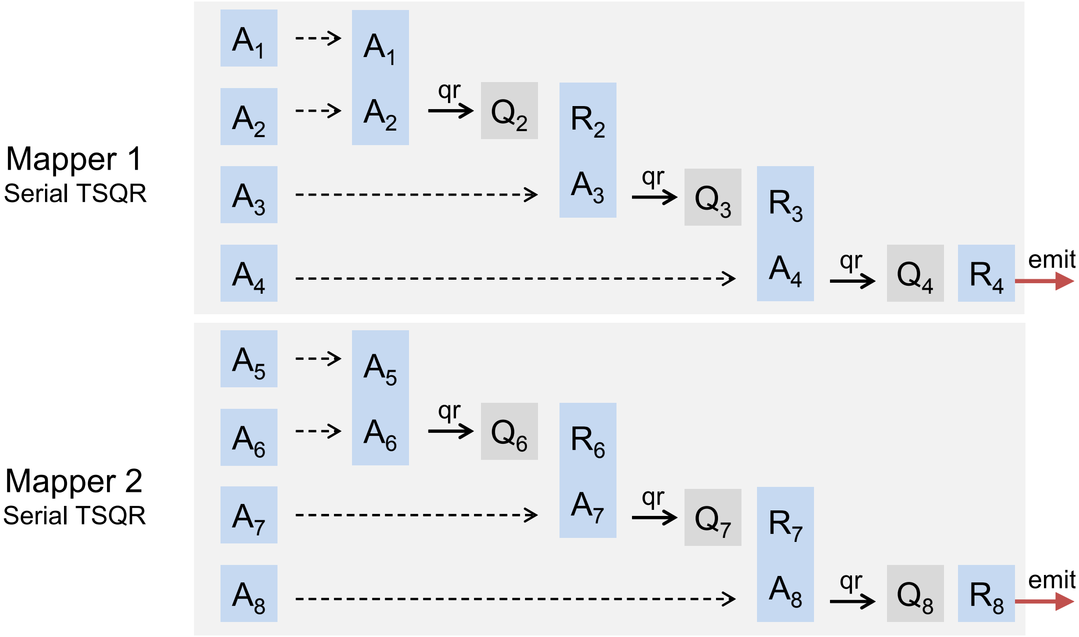


Figure 1: A pictorial description of the two mappers. The dashed arrows describe reading a block of rows from the input.

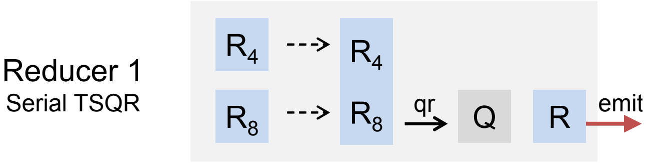


Figure 2: A pictorial description of the reducer and the steps involved in its serial TSQR procedure.

We now combine these two computation trees to implement a TSQR algorithm on a MapReduce computer.

2.1 A TSQR reduction tree for MapReduce

In the previous section, we saw how to compute the $\boldsymbol{R}$ factor in a QR factorization for a tall-and-skinny matrix in two scenarios: a maximally parallel computation and a completely serial computation. To implement a TSQR procedure on a MapReduce architecture, we combine these two cases.

In this implementation, the matrix exists as a set of rows. That is, each record, or key/value pair, in the MapReduce environment is a row identifier and an array containing the values in the row. This storage implies that a tall-and-skinny matrix will have many records. Due to the distributed nature of the file systems underlying a MapReduce system, this matrix will be partitioned across the many different computers and disks constituting the entire MapReduce cluster. In terms of the examples from the previous section, each submatrix $\boldsymbol{A}_i$ corresponds to a different split of the distributed data. Also, note that the matrix need not be stored explicitly. In Section 3.5, we describe how to solve a least squares problem on a single file representing a collection of images. There, we build the rows of the least-squares matrix directly from the raw image data.

Given the explicit or implicit storage of the matrix $\boldsymbol{A}$, each mapper in the MapReduce TSQR implementation runs a serial TSQR routine, acquiring the matrix a single row at a time. After it has acquired all the rows available, it outputs the $\boldsymbol{R}$ factor for these rows. Each row of the output has a random key. Thus, the input keys are ignored. Also, each reducer runs the same serial TSQR procedure. There is no need for a local "combine" operation because the identity of each row is ignored and the map already outputs the most highly reduced form of its data. A high level view of the operations in the map and reduce is shown in Figures 1 and 2. A complete implementation of this mapper and reducer using the hadoopy Python framework for Hadoop streaming programs is given in Figure 3.

When this MapReduce program is run with a single reducer, the output from that reducer is the $\boldsymbol{R}$ factor. Using only one reducer, however, eliminates some opportunities for parallelism. In the Hadoop file system, for instance, large files are divided into multiple segments called *splits*. The total number of mappers is the number of files multiplied by the number of splits. In most realistic scenarios, there will be more than a single file constituting the input matrix $\boldsymbol{A}$. This phenomenon often happens for one of two reasons. The first is that the data were collected in discrete increments, which are analyzed together – this would be the case for an aggregation of daily log files. The second is that the data are output by a previous MapReduce job, in which the number of files is equal to the number of reducers. Using many mappers and only a single reducer is problematic for this application because the reducer must compute a serial TSQR on the output from all the mappers, which could be a lengthy serial computation. An alternative is to use a multi-iteration TSQR approach; see Figure 4. When using two MapReduce iterations, the output from the mappers are fed into a large set of reducers. The output of these reducers

```python
import random, numpy, hadoopy
class SerialTSQR:
  def __init__(self,blocksize,isreducer):
    self.bsize=blocksize
    self.data = []
    if isreducer: self.__call__ = self.reducer
    else: self.__call__ = self.mapper

  def compress(self):
    R = numpy.linalg.qr(numpy.array(self.data),'r')
    # reset data and re-initialize to R
    self.data = []
    for row in R:
      self.data.append([float(v) for v in row])

  def collect(self,key,value):
    self.data.append(value)
    if len(self.data)>self.bsize*len(self.data[0]):
      self.compress()

  def close(self):
    self.compress()
    for row in self.data:
      key = random.randint(0,2000000000)
      yield key, row

  def mapper(self,key,value):
    self.collect(key,value)

  def reducer(self,key,values):
    for value in values: self.mapper(key,value)

if __name__=='__main__':
  mapper = SerialTSQR(blocksize=3,isreducer=False)
  reducer = SerialTSQR(blocksize=3,isreducer=True)
  hadoopy.run(mapper, reducer)
```

Figure 3: A complete hadoopy implementation of the mapper and reducer. See the text for more description.

then becomes the matrix input for another TSQR procedure – although this iteration can use an identity mapper to eliminate extra work involved in processing the output from the previous reduce, which is guaranteed not to change the output. As long as the last iteration has only a single reducer, the algorithm outputs the correct $\boldsymbol{R}$ factor.

The hadoopy implementation in Figure 3 does not contain the logic to handle a multi-stage iteration; please see our publicly available codes for that implementation. In both implementations, we did not invest any effort in constructing a custom data partitioner to ensure data locality between the mappers and reducers in a multi-stage iteration. Instead, we investigated increasing the minimum split size to reduce the number of mappers; see Section 3.4 for these experiments.

2.2 Solving a least-squares problem

Thus far, this manuscript has described how to compute a TSQR factorization and implement that algorithm on a MapReduce system. This algorithm is a key ingredient in solving least-squares problems. Consider a full-rank, least squares problem with a tall-and-skinny matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $m \gg n$:

$$\text{minimize } \|\mathbf{b} - \boldsymbol{A}\mathbf{x}\|.$$

The following derivation mimics that found in many numerical linear algebra textbooks and is repeated to illustrate how a QR factorization may be used. Let $\boldsymbol{A} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{R}}$ be a full-QR of $\boldsymbol{A}$. In the full-QR factorization, $\tilde{\boldsymbol{Q}}$ is square and orthogonal, and $\tilde{\boldsymbol{R}} = [\boldsymbol{R}^T \mathbf{0}^T]^T$, where $\boldsymbol{R}$ is $n \times n$ and upper triangular. Note that $\tilde{\boldsymbol{Q}} = \begin{bmatrix} \boldsymbol{Q} & \tilde{\boldsymbol{Q}}_2 \end{bmatrix}$ where $\boldsymbol{Q}$ is the thin-QR factor. Then, by orthogonality,

$$\|\mathbf{b} - \boldsymbol{A}\mathbf{x}\| = \|\tilde{\boldsymbol{Q}}^T\mathbf{b} - \tilde{\boldsymbol{Q}}^T\boldsymbol{A}\mathbf{x}\| = \left\| \begin{bmatrix} \boldsymbol{Q}^T\mathbf{b} \\ \tilde{\boldsymbol{Q}}_2^T\mathbf{b} \end{bmatrix} - \begin{bmatrix} \boldsymbol{R}\mathbf{x} \\ \mathbf{0}_{m-n \times n} \end{bmatrix} \right\|,$$

for any vector $\mathbf{x}$. Because $\boldsymbol{R}$ is full rank by assumption, the solution of the linear system $\boldsymbol{R}\mathbf{x} = \boldsymbol{Q}^T\mathbf{b}$ is the solution of the least squares problem. The value of the objective function at the minimum is $\|\tilde{\boldsymbol{Q}}_2^T\mathbf{b}\| = \sqrt{\|\mathbf{b}\|^2 - \|\boldsymbol{Q}^T\mathbf{b}\|^2}$.

Using this approach to compute the solution requires the vector $\boldsymbol{Q}^T\mathbf{b}$. Computing this vector seemingly requires the matrix $\boldsymbol{Q}$, but $\boldsymbol{Q}^T\mathbf{b}$ may be computed without explicitly forming $\boldsymbol{Q}$. We illustrate this idea by returning to the example in Section 2. Let $\boldsymbol{A}$ be the $4n \times n$ matrix from the example, and let $\mathbf{b} = \begin{bmatrix} \mathbf{b}_1^T & \mathbf{b}_2^T & \mathbf{b}_3^T & \mathbf{b}_4^T \end{bmatrix}^T$ be partitioned conformally with $\boldsymbol{A}$. (In a MapReduce environment, this means that we have the values of $\mathbf{b}$ stored with their corresponding rows of $\boldsymbol{A}$.) Using the QR factorization from before, note that

$$\boldsymbol{Q}^T\mathbf{b} = \boldsymbol{Q}_7^T \left[\begin{array}{cc} \boldsymbol{Q}_5^T & \\ \hline & \boldsymbol{Q}_6^T \end{array}\right] \begin{bmatrix} \boldsymbol{Q}_1^T & & & \\ & \boldsymbol{Q}_2^T & & \\ & & \boldsymbol{Q}_3^T & \\ & & & \boldsymbol{Q}_4^T \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \\ \mathbf{b}_4 \end{bmatrix}.$$

What this expression shows is that outputting $\boldsymbol{Q}_i^T\mathbf{b}_i$ from each small QR factorization is enough to compute $\boldsymbol{Q}^T\mathbf{b}$ without explicitly forming $\boldsymbol{Q}$.

2.3 The Q factor and a tall-and-skinny SVD

Of course, many applications require the $\boldsymbol{Q}$ factor. Given $\boldsymbol{R}$, we can compute $\boldsymbol{Q} = \boldsymbol{A}\boldsymbol{R}^{-1}$. This computation is easy to implement on a MapReduce system. Using a similar idea, we can also compute the SVD of a tall and skinny matrix (see ref. [12] for more about the SVD). However, these approaches have numerical stability problems (see ref. [17] for instance) and require a careful study of the orthogonality of the computed factors. We hope to investigate these issues soon.

2.4 Alternative approaches

Normal equations.

An elementary technique to solve a least-squares problem is to use the *normal equations* given by the optimality conditions for the least-squares problems:

$$\boldsymbol{A}^T\boldsymbol{A}\mathbf{x} = \boldsymbol{A}^T\mathbf{b}.$$

Let $\boldsymbol{B} = \boldsymbol{A}^T\boldsymbol{A}$. Then the Cholesky factorization of $\boldsymbol{B}$ is $\boldsymbol{B} = \boldsymbol{R}^T\boldsymbol{R}$ for an upper-triangular matrix $\boldsymbol{R}$. For a full-rank matrix $\boldsymbol{A}$, this Cholesky factor $\boldsymbol{R}$ *is the same* as the $\boldsymbol{R}$ factor in the QR factorization. Consequently, another approach to compute $\boldsymbol{R}$ is to compute $\boldsymbol{A}^T\boldsymbol{A}$ and compute a Cholesky factorization. This computation is also easy to do in a MapReduce environment. However, this approach has considerable numerical stability issues [19]. In our experiments, it has performance comparable to the TSQR approach, which has good numerical stability assurances.

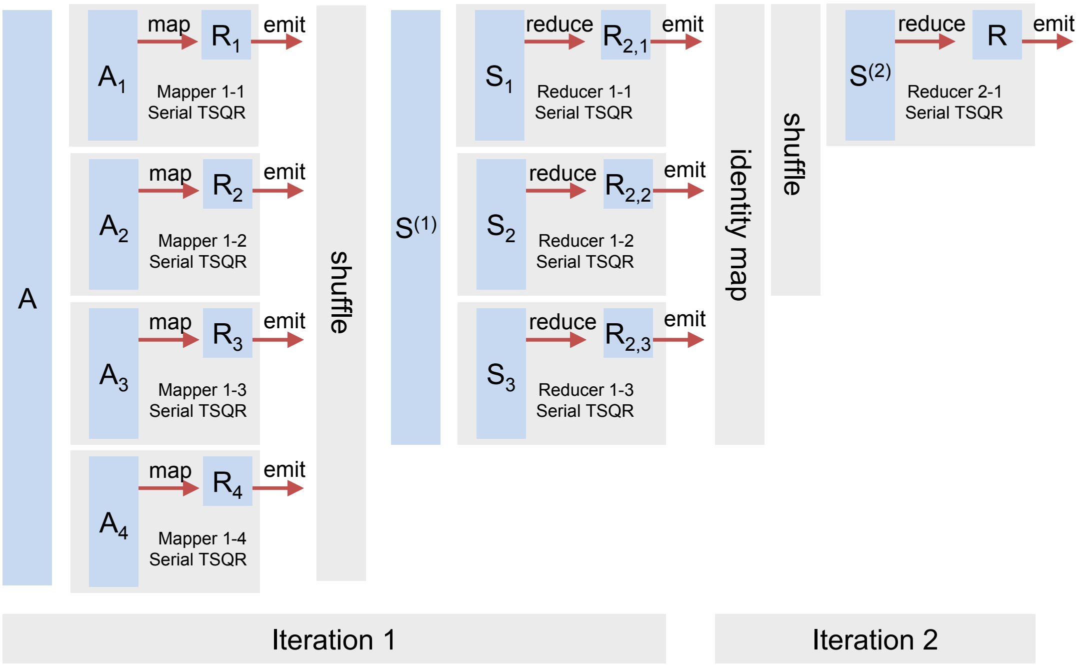


Figure 4: A pictorial description of a multiple iteration routine.

Random sampling.

Another approach is to employ random sampling of the matrix, or a random projection of the matrix (see ref. [9], and other references within). The idea is that most of the important quantities in a least-squares problem or QR factorization lie in a low-dimensional space, which can be obtained by randomly sampling rows of the matrix after an appropriate projection. These algorithms have rigorous approximation bounds and failure probabilities. However, using a random sampling technique still requires touching all the entries of the matrix. In our experiments, we observe that the additional work performed in the TSQR approach does not consume much additional time, and hence, we have not investigated randomized techniques, because the deterministic techniques have sufficient performance.

3. EXPERIMENTS

We implemented the TSQR algorithm outlined in the previous section in Hadoop 0.21 [20] streaming. Hadoop streaming is a simple interface to a MapReduce system, which pipes a stream of key-value pairs to and from an arbitrary program via the standard input and output streams. We constructed this implementation using three different interfaces to Hadoop streaming: the Dumbo python interface [3], the hadoopy python interface [23], and a new custom C++ interface. The two python frameworks provide an easy-to-use environment and a rich set of numerical libraries via the numpy Python package [15]. The C++ interface can be viewed as a reference point for any Hadoop streaming approach. In the future, we hope to evaluation against other TSQR approaches in MPI and grid environments.

We begin by describing our suite of synthetic test matrices (Table 1) that we use to evaluate the performance of the TSQR algorithms (Section 3.1). These problems are used to provide insight into how the performance of the algorithm changes when tweaking different parameters. The first set of experiments shows the difference in performance from each of the three different Hadoop streaming frameworks. Unsurprisingly, the C++ implementation is the fastest (Table 2). Please see Section 3.2 for additional discussion on these frameworks. The next two sets of experiments investigate the effect of the local block size and Hadoopy split size on the performance. The block size parameter determines the maximum size of a local QR factorization; and the split size determines the number of Hadoop mappers launched. The best performance results from a large, but not too large, block size (Table 3) and a large split size (Table 4). The final experiment uses the TSQR routine to solve a least squares regression problem and find the principal components of the tinyimages dataset [18], a 230 GB collection of nearly 80,000,000 small pictures. Using our slowest framework, Dumbo, we can solve both of these problems in around 30 minutes and project the C++ code would take 5-8 minutes.

All of the experiments are performed on a Hadoop cluster with 64 nodes. Each node has a single quad-core Intel Core i7-920 processor, 12 GB of RAM, and 4 2TB 7200 RPM hard disks. The nodes are split between two racks. The within rack connections are Gigabit Ethernet, and there is a single 10 Gigabit connection between racks.

3.1 Synthetic problems

In order to construct large scale synthetic problems, we implement the data generators directly in MapReduce using a Dumbo and numpy implementation. Let M be the number of mappers used, m be the number of rows in the final matrix and n be the number of columns. We deterministically construct $\boldsymbol{R}$ as an $n \times n$ upper-triangular matrix of all ones. This choice lets us quickly check that the computed answers are correct, which they were for all experiments here. In the first iteration of the construction, each mapper is assigned

Table 1: The synthetic test matrices we use in this manuscript. See §3.1.

Rows	Columns	HDFS Size	Col. sums (secs.)
1,000,000,000	50	453.3 GB	145
500,000,000	100	423.3 GB	161
100,000,000	500	381.2 GB	128
50,000,000	1000	375.8 GB	127

M/m rows to generate. Then each mapper generates a local, random orthogonal matrix $\boldsymbol{Q}_i$ with $2n$ rows. This matrix is applied to $\boldsymbol{R}$, and the resulting $2n$ rows are output. The mappers repeat this process until they have output all m rows. The output keys for these rows are random. The reduce step is just an identity reducer. However, with the random output keys, the reduce step is a random permutation of these rows, which is another orthogonal matrix transform. On subsequent iterations, the mappers read in up to $2r$ rows, and perform yet another orthogonal transformation with another random $\boldsymbol{Q}_i$. Overall, there are three total iterations: Mapper 1 to generate an initial matrix with a known $\boldsymbol{R}$ and local orthogonal transforms; Reducer 1 to perform a permutation of the rows; Mappers 2 and 3 to further transform the matrix with additional orthogonal factors; and Reducers 2 and 3 to perform additional permutations. The result is not a uniformly sampled random matrix with $\boldsymbol{R}$ as the QR factor, but an easy-to-implement approximation. See Table 1 for the details on the number of rows and columns in the test problems. They are all constructed to be around 400GB in size, and with many more rows than columns. The final number of reducers is 1000, and so each matrix consists of 1000 separate files, stored in HDFS. We also provide the time of a reference streaming computation on each matrix. This computation is the total time it takes to compute the column sums using our C++ streaming framework (described next), and approximates the minimum time to do one non-trivial MapReduce iteration. The reported times were consistent across three repetitions of the experiment. (We thank James Demmel for suggesting this performance benchmark.)

3.2 Prototyping frameworks

Each of the three streaming frameworks receive key and values pairs as binary encoded "TypedBytes" [3]. These TypedBytes allow binary data to be passed between Hadoop and Hadoop streaming mappers and reducers, and include integer, floating point, and array primitives. The values are encoded with a big endian byte order following the Java convention. These values must then be converted into the native byte order before they can be used, and this conversion operation occupies the majority of the time in decoding TypedBytes data. Once a key-value pair has been decoded, the three streaming implementations run the same algorithm as illustrated in Figure 3. On output, they re-encode the data for each row in big endian byte order.

In Table 2, we show the performance of the three implementations for the test matrix with 500 columns. This table shows the performance of three different Hadoop streaming frameworks using the TSQR algorithm. The QR columns give the total time spent doing local QR operations in each framework. The test matrix has 500 columns and uses a two-iteration approach with 250 reducers in the first iteration and 1 reducer in the second. We used a two-iteration TSQR procedure, and show the total seconds taken by the entire computation, each iteration, and the individual map and reduce steps. Note that the mappers and reducers often do bits of work in parallel, and thus, the sum of the map and reduce time for an iteration is more than the total iteration time. Also, we show the sum of time spent doing the QR factorizations. These experiments show that hadoopy is a more efficient Python framework for Hadoop streaming than Dumbo. Additionally, the hadoopy framework takes roughly twice as long as a C++ implementation. It seems the hadoopy linear algebra (using numpy and ATLAS [22]) routines incur a significantly higher overhead than the C++ code (using ATLAS), which could bias the comparison.

3.3 Local block-sizes

The serial TSQR procedure implemented by all the mappers and reducers has a block size parameter to determine how many rows to read before computing a local QR factorization. These block size parameters are expressed in terms of an integer multiple of the number of columns. For instance, a block size of 3 means that the mappers will read 3 rows for each column before computing a local QR factorization. We think of a block as an $n \times n$ chuck of the matrix, where n is the number of columns. As we vary the number of columns in the matrix, changing the block size can cause a meaningful performance difference; see Table 3. This computation uses a two iteration TSQR tree with our C++ code; it shows increasing the blocksize increases performance, but making it too large reduces performance. Each row in the table gives the number of columns in the matrix, the number of mappers used for the first iteration, and the time in seconds for each the two iterations. The two iteration algorithm used 250 reducers in the first iteration and 1 reducer in the second iteration.

The serial TSQR must repeat one block of work for each subsequent QR compression step. Consequently, a small block size causes a significant fraction of the total work to be redundant. The problem with a large block size is that it requires higher memory traffic. (Although, this could be improved with a better implementation of QR in ATLAS [8].)

Table 2: Results for the three Hadoop streaming frameworks. See §3.2.

	Iter 1					Iter 2				Overall
	Map		Red.		Total	Map	Red.		Total	
	Secs.	QR (s.)	Secs.	QR (s.)	Secs.	Secs.	Secs.	QR (s.)	Secs.	Secs.
Dumbo	911	67725	884	2160	960	5	214	80	217	1177
hadoopy	581	70909	565	2263	612	5	112	81	118	730
C++	326	15809	328	485	350	5	34	15	37	387

Table 3: Results when varying block size. The best performance results are bolded. See §3.3 for details.

		Iter. 1		Iter. 2
Cols.	Blks.	Maps	Secs.	Secs.
50	2	8000	424	21
—	3	—	399	19
—	5	—	408	19
—	10	—	401	19
—	20	—	396	20
—	50	—	406	18
—	**100**	—	**380**	**19**
—	200	—	395	19
100	2	7000	410	21
—	3	—	384	21
—	5	—	390	22
—	**10**	—	**372**	**22**
—	20	—	374	22
1000	2	6000	493	199
—	3	—	432	169
—	**5**	—	**422**	**154**
—	10	—	430	202
—	20	—	434	202

3.4 Split size

There are three factors that control the TSQR tree on Hadoop: the number of mappers, the number of reducers, and the number of iterations. In this section, we investigate the trade-off between decreasing the number of mappers, which is done by increasing the minimum split size in HDFS, and using additional iterations. Using additional iterations provides the opportunity to exploit parallelism via a reduction tree. Table 4 show the total computation time for our C++ code when used with various split sizes and one or two iterations. The block size used was the best performing case from the previous experiment. Each row states the number of columns, the number of iterations used (for two iterations, we used 250 reducers in the first iteration), the split size, and the total computation time. With a split size of 512 MB, each mapper consumes an entire input file of the matrix (recall that the matrices are constructed by 1000 reducers, and hence 1000 files). The two iteration test used 250 reducers in the first iteration and 1 reducer in the second iteration. The one iteration test used 1 reducer in the first iteration, which is required to get the correct final answer. (In the 1000 column test, using a smaller split size of 64 or 256 MB generated too much data from the mappers for a single reducer to handle efficiently.)

The results are different between 50 columns and 1000 columns. With 50 columns, a one iteration approach is faster, and increasing the split size dramatically reduces the computation time. This results from two intertwined behaviors: first, using a larger split size sends less data to the final reducer, making it run faster; and second, using a larger split size reduces the overhead with Hadoop launching additional map tasks. With 1000 columns, the two iteration approach is faster. This happens because each $\boldsymbol{R}$ matrix output by the mappers is 400 times larger than with the 50 column experiment. Consequently, the single reducer takes much longer in the one iteration case. Using an additional iteration allows us to handle this reduction with more parallelism.

Table 4: Results when varying split size. See §3.4.

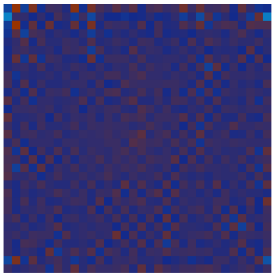
Cols.	Iters.	Split (MB)	Maps	Secs.
50	1	64	8000	388
—	—	256	2000	184
—	—	512	1000	149
—	2	64	8000	425
—	—	256	2000	220
—	—	512	1000	191
1000	1	512	1000	666
—	2	64	6000	590
—	—	256	2000	432
—	—	512	1000	337

3.5 Tinyimages: regression and PCA

Our final experiment shows this algorithm applied to a real world dataset. The tinyimages collection is a set of almost 80,000,000 images. Each image is 32-by-32 pixels. The image collection is stored in a single file, where each 3072 byte segment consists of the red, green, and blue values for each of the 1024 pixels in the image. We wrote a custom Hadoop InputFormat to read this file directly and transmit the data to our Hadoop streaming programs as a set of bytes. We used the Dumbo python framework for these experiments. In the following two experiments, we translated all the color pixels into shades of gray. Consequently, this dataset represents an 79,302,017-by-1024 matrix.

We first solved a regression problem by trying to predict the sum of red-pixel values in each image as a linear combination of the gray values in each image. Formally, if r_i is the sum of the red components in all pixels of image i, and $G_{i,j}$ is the gray value of the jth pixel in image i, then we wanted to find $\min \sum_i (r_i - \sum_j G_{i,j} s_j)^2$. There is no particular importance to this regression problem, we use it merely as a demonstration.

The coefficients s_j are displayed as an image at the right. They reveal regions of the image that are not as important in determining the overall red component of an image. The color scale varies from light-blue (strongly negative) to blue (0) and red (strongly positive). The computation took 30 minutes using the Dumbo framework and a two-iteration job with 250 intermediate reducers.



We also solved a principal component problem to find a principal component basis for each image. Let $\boldsymbol{G}$ be matrix of $G_{i,j}$'s from the regression and let u_i be the mean of the ith row in $\boldsymbol{G}$. The principal components of the images are given by the right singular vectors of the matrix $\boldsymbol{G} - \mathbf{u}\mathbf{e}^T$ where $\mathbf{u}$ are all of the mean values as a vector and $\mathbf{e}$ is the 1024-by-1 vector of ones. That is, let $\boldsymbol{G} - \mathbf{u}\mathbf{e}^T = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ be the SVD, then the principal components are the columns of $\boldsymbol{V}$. We compute $\boldsymbol{V}$ by first doing a TSQR of $\boldsymbol{G} - \mathbf{u}\mathbf{e}^T$, and then computing an SVD of the final $\boldsymbol{R}$, which is a small 1024-by-1024 matrix. The principal components are plotted as images in Figure 5. These images show a reasonable basis for images and are reminiscent of the basis in a discrete cosine transform.

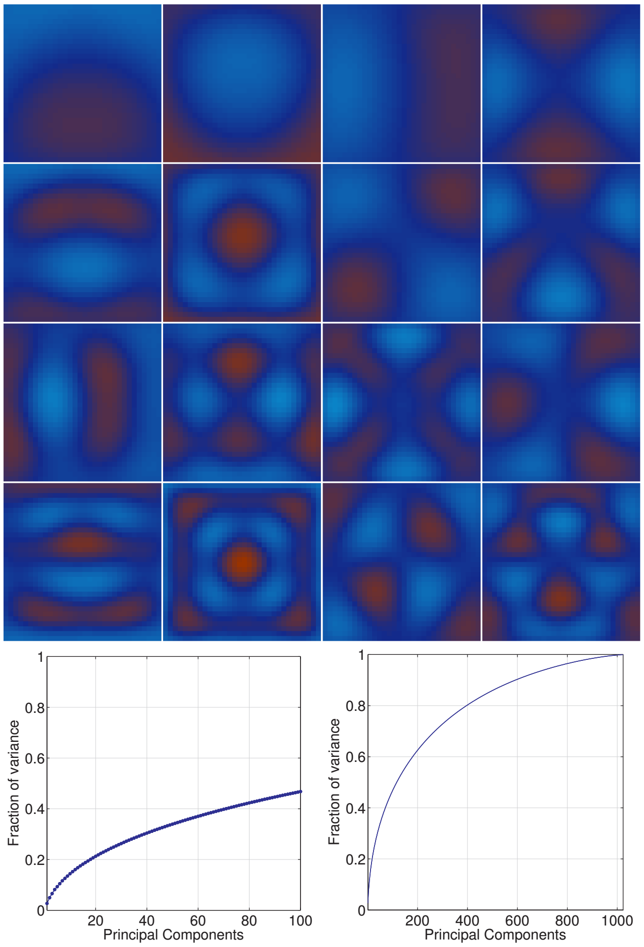


Figure 5: The 16 most important principal component basis functions (by rows) and the amount of variance explained by the top 100 (bottom left) and all principal components (bottom right).

4. CONCLUSION

In this manuscript, we have illustrated the ability of MapReduce architectures to solve massive least-squares problems through a tall and skinny QR factorization. We choose to implement these algorithms in a simple Hadoop streaming framework to provide prototype implementations so that others can easily adapt the algorithms to their particular problem. These codes are all available online.[1] We envision that the TSQR paradigm will find a place in block-analogues of the various iterative methods in the Mahout project. These methods are based on block analogues of the Lanczos process, which replace vector normalization steps with QR factorizations. Because the TSQR routine solves linear regression problems, it can also serve as the least-squares sub-routine for an iteratively reweighted least-squares algorithm for fitting general linear models.

A key motivation for our MapReduce TSQR implementation comes from a residual minimizing model reduction method [5] for approximating the output of a parameterized differential equation model. Methods for constructing reduced order models typically involve a collection of solutions (dubbed *snapshots* [16]) – each computed at its respective input parameters. Storing and managing the terascale data from these solutions is itself challenging, and the hard disk storage of MapReduce is a natural fit.

5. REFERENCES

[1] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langem. QR factorization of tall and skinny matrices in a grid computing environment. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, April 2010.

[2] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, Penn., 1996.

[3] K. Bosteels. Fuzzy techniques in the usage and construction of comparison measures for music objects, 2009.

[4] 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.

[5] P. G. Constantine and Q. Wang. Residual minimizing model reduction for parameterized nonlinear dynamical systems, arxiv:1012.0351, 2010.

[6] B. Dagnon and B. Hindman. TSQR on EC2 using the Nexus substrate. `http://www.cs.berkeley.edu/~agearh/cs267.sp10/files/writeup_dagnon.pdf`, 2010. Class Project writeup for CS267 and University of California Berkeley.

[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-avoiding parallel and sequential QR factorizations. *arXiv*, 0806.2159, 2008.

[9] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós. Faster least squares approximation. *Numerische Mathematik*, 117(2):219–249, 2011.

[10] J. G. F. Francis. The QR transformation a unitary analogue to the LR transformation – part 1. *The Computer Journal*, 4:265–271, 1961.

[11] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1981.

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

[13] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Rev.*, 20:740–777, 1978.

[14] J. Langou. Computing the R of the QR factorization of tall and skinny matrix using mpi_reduce. *arXiv*, math.NA:1002.4250, 2010.

[15] T. E. Oliphant. *Guide to NumPy*. Provo, UT, Mar. 2006.

[16] L. Sirovich. Turbulence and the dynamics of coherent structures. Part 1: Coherent structures. *Quar*, 45(3):561–571, 1987.

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

[18] A. Torralba, R. Fergus, and W. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958 –1970, November 2008.

[19] L. N. Trefethen and D. I. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[20] Various. Hadoop version 0.21. `http://hadoop.apache.org`, 2010.

[21] F. Wang. Implement linear regression. `https://issues.apache.org/jira/browse/MAHOUT-529`. Mahout-529 JIRA, accessed on February 10, 2011.

[22] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[23] B. White. hadoopy. `http://bwhite.github.com/hadoopy`.

Acknowledgments. We are exceedingly grateful to Mark Hoemmen for many discussions about the TSQR factorization. We would also like to thank James Demmel for suggesting examining the reference streaming time. Finally, we are happy to acknowledge the fellow MapReduce "computers" at Sandia for general Hadoop help: Craig Ulmer, Todd Plantenga, Justin Basilico, Art Munson, and Tamara G. Kolda.

[1] See `http://www.github.com/dgleich/mrtsqr`.