Scalable Matrix Inversion Using MapReduce

Jingen Xiang*
SAP
Waterloo, Ontario, Canada
xiangjingen@gmail.com

Huangdong Meng*
University of Toronto
Toronto, Ontario, Canada
mhd0371@gmail.com

Ashraf Aboulnaga*
Qatar Computing Research Institute
Doha, Qatar
aaboulnaga@qf.org.qa

ABSTRACT

Matrix operations are a fundamental building block of many computational tasks in fields as diverse as scientific computing, machine learning, and data mining. Matrix inversion is an important matrix operation, but it is difficult to implement in today's popular parallel dataflow programming systems, such as MapReduce. The reason is that each element in the inverse of a matrix depends on multiple elements in the input matrix, so the computation is not easily partitionable. In this paper, we present a scalable and efficient technique for matrix inversion in MapReduce. Our technique relies on computing the LU decomposition of the input matrix and using that decomposition to compute the required matrix inverse. We present a technique for computing the LU decomposition and the matrix inverse using a pipeline of MapReduce jobs. We also present optimizations of this technique in the context of Hadoop. To the best of our knowledge, our technique is the first matrix inversion technique using MapReduce. We show experimentally that our technique has good scalability, enabling us to invert a $10^5 \times 10^5$ matrix in 5 hours on Amazon EC2. We also show that our technique outperforms ScaLAPACK, a state-of-the-art linear algebra package that uses MPI.

Categories and Subject Descriptors

G.1.3 [**Numerical Linear Algebra**]: Matrix inversion; C.2.4 [**Distributed Systems**]: Distributed applications

Keywords

linear algebra; matrix inversion; analytics; MapReduce; Hadoop

1. INTRODUCTION

Parallel dataflow programming systems like MapReduce [8] and Pregel [21] have become very popular as platforms for scalable, analytical, data intensive computing. These systems offer the scalability and fault tolerance that are required to run in a cloud computing environment, plus simple programming models and easy-to-use programmer interfaces. Rich software ecosystems and large user communities have developed around these systems, and they are being used for many large-scale computational tasks in diverse scientific, business, and web applications. Due to the success and popularity of MapReduce and similar parallel dataflow systems, they are being used not only for the data intensive tasks for which they were originally designed, but also for computationally intensive tasks that have traditionally been assigned to high-performance computing systems. Therefore, there is significant benefit in developing optimized implementations of computationally intensive tasks using these systems. In this paper, we focus on MapReduce, arguably the most popular parallel dataflow programming system, and we present a technique for matrix inversion in MapReduce.

Matrix operations are a fundamental building block of many computational tasks in diverse fields including physics, bioinformatics, simulation, machine learning, data mining, and many others. In most of these fields, there is a need to scale to large matrices to obtain higher fidelity and better results (e.g., running a simulation on a finer grid, or training a machine learning model with more data). To scale to large matrices, it is important to design efficient parallel algorithms for matrix operations, and using MapReduce is one way to achieve this goal. There has been prior work on implementing matrix operations in MapReduce (e.g., [13]), but that work does not handle matrix inversion even though matrix inversion is a very fundamental matrix operation. Matrix inversion is difficult to implement in MapReduce because each element in the inverse of a matrix depends on multiple elements in the input matrix, so the computation is not easily partitionable as required by the MapReduce programming model. In this paper, we address this problem and design a novel partitionable matrix inversion technique that is suitable for MapReduce. We implement our technique in Hadoop, and develop several optimizations as part of this implementation.

Before we present our technique, we further motivate the importance of matrix inversion by presenting some of its applications in different fields. A key application of matrix inversion is solving systems of linear equations. To solve the equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are both vectors with n elements, one can multiply both sides of the equation on the left by the the matrix inverse $\mathbf{A}^{-1}$, to get $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Matrix inversion is also related to finding eigenvalues and eigenvectors, which is a central problem in many physics and engineering applications. The eigenvalues and eigenvectors of an $n \times n$ matrix can be computed using the inverse iteration method. This method assumes that there is an approximate eigenvalue μ and an approximate eigenvector $\mathbf{v}_0$ for matrix $\mathbf{A}$. The method uses an iteration step to compute an increasingly accurate eigenvector, $\mathbf{v}_{k+1} = \frac{(\mathbf{A}-\mu\mathbf{I}_n)^{-1}\mathbf{v}_k}{||(\mathbf{A}-\mu\mathbf{I}_n)^{-1}\mathbf{v}_k||}$, where I_n is an $n \times n$ identity matrix. At any step in the iteration, the current eigenvalue can be computed as

*Work done at the University of Waterloo.

HPDC 2014 Vancouver, Canada

http://dx.doi.org/10.1145/2600212.2600220.

$\lambda = \frac{\mathbf{v}^T\mathbf{A}\mathbf{v}}{\mathbf{v}^T\mathbf{v}}$, for real-valued $\mathbf{A}$ and $\mathbf{v}$. The efficiency of this iterative method relies on the ability to efficiently invert matrix $\mathbf{A} - \mu\mathbf{I}_n$.

Matrix inversion is also widely used in computed tomography (CT). In CT, the relationship between the original image of the material ($\mathbf{S}$) and the image ($\mathbf{T}$) detected by the detector can be written as: $\mathbf{T} = \mathbf{MS}$, where $\mathbf{M}$ is the projection matrix [28]. In order to reconstruct the original image, we can simply invert the projection matrix and calculate the product of $\mathbf{M}^{-1}$ and $\mathbf{T}$. As the accuracy of the detector increases, the number of image pixels increases and hence the order of the projection matrix ($\mathbf{M}$) also increases, motivating the need for scalable matrix inversion techniques. Image reconstruction using matrix inversion can also be found in other fields such as astrophysics [30].

In bioinformatics, matrix inversion is used to solve the problem of protein structure prediction [22]. A scalable matrix inversion technique would enable novel insights into the evolutionary dynamics of sequence variation and protein folding.

These are but a few of the numerous applications that rely on matrix inversion. In some cases, it may be possible to avoid matrix inversion by using alternate numerical methods (e.g., pseudo-Newton methods for convex optimization), but it is clear that a scalable and efficient matrix inversion technique such as the one we present would be highly useful in many applications.

The fact that our technique uses MapReduce and Hadoop has several advantages. First, our technique benefits from the scalability and fault tolerance of MapReduce. Second, our technique is part of the rich and popular software ecosystem of Hadoop that includes many systems and libraries such as HDFS, Pig, Mahout, Zookeeper, Cassandra, etc. Thus, our matrix inversion technique can easily be integrated with other parts of this ecosystem as needed. Finally, it is important to note that matrix inversion is often not a standalone activity, but rather one component of a rich data analysis workflow. These days, such a workflow is quite likely implemented in Hadoop, so the input matrix to be inverted would be generated by a MapReduce job and stored in HDFS, and the inverse matrix produced as output also needs to be stored in HDFS to be consumed by another MapReduce job. Therefore, while it may be possible to implement matrix inversion using other parallelization platforms such as MPI, a MapReduce matrix inversion technique that can be used as a pluggable component in complex Hadoop data analysis workflows is highly desirable. It may be possible to switch between MapReduce and MPI to perform scalable matrix inversion in these workflows, but staying within MapReduce is certainly more convenient.

Our goal is not to demonstrate that MapReduce is fundamentally superior to MPI or other parallelization platforms. As a matter of fact, some of the optimizations that we propose can be implemented in MPI. Instead, our goal is to provide a novel and powerful matrix inversion implementation that can be used in MapReduce systems, and to demonstrate the judicious mathematical manipulation and careful systems engineering behind this implementation. We experimentally demonstrate by comparing to ScaLAPACK [4] that we do not lose performance by using MapReduce, especially for large matrices (Section 7.5).

Our matrix inversion technique relies on computing the LU decomposition of the input matrix and using that decomposition to compute the required matrix inverse. We chose to use LU decomposition for matrix inversion since it enables us to partition the computation in a way that is suitable for MapReduce. In particular, we developed a technique for computing the LU decomposition of the input matrix using subtasks that do not communicate among each other except through intermediate results, which exactly matches the way mappers and reducers communicate in the MapReduce computational model. In addition, our LU decomposition technique is carefully designed to partition the computation into subtasks that require the same amount of computation, which ensures that the load among the mappers and reducers is balanced and no map or reduce task can hold up the computation. Our LU decomposition technique is implemented as a pipeline of MapReduce jobs, where the number of jobs in the pipeline and the data movement between the jobs can be precisely determined before the start of the computation.

We also developed a MapReduce technique for computing the inverse of the input matrix from its LU decomposition, and optimizations to improve numerical accuracy and performance in the context of Hadoop.

Note that the crux of our technique is partitioning the computation required for matrix inversion into independent subtasks. Such partitioning is required for any cluster computing framework. Thus, while this paper focuses on MapReduce, our technique can be used as a basis for implementing matrix inversion in other cluster computing systems such as Spark [34] or DryadLINQ [33]. We leave this point as a direction for future work.

The contributions of this paper are as follows:

- The choice of LU decomposition for matrix inversion in order to enable a MapReduce implementation.
- An algorithm for computing the LU decomposition that partitions the computation into independent subtasks.
- An implementation of the proposed algorithms as a pipeline of MapReduce jobs. The source code of this implementation is available on GitHub [31].
- Optimizations of this implementation to improve numerical accuracy, I/O performance, and memory locality.
- An extensive experimental evaluation on Amazon EC2.

The rest of the paper is organized as follows. In Section 2, we present some basic matrix inversion algorithms on a single node. In Section 3, we present related work. Our algorithm is introduced in Section 4, and our implementation in Section 5. We describe optimizations of the implementation in Section 6. We present an experimental evaluation in Section 7. Section 8 concludes.

2. MATRIX INVERSION PRELIMINARIES

For any matrix $\mathbf{A}$, let $[\mathbf{A}]_{ij}$ denote its element of the i-th row and the j-th column, and denote by $[\mathbf{A}]_{[x_1 \ldots x_2][y_1 \ldots y_2]}$ the block defined by the beginning row x_1 (inclusive) and the ending row x_2 (exclusive), and by the beginning column y_1 (inclusive) and the ending column y_2 (exclusive).

A square matrix $\mathbf{A}$ is a matrix with the same number of rows and columns. The number of rows and columns n is called the *order* of the matrix. The inverse of $\mathbf{A}$ is another square matrix, denoted by $\mathbf{A}^{-1}$, such that $\mathbf{AA^{-1}} = \mathbf{A^{-1}A} = \mathbf{I}_n$, where $\mathbf{I}_n$ is the identity matrix of order n. A square matrix $\mathbf{A}$ is invertible if and only if $\mathbf{A}$ is non-singular, that is, $\mathbf{A}$ is of full rank n. The rank of a matrix is the number of rows (or columns) in the largest collection of linearly independent rows (or columns) of a matrix. Therefore a square matrix is invertible if and only if there are no linearly dependent rows (or columns) in this matrix.

The inverse of a matrix can be computed using many methods [23], such as Gauss-Jordan elimination, LU decomposition (also called LU factorization) [19], Singular Value Decomposition (SVD) [27], and QR decomposition [7]. In order to clarify our

choice of matrix inversion method for MapReduce, we briefly discuss the different methods. We will show that most of these methods are not easily parallelizable in a MapReduce setting, and justify our choice of LU decomposition. A more detailed discussion can be found in [29].

Gauss-Jordan elimination is a classical and well-known method to calculate the inverse of a matrix [23]. This method has two different variants: row elimination and column elimination. They are quite similar so we only discuss the method using row elimination. The method first concatenates the matrix $\mathbf{A}$ and the identity matrix $\mathbf{I}_n$ into a new matrix $[\mathbf{A}|\mathbf{I}_n]$. Then, using elementary row operations which include row switching, row multiplication, and row addition, the method transforms the left side to the identity matrix, which leaves the right side as the inverse of matrix $\mathbf{A}$. Specifically,

$$[\mathbf{A}|\mathbf{I}_n] \xrightarrow{\text{row operations}} [\mathbf{U}|\mathbf{B}] \xrightarrow{\text{row operations}} [\mathbf{I_n}|\mathbf{A^{-1}}] \qquad (1)$$

where $\mathbf{U}$ is an upper triangular matrix (nonzero elements only on the diagonal and above). The Gauss-Jordan elimination method first converts the matrix $\mathbf{A}$ into an upper triangular matrix in n steps using linear operations on the rows of the matrix as follows. In the first step, the first row is multiplied by a constant such that the first element in this row equals to 1, and the first row times a constant is subtracted from the i-th ($1 < i \leq n$) row of $[\mathbf{A}|\mathbf{I}_n]$ such that the first element in the i-th ($1 < i \leq n$) row is 0. In the k-th step, the k-th row is multiplied by a constant such that the k-th element in this row equals to 1, and the k-th row times a constant is subtracted from the i-th ($k < i \leq n$) row of $[\mathbf{A}|\mathbf{I}_n]$ such that the k-th element in the i-th ($k < i \leq n$) row is 0. If the k-th element of the k-th row is already 0 or close to 0 before the subtraction, we first swap the k-th row with any row below k where the k-th element is not 0. This is called *pivoting*, and it improves numerical accuracy. After $n-1$ steps, the left part of matrix $[\mathbf{A}|\mathbf{I}_n]$ is converted into an upper triangular matrix.

Next, the method converts the upper triangular matrix into an identity matrix using row operations similar to the ones described in the previous paragraph. This requires n steps, and it also converts the right part into the inverse of matrix $\mathbf{A}$. The Gauss-Jordan method uses n^3 multiplication operations and n^3 addition operations to invert an $n \times n$ matrix, which is as efficient as, if not better than, any other method. However, due to the large number of steps that depend on each other in a sequential fashion, this method is difficult to parallelize in MapReduce since it would require a large number of MapReduce jobs that are executed sequentially. For example, Quintana et al. [25] propose a parallel (not MapReduce) matrix inversion algorithm based on Gauss-Jordan elimination, but this method requires n iterations so a MapReduce implementation would require a pipeline of n MapReduce jobs. A main contribution of our paper is reducing this number to around n/n_b jobs (details in Section 4).

The LU decomposition method, also called LU factorization, first decomposes the original matrix into a product of two matrices $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is a lower triangular matrix that has nonzero elements only on the diagonal and below, and $\mathbf{U}$ is an upper triangular matrix that has nonzero elements only on the diagonal and above. Since the inverse of a triangular matrix is easy to compute using back substitution (described in Section 4), we compute the inverse of $\mathbf{A}$ as $\mathbf{U}^{-1}\mathbf{L}^{-1}$. The LU decomposition method uses the same number of multiplication and addition operations as the Gauss-Jordan method. However the LU decomposition method is much easier to parallelize because the LU decomposition can be implemented using a recursive block method. Therefore, in this paper, we use the LU decomposition method (details in Section 4).

Instead of decomposing matrix $\mathbf{A}$ into the product of two matrices, the SVD decomposition method [23] decomposes matrix $\mathbf{A}$ into the product of three matrices, $\mathbf{A} = \mathbf{UWV}^T$ ($\mathbf{V}^T$ is the transpose of matrix $\mathbf{V}$). $\mathbf{W}$ is a diagonal matrix with only positive or zero elements. $\mathbf{U}$ and $\mathbf{V}$ are both orthogonal matrices (i.e., $\mathbf{UU}^T = \mathbf{VV}^T = \mathbf{I}_n$), such that the inverse of $\mathbf{A}$ can be given by $\mathbf{A}^{-1} = \mathbf{VW}^{-1}\mathbf{U}^T$, where the inverse of diagonal matrix $\mathbf{W}$ is easily obtained by $[\mathbf{W^{-1}}]_{ii} = 1/[\mathbf{W}]_{ii}$ in running time $O(n)$. The SVD method needs frequent row exchanges, which means that the computation cannot be partitioned into independent subtasks, so it is not suitable for MapReduce.

The QR decomposition first decomposes the original matrix $\mathbf{A}$ into a product of an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$, i.e., $\mathbf{A} = \mathbf{QR}$ and $\mathbf{A}^{-1} = \mathbf{R}^{-1}\mathbf{Q}^T$. One way to compute the QR decomposition is using the Gram-Schmidt process [23]. This effectively decomposes the matrix but requires computing a sequence of n vectors where each vector relies on all previous vectors (i.e., n steps are required). Thus, the QR decomposition method is not easy to parallelize in MapReduce.

To recap: There are multiple ways to compute the inverse of a matrix. Many of these ways use an optimal number of multiplication and addition operations, but are not easy to parallelize. In this paper we use LU decomposition since it uses the optimal number of operations and is easy to parallelize. LU decomposition can also be used with pivoting to improve numerical accuracy.

3. RELATED WORK

Several software packages have been developed that support matrix inversion, such as LINPACK [10], LAPACK [2], and ScaLAPACK [4]. The LINPACK package is written in Fortran and designed for supercomputers. The LAPACK package is developed from LINPACK and is designed to run efficiently on shared-memory vector supercomputers. ScaLAPACK is a software package that tries to provide a high-performance linear algebra library for parallel distributed memory machines instead of shared-memory in LAPACK. This package provides some routines for matrix inversion (see Section 7.5 for details). However, this package does not provide any fault tolerance, while our algorithm provides fault tolerance through the use of MapReduce. In addition, we show that the scalability of ScaLAPACK is not as good as our algorithm.

Parallel algorithms for inverting some special matrices also appear in the literature. Lau, Kumar, and Venkatesh [17] propose algorithms for parallel inversion of sparse symmetric positive matrices on SIMD and MIMD parallel computing platforms. It is not a surprise that these algorithms perform better than general algorithms that do not take into account any special properties of the input matrix. For symmetric positive definite matrices (not necessarily sparse), Bientinesi, Gunter, and Geijn [3] present a parallel matrix inversion algorithm based on the Cholesky factorization for symmetric matrices. Their implementation is based on the Formal Linear Algebra Methodology Environment (FLAME). The implementation shows good performance and scalability, but it does not work for general matrices and is not suitable for large clusters.

The most important step in our matrix inversion technique is LU decomposition. This decomposition has been investigated by many researchers. Agullo et al. [1] have shown that the LU decomposition in double precision arithmetic can reach a throughput of 500 Gflops. That work uses powerful CPUs, GPUs, and large memory. Although this method can solve the LU decomposition problem very efficiently, it is a centralized method that is not suitable for MapReduce. Moreover, it needs special hardware (GPUs) and large memory.

Zheng et al. [36] present an implementation of LU decomposition on a multi-core digital signal processor that does pre-fetching and pre-shuffling in MapReduce [26]. The algorithm is very simple and only runs row operations in reduce tasks, using the LU decomposition algorithm on a single node. That is, one reduce task computes one row as in lines 10–12 in Algorithm 1, so that the method needs n MapReduce tasks to decompose an $n \times n$ matrix, which represents very poor scalability.

Matrix inversion using LU decomposition has been investigated recently by Dongarra et al. [9], where a tile data layout [1] is used to compute the LU decomposition and the upper triangular matrix inversion. In that paper, a run time environment called QUARK is used to dynamically schedule numerical kernels on the available processing units in order to reduce the synchronization required between different CPU cores. The algorithm is suitable for multi-core architectures with shared memory and it achieves better performance than other numerical libraries, such as LAPACK and ScaLAPACK. However, this algorithm is not suitable for a distributed environment, since it relies on large shared memory. Hence, its scalability is limited.

Zhang and Yang [35] investigate I/O optimization for big array analytics. They improve the performance of a broad range of big array operations by increasing sharing opportunities. Our technique also optimizes I/O for big matrices (Section 6.2). However, our technique mainly focuses on reusing the data in memory as many times as possible to reduce the need for reading data from disk.

To the best of our knowledge, there are no matrix inversion algorithms using MapReduce, although there are several software systems for other matrix operations using MapReduce. One of these systems is SystemML [13], which provides a high-level language for expressing some matrix operations such as matrix multiplication, division, and transpose, but not matrix inversion. SystemML provides MapReduce implementations of these operations, and it achieves good scalability and fault tolerance. A MapReduce implementation of matrix factorization using distributed stochastic gradient descent is described in [12], and is extended to Spark in [18]. These works do not help in solving the matrix inversion problem.

Two other systems that support matrix operations are the MADlib library [15] and MadLINQ [24]. MADlib does not use MapReduce but instead focuses on parallel analytics inside relational database systems. MADlib includes a conjugate gradient method to solve linear equations, but it does not support parallel matrix inversion. MadLINQ provides a programming model for matrix operations and an execution framework based on DryadLINQ [33], but it does not directly address matrix inversion.

4. MATRIX INVERSION USING LU DECOMPOSITION

As discussed in Section 2, LU decomposition is the fundamental step in our solution to the matrix inversion problem. In this section, we show how we compute the lower triangular matrix $\mathbf{L}$ and the upper triangular matrix $\mathbf{U}$ in both single node and parallel settings.

4.1 LU Decomposition on a Single Node

The LU decomposition algorithm on a single node is widely studied. It has two variants: with and without pivoting. Since pivoting can significantly improve numerical accuracy, we only discuss the algorithm with pivoting (in Algorithm 1, the row having the maximum j-th element among rows j to n is selected in the j-th loop). Although this algorithm can be found in many references (e.g., [14] and [23]), we present it here for completeness.

Algorithm 1 LU decomposition on a single node.

```
1: function LUDecomposition(A)
2: for i = 1 to n do
3:     j = {j | [A]_ji = max([A]_ii, [A]_{i+1i}, ..., [A]_ni)}
4:     Add j to P
5:     Swap i-th row with j-th row if i ≠ j
6:     for j = i + 1 to n do
7:         [A]_ji = [A]_ji / [A]_ii
8:     end for
9:     for j = i + 1 to n do
10:        for k = i + 1 to n do
11:            [A]_jk = [A]_jk − [A]_ji × [A]_ik
12:        end for
13:    end for
14: end for
15: return (A, P)    /* i.e., return (L, U, P) */
```

Let $a_{ij} = [\mathbf{A}]_{ij}$, $l_{ij} = [\mathbf{L}]_{ij}$, and $u_{ij} = [\mathbf{U}]_{ij}$. The LU decomposition $\mathbf{A} = \mathbf{LU}$ can be presented as follows (the blank elements in the $\mathbf{L}$ and $\mathbf{U}$ matrices are zeros):

$$\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ a_{31} & a_{32} & ... & a_{3n} \\ ... & & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} = \tag{2}$$

$$\begin{pmatrix} l_{11} & & & \\ l_{12} & l_{22} & & \\ l_{13} & l_{23} & l_{33} & \\ ... & & & \\ l_{n1} & l_{2n} & ... & l_{nn} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & ... & u_{1n} \\ & u_{22} & ... & u_{2n} \\ & & u_{33} & ... \\ & & & ... \\ & & & u_{nn} \end{pmatrix}$$

This matrix multiplication can be viewed as a system of linear equations. Since the difference between the number of unknown arguments (l_{ij} and u_{ij}) and the number of equations is n, there are n free arguments that can be set to any value. Generally, these n free arguments are chosen to be l_{ii} or u_{ii} ($i = 1, ..., n$) and they are all set to be 1.0. In our work, we set all l_{ii} to 1.0. The other remaining unknown arguments can be derived using the following equations:

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{j-1} l_{jk} u_{kj} \\ l_{ij} &= \frac{1}{u_{jj}} (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}) \end{aligned} \tag{3}$$

In order to improve numerical accuracy, the rows of the original matrix $\mathbf{A}$ are permuted, and we decompose the pivoted (i.e., permuted) matrix $\mathbf{PA}$ instead of the original one. That is, we find $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{PA} = \mathbf{LU}$. The permutation matrix $\mathbf{P}$ is a square binary matrix, in each row and column of which there is exactly one entry that is 1, and 0's elsewhere. Multiplying this matrix into $\mathbf{A}$ will permute the rows or columns of $\mathbf{A}$. It should be noted that pivoting in LU decomposition does not affect the final inverse of matrix $\mathbf{A}$ because we can apply the permutation matrix $\mathbf{P}$ to the product of the inverses of $\mathbf{L}$ and $\mathbf{U}$. That is, we can compute $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$ to obtain the inverse of original matrix $\mathbf{A}$ since $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{PA} = \mathbf{I}_n$. Computing $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$ is equivalent to permuting the columns in $\mathbf{U}^{-1}\mathbf{L}^{-1}$ according to $\mathbf{P}$. The pseudocode of LU decomposition with pivoting is shown in Algorithm 1.

In this algorithm, the result lower triangular matrix and upper triangular matrix are stored in place of the input matrix $\mathbf{A}$. That is,

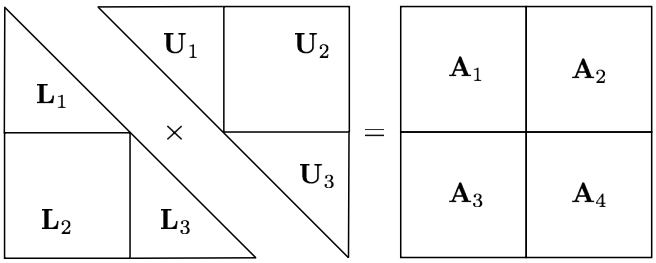

Figure 1: Block method for LU decomposition.

at the end of the algorithm, the lower triangle of $\mathbf{A}$ will be replaced with the lower triangle of $\mathbf{L}$ (excluding the diagonal elements of $\mathbf{L}$, which are all set to 1.0 and do not need to be stored), and the upper triangle of $\mathbf{A}$ will be replaced with the upper triangle of $\mathbf{U}$.

Since there is only one nonzero element in each row or column of the permutation matrix $\mathbf{P}$, the permutation of rows can be stored in an array $\mathbf{S}$, where $[\mathbf{S}]_i$ indicates the permuted row number for the i-th row of $\mathbf{A}$.

After decomposing $\mathbf{A}$ into $\mathbf{L}$ and $\mathbf{U}$, we need to invert $\mathbf{L}$ and $\mathbf{U}$ separately. The inverse of a lower triangular matrix is given by

$$[\mathbf{L}^{-1}]_{ij} = \begin{cases} 0 & \text{for } i < j \\ \frac{1}{[\mathbf{L}]_{ii}} & \text{for } i = j \\ -\frac{1}{[\mathbf{L}]_{ii}} \sum_{k=j}^{i-1} [\mathbf{L}]_{ik} [\mathbf{L}^{-1}]_{kj} & \text{for } i > j \end{cases} \quad (4)$$

The inverse of the upper triangular matrix can be computed similarly. In fact, in our implementation, we invert the upper triangular matrix $\mathbf{U}$ by computing the inverse of $\mathbf{U}^T$, which is a lower triangular matrix, as an optimization (details in Section 5).

4.2 Block Method for Parallel LU Decomposition

The classical LU algorithm is not suitable for parallelization in MapReduce, so we use a block method instead to compute the LU decomposition in parallel. Block methods have been used before for LU decomposition. For example, the tile LU algorithm [1] splits the matrix into square submatrices and updates these submatrices one-by-one. Our block method splits the input matrix as illustrated in Figure 1. In this method, the lower triangular matrix $\mathbf{L}$ and the upper triangular matrix $\mathbf{U}$ are both split into three submatrices, while the original matrix $\mathbf{A}$ is split into four submatrices. These smaller matrices satisfy the following equations:

$$\begin{aligned} \mathbf{L}_1\mathbf{U}_1 &= \mathbf{P}_1\mathbf{A}_1 \\ \mathbf{L}_1\mathbf{U}_2 &= \mathbf{P}_1\mathbf{A}_2 \\ \mathbf{L}_2^{'}\mathbf{U}_1 &= \mathbf{A}_3 \\ \mathbf{L}_3\mathbf{U}_3 &= \mathbf{P}_2(\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2) \\ \mathbf{L}_2 &= \mathbf{P}_2\mathbf{L}_2^{'} \end{aligned} \quad (5)$$

where both $\mathbf{P}_1$ and $\mathbf{P}_2$ are permutations of rows. The entire LU decomposition can be represented as

$$\mathbf{LU} = \begin{pmatrix} \mathbf{P}_1 & \\ & \mathbf{P}_2 \end{pmatrix} \mathbf{A} = \mathbf{PA}$$

where $\mathbf{P}$ is also a permutation of rows obtained by augmenting $\mathbf{P}_1$ and $\mathbf{P}_2$.

This method partitions an LU decomposition into two smaller LU decompositions and two linear equations which can easily be computed in parallel, as we explain next. The logical order of computing the L and U blocks on the left-hand-side of Equation 5 is as follows: First, $\mathbf{L}_1$ and $\mathbf{U}_1$ are computed from $\mathbf{A}_1$. Then, $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ are computed from $\mathbf{L}_1$, $\mathbf{U}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$. Third, $\mathbf{L}_3$ and $\mathbf{U}_3$ are computed from $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$. Finally, $\mathbf{L}_2$ is computed from $\mathbf{L}_2^{'}$ and $\mathbf{P}_2$.

First, let us examine the computation of $\mathbf{L}_1$ and $\mathbf{U}_1$. If submatrix $\mathbf{A}_1$ is small enough, e.g., order of 10^3 or less, it can be decomposed into $\mathbf{L}_1$ and $\mathbf{U}_1$ on a single node very efficiently (about 1 second on a typical modern computer). In our MapReduce implementation, we decompose such small matrices in the MapReduce master node using Algorithm 1.

If submatrix $\mathbf{A}_1$ is not small enough, we can recursively partition it into smaller submatrices as in Figure 1 until the final submatrix is small enough to decompose on a single node. Note that while this is conceptually a recursive computation, the number of partitioning steps (i.e., the depth of recursion) can be precomputed at the start of the matrix inversion process, so that the computation is implemented by a predefined pipeline of MapReduce jobs. In this pipeline, the input matrix is read only once and the partitioned matrix is written only once, as described in Section 5.2.

After obtaining $\mathbf{L}_1$ and $\mathbf{U}_1$, the elements of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ can be computed using the following two equations (for simplicity, we present the equations without pivoting since pivoting does not increase the computational complexity):

$$\begin{aligned} [\mathbf{L}_2^{'}]_{ij} &= \frac{1}{[\mathbf{U}_1]_{ii}} \left([\mathbf{A}_3]_{ij} - \sum_{k=1}^{i-1} [\mathbf{L}_2^{'}]_{ik} [\mathbf{U}_1]_{kj} \right) \\ [\mathbf{U}_2]_{ij} &= \frac{1}{[\mathbf{L}_1]_{ii}} \left([\mathbf{A}_2]_{ij} - \sum_{k=1}^{i-1} [\mathbf{L}_1]_{ik} [\mathbf{U}_2]_{kj} \right) \end{aligned} \quad (6)$$

From these equations, it is clear that the elements in one row of $\mathbf{L}_2^{'}$ are independent of the elements in other rows. Similarly, the elements in one column of $\mathbf{U}_2$ are independent of the elements in other columns. Therefore, each row of $\mathbf{L}_2^{'}$ and each column of $\mathbf{U}_2$ can be computed independently, so we can parallelize the computation of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$. In MapReduce, we can use one map function (multiple copies of which are executed in parallel in multiple map tasks) to compute $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ in parallel. We use the reduce function of the same MapReduce job to compute $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ in parallel in the reduce tasks.

After obtaining $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$, we decompose it into $\mathbf{L}_3$ and $\mathbf{U}_3$. If $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is small enough, $\mathbf{L}_3$ and $\mathbf{U}_3$ are computed on the MapReduce master node using Algorithm 1. Otherwise, $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is further partitioned and the computation proceeds recursively. As with $\mathbf{A}_1$, the number of partitioning steps can be precomputed and the recursive computation can be implemented by a predefined pipeline of MapReduce jobs. One difference between $\mathbf{A}_1$ and $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is that $\mathbf{A}_1$ can be read from the input matrix and completely partitioned into as many pieces as necessary before the LU decomposition starts, while $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ can be computed and partitioned only after $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ are computed. Note that while $\mathbf{A}_1$ and $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ may need additional partitioning, $\mathbf{A}_2$ and $\mathbf{A}_3$ never need additional partitioning due to the easily parallelizable nature of computing $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ using Equation 6.

The pseudocode of block LU decomposition is shown in Algorithm 2. In this algorithm, at the end of the block decomposition, the permutation matrix $\mathbf{P}$ is obtained by augmenting $\mathbf{P}_1$ and $\mathbf{P}_2$. The lower triangular matrix $\mathbf{L}$ is obtained by augmenting $\mathbf{L}_1$, $\mathbf{L}_2^{'}$ permuted by $\mathbf{P}_2$, and $\mathbf{L}_3$. The upper triangular matrix $\mathbf{U}$ is obtained by augmenting $\mathbf{U}_1$, $\mathbf{U}_2$, and $\mathbf{U}_3$ (Figure 1).

Algorithm 2 Block LU decomposition.

1: **function** BlockLUDecom($\mathbf{A}$)
2: **if** A is small enough **then**
3: ($\mathbf{L}, \mathbf{U}, \mathbf{P}$) = LUDecompoistion($\mathbf{A}$)
4: **else**
5: Partition $\mathbf{A}$ into $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3, \mathbf{A}_4$
6: ($\mathbf{L}_1, \mathbf{U}_1, \mathbf{P}_1$) = BlockLUDecom($\mathbf{A_1}$)
7: Compute $\mathbf{U}_2$ from $\mathbf{A}_2$, $\mathbf{U}_1$ and $\mathbf{P}_1$
8: Compute $\mathbf{L}_2^{'}$ from $\mathbf{A}_3$ and $\mathbf{U}_1$
9: Compute $\mathbf{B} = \mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$
10: ($\mathbf{L}_3, \mathbf{U}_3, \mathbf{P}_2$) = BlockLUDecom($\mathbf{B}$)
11: $\mathbf{P}$ = Combination of $\mathbf{P}_1$ and $\mathbf{P}_2$
12: $\mathbf{L}$ = Combination of $\mathbf{L}_1$, $\mathbf{L}_2^{'}$, $\mathbf{L}_3$, and $\mathbf{P}_2$
13: $\mathbf{U}$ = Combination of $\mathbf{U}_1$, $\mathbf{U}_2$ and $\mathbf{U}_3$
14: **end if**
15: **return** ($\mathbf{L}, \mathbf{U}, \mathbf{P}$)

It should be noted that the MapReduce implementation of block LU decomposition consists of a pipeline of MapReduce jobs. MapReduce implementations of other matrix inversion methods such as Gauss-Jordan elimination or QR decomposition would also consist of pipelines of MapReduce jobs. However, a key advantage of block LU decomposition, and the main reason we chose it over other matrix inversion methods, is that the number of iteration steps (i.e., the number of MapReduce jobs in the pipeline) can be reduced by LU decomposing small matrices on one computer (the MapReduce master node). If n_b is the maximum order of a matrix that can be LU decomposed on a single computer in a few seconds, then block LU decomposition would require around n/n_b iterations (modulo rounding if n is not a power of 2 and is not divisible by n_b). In contrast, we were unable to reduce the number of iterations required by other methods such as Gauss-Jordan elimination or QR decomposition below n. The reason is that the computation in these other methods proceeds one vector at a time, and we have n vectors, while the computation in block LU decomposition proceeds on block at a time, and we have around n/n_b blocks.

To quantify the difference in the number of iterations, consider that in our experiments we use $n_b = 3200$. For this n_b, inverting a matrix with $n = 10^5$ requires 32 iterations using block LU decomposition as opposed to 10^5 iterations using, say, QR decomposition.

4.3 Computing the Matrix Inverse

After obtaining the lower triangular matrix $\mathbf{L}$ and the upper triangular matrix $\mathbf{U}$, we can compute the inverses of these two matrices using Equation 4. Inspecting this equation, we can see that a column of the matrix inverse is independent of other columns of the inverse. Therefore, the columns can be computed independently in parallel. After computing the inverses of $\mathbf{L}$ and $\mathbf{U}$, the inverse of the original matrix can be obtained by multiplying $\mathbf{U}^{-1}$ by $\mathbf{L}^{-1}$, which can also be done in parallel, and then permuting the resulting matrix according to array $\mathbf{S}$ (recall that $\mathbf{S}$ is a compact representation of $\mathbf{P}$). That is, $[\mathbf{A}^{-1}]_{[\mathbf{S}]_i j} = \sum_{k=1}^{n} [\mathbf{U}^{-1}]_{ik} [\mathbf{L}^{-1}]_{kj}$.

5. IMPLEMENTATION IN MAPREDUCE

In this section, we discuss the implementation of our algorithm in MapReduce. This implementation is available on GitHub [31]. It involves several steps: (1) We use the master compute node to create some control files in HDFS, which are used as input files for the mappers of all MapReduce jobs. (2) We launch a MapReduce job to recursively partition the input matrix $\mathbf{A}$. (3) We launch a series of MapReduce jobs to compute $\mathbf{L}_2^{'}$, $\mathbf{U}_2$, and $\mathbf{B} = \mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ for the different partitions of $\mathbf{A}$ as in Algorithm 2. Matrices $\mathbf{L}_1$, $\mathbf{U}_1$, $\mathbf{L}_3$, and $\mathbf{U}_3$ are computed in the master nodes of these MapReduce jobs if they are small enough (line 3 in Algorithm 2). Otherwise, they are computed by other MapReduce jobs assigned by the master node (line 15 in Algorithm 2). (4) We launch a final MapReduce job to produce the final output by computing $\mathbf{U}^{-1}$, $\mathbf{L}^{-1}$ and $\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$.

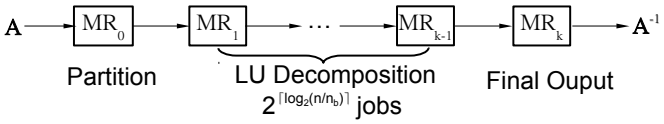

Figure 2: MapReduce pipeline for matrix inversion.

The number of MapReduce jobs required to compute the LU decomposition (Step 3) depends on the order n of matrix $\mathbf{A}$, and on the bound value n_b. Recall that n_b is the order of the largest matrix that can be LU decomposed on a single node (in our case the MapReduce master node). The number of MapReduce jobs is given by $2^{\lceil \log_2 \frac{n}{n_b} \rceil}$. Thus, we have a pipeline of MapReduce jobs as shown in Figure 2. For typical values of n and n_b, the number of MapReduce jobs in this pipeline is small enough that coordination and bookkeeping overheads are minimal. In our experiments, the maximum number of MapReduce jobs in any pipeline was 33.

The bound value should be set so that the time to LU decompose a matrix of order n_b on the master node is approximately equal to the constant time required to launch a MapReduce job. If the running time to decompose a matrix of order n_b on the master node is significantly less than the launch time of a MapReduce job, there will be more MapReduce jobs than necessary and we can increase n_b to reduce the number of MapReduce jobs and also reduce the total running time of LU decomposition. On the other hand, if the running time on the master node is significantly larger than the launch time of a job, the LU decomposition on the master node becomes the bottleneck and we can improve performance by reducing n_b so that we partition the matrix into smaller submatrices. Based on this reasoning and measurements conducted in our experimental environment, we set n_b to 3200 in our experiments.

In our implementation, we use the Java data type **double** (64 bits) for the input matrix and all our calculations. This data type provided sufficient numerical accuracy in all our experiments. We leave a deeper investigation of numerical stability for future work.

Next, we discuss the steps of the implementation. For the purpose of this discussion, Figure 3 shows an example of how the input matrix is partitioned, and Figure 4 shows the HDFS directory structure used by our implementation. HDFS directory "Root" is the work directory, and file "a.txt" in "Root" is the input matrix.

5.1 Input Files on MapReduce

In the first step, the master compute node controlling the entire workflow creates m_0 files in HDFS, where m_0 is the number of compute nodes. These files are used as input files for all launched MapReduce jobs and they are stored in "Root/MapInput/" (purple labels in Figure 4). The purpose of these files is provide an identifier for each mapper, so that the mapper can determine its role based on this identifier. Each file contains one integer: the first file A.0 contains 0, the second file A.1 contains 1, …, and the last file A.$m_0 - 1$ contains $m_0 - 1$. The mappers use these files to control the computation, and they produce the output required for inverting the input matrix by writing directly to HDFS.

5.2 Data Partitioning and I/O Efficiency

In this section, we discuss how the input matrix is partitioned for LU decomposition and how the different partitions flow through the

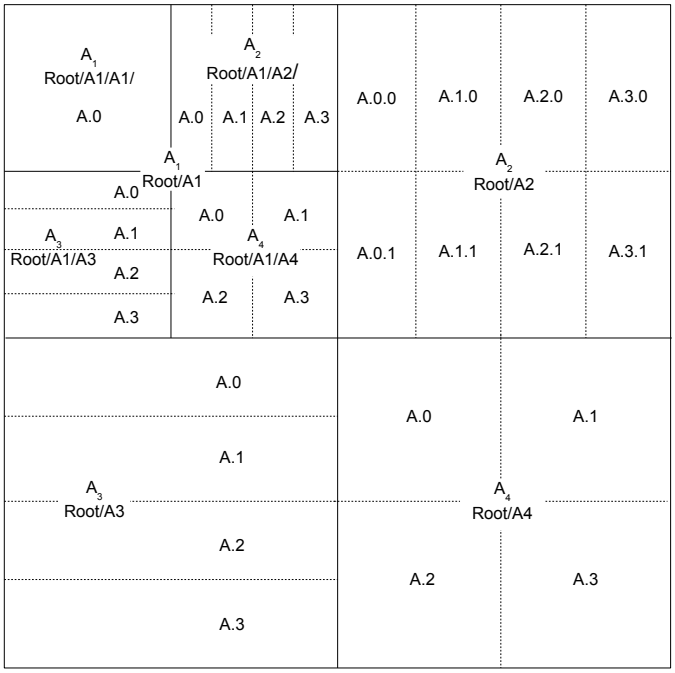

Figure 3: Matrix partitioning for LU decomposition.

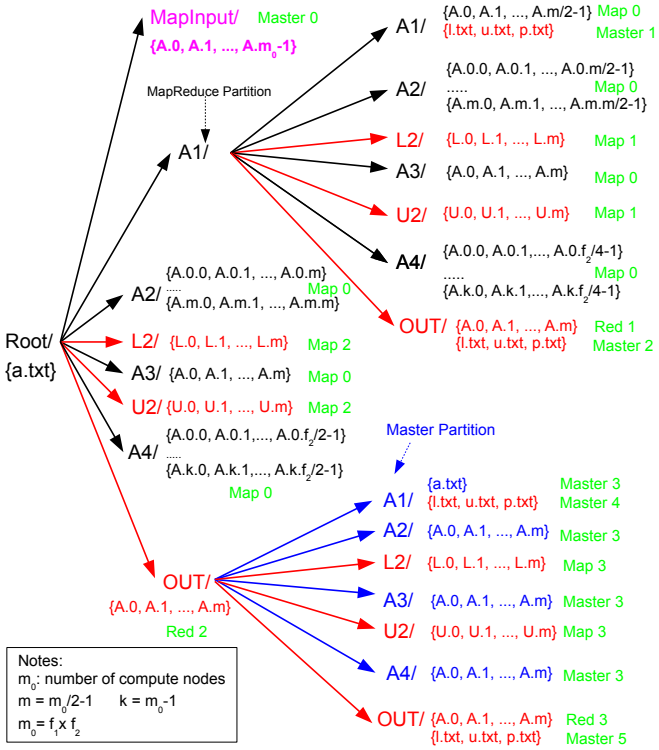

Figure 4: HDFS directory structure used by implementation.

MapReduce pipeline. We also discuss how the input, intermediate, and output data files are stored in HDFS to improve I/O efficiency. In Figure 4, the green labels indicate which process produces each result or intermediate data file. For example "Map 0" means that the result is produced by the mappers in the first MapReduce job, and "Master 0" means that the result is produced by the master node of that job. Therefore, these labels also represent the computation process of the LU decomposition.

We launch a MapReduce job to partition matrix $\mathbf{A}$. This is a map-only job where the mappers do all the work and the reduce function does nothing. This is the only partitioning job, and it recursively partitions $\mathbf{A}$ into as many submatrices as needed, according to the depth of recursion implied by n and n_b. The mappers of this job read their input files from "Root/MapInput/". The integer read by each mapper tells that mapper which rows of the input matrix in the HDFS file "Root/a.txt" to read and partition. In order to improve I/O performance, each map function reads an equal number of consecutive rows from this file to increase I/O sequentiality. Worker j (the mapper assigned input file "Root/MapInput/A.j") reads rows $r_1 = \frac{nj}{m_0}$ to $r_2 = r_1 + \frac{n}{m_0}$ (exclusively). Each block is written to "Root/A1" and "Root/A2" if the block is in the first half of $\mathbf{A}$. Otherwise it is written to "Root/A3" and "Root/A4".

In order to improve I/O efficiency while reading submatrices from disk in subsequent MapReduce jobs, each submatrix, whether $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, or $\mathbf{A}_4$, is split into multiple parts, each of which is stored in a separate file. This ensures that there will never be multiple mappers that simultaneously read the same file. For example, $\mathbf{A}_3$ is stored in $\frac{m_0}{2}$ files because we use only half the compute nodes to compute $\mathbf{L}_2'$ using $\mathbf{A}_3$, while the other half are used to compute $\mathbf{U}_2$ using $\mathbf{A}_2$ (details later). Therefore, $m = \frac{m_0}{2} - 1$ in Figure 4. This approach also ensures that no two mappers write data into the same file, thereby eliminating the need for synchronization between mappers and improving I/O efficiency. Mappers and reducers in subsequent jobs also write their data into independent files, so synchronization on file writes is never required and I/O efficiency is maintained. The separate files written by worker nodes are shown in Figure 4, for example, "L2/L.1".

In Figure 3, which shows an example of matrix $\mathbf{A}$ partitioned by four mappers, the square blocks surrounded by solid lines are submatrices, while the rectangular blocks divided by dashed lines are separate files storing these submatrices. In Figure 4, the black labels within braces are the file names of partitioned submatrices. The value f_2 is the maximum factor of m_0 less than $\sqrt{m_0}$ (see the discussion in Section 6.2). The depth d of the directory structure equals the depth of data partitioning given by $\lceil \log_2 \frac{n}{n_b} \rceil$ (that depth is 2 in Figure 4). The pseudocode of the data partitioning algorithm for LU decomposition is given in Algorithm 3. This listing shows one map function partitioning the block of data in rows r_1 to r_2.

Partitioning submatrix $\mathbf{B} = \mathbf{A}_4 - \mathbf{L}_2'\mathbf{U}_2$ is handled differently from the input matrix. This submatrix is produced by m_0 reducers of a MapReduce job and stored in m_0 files. These files are not read simultaneously by many mappers or reducers, so instead of materializing the data partitions of $\mathbf{B}$ after this submatrix is produced, we only record the indices of the beginning and ending row, and the beginning and ending column, of each partition in this submatrix. We also record the names of the files storing this data. Using this approach, the files in "Root/OUT/A1", "Root/OUT/A2", "Root/OUT/A3", and "Root/OUT/A4" (indicated by blue labels in Figure 4) are very small (in general, less than 1 KB). The running time to partition $\mathbf{A}_4 - \mathbf{L}_2'\mathbf{U}_2$ with such a method is quite short (less than 1 second). Therefore, it is not necessary to launch a MapReduce task to partition this matrix. In our implementation, we partition this matrix in the master node.

5.3 LU Decomposition Using MapReduce

After partitioning $\mathbf{A}$, we use Algorithm 2 to compute the LU decomposition. Since $\mathbf{A}$ has been partitioned, line 5 is ignored, and $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, and $\mathbf{A}_4$ are read from HDFS files. We launch one

Algorithm 3 Data partitioning for LU decomposition.

1: **function** Partition($\mathbf{A}$, r_1, r_2, n, n_b, m, f_1, f_2, "path")
2: /* $\mathbf{A}$ is the original matrix. r_1 is the index of the beginning row to be saved by this function and r_2 is the index of the ending row. n is the order of the matrix. n_b is the bound value for data partitioning. m is the number of map workers partitioning the current submatrix (e.g., the submatrix in the directory Root/A1 shown in Figure 3), f_1 and f_2 indicate that $\mathbf{A}_4$ is partitioned to $f_1 \times f_2$ blocks according to the optimization in Section 6.2. */
3: **if** $n < n_b$ **then**
4: /* Save A1*/
5: Save $[\mathbf{A}]_{[r_1 \ldots r_2][0 \ldots n]}$ to "path/A1/A.$\frac{r_1 m}{n}$"
6: **else**
7: **if** $r_1 < \frac{n}{2}$ **then**
8: Partition($\mathbf{A}$, r_1, r_2, $\frac{n}{2}$, n_b, $\frac{m}{2}$, f_1, f_2, "path/A1")
9: /* Save A2 */
10: **for** $i = 0$ to m - 1 **do**
11: Save $[\mathbf{A}]_{[r_1 \ldots r_2][\frac{n}{2} \ldots n]}$ to "path/A2/A.i.$\frac{r_1 m}{n}$"
12: **end for**
13: **else**
14: /* Save A3 */
15: **for** $i = 0$ to $\frac{(r_2 - r_1)m}{2n} - 1$ **do**
16: $k = \frac{(2r_1 - n)m}{4n} + i$
17: Save $[\mathbf{A}]_{[r_1 \ldots r_1 + \frac{2m(i+1)}{n}][0 \ldots \frac{n}{2}]}$ to "path/A3/A.k"
18: **end for**
19: /* Save A4 */
20: **for** $j = 0$ to $f_2 - 1$ **do**
21: $l = (\frac{2r_1 f_1}{n} - f_1)f_2 + j$
22: **for** $i = 0$ to $\frac{2(r_2 - r_1)f_1}{n} - 1$ **do**
23: Save $[\mathbf{A}]_{[r_1 \ldots r_1 + \frac{(i+1)n}{2f_1}][\frac{n}{2} \ldots \frac{n}{2} + \frac{(j+1)n}{2f_2}]}$ to "path/A4/A.l.i"
24: **end for**
25: **end for**
26: **end if**
27: **end if**

MapReduce job for lines 7–9 of this algorithm. One MapReduce job is sufficient regardless of the size of the input block. The map function of this job computes $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$, while the reduce function computes $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ (red labels in Figure 4). In our implementation, we use half of the mappers to compute $\mathbf{L}_2^{'}$ and the other half to compute $\mathbf{U}_2$, since computing $\mathbf{L}_2^{'}$ has the same computational complexity as computing $\mathbf{U}_2$. Each mapper reads an input file from the directory "Root/MapInput/". If the value in this file is $\leq m = \frac{m_0}{2} - 1$, the mapper computes part of $\mathbf{L}_2^{'}$. Otherwise it computes part of $\mathbf{U}_2$. This is illustrated in Figure 5.

If worker j is computing part of $\mathbf{L}_2^{'}$, this worker is assigned the file "$\mathbf{A}.j$". The worker reads $\mathbf{L}_1$ from HDFS directory "Root/A1" and $\mathbf{A}_{2.j}$ from files "Root/A2/A.j.0, Root/A2/A.j.1, ..., Root/A.$j.m$", and computes one part of $\mathbf{L}_2^{'}$, which is written to "Root/L2/L.j".

Each mapper in this MapReduce job emits one (*key, value*) pair containing (j, j), where j is the value read by the mapper from its input file. These (*key, value*) pairs are used to control which part of $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ each reducer should compute. In our implementation, we use block wrap for matrix multiplication (Section 6.2), so worker j computes the j-th block of $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$. The detailed files read and written are shown in Figure 5. It should be noted

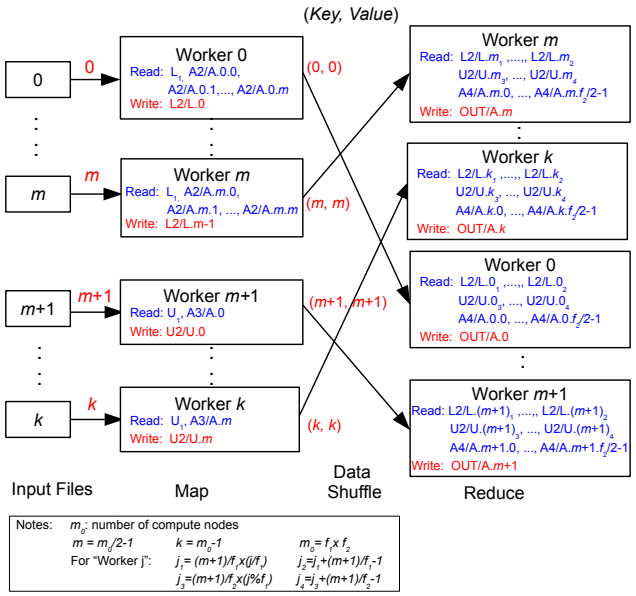

Figure 5: MapReduce job to compute $\mathbf{L}_2^{'}$, $\mathbf{U}_2$, and $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$.

Algorithm	Write	Read	Transfer	Mults	Adds
Our Algorithm	$\frac{3}{2}n^2$	$(l+3)n^2$	$(l+3)n^2$	$\frac{1}{3}n^3$	$\frac{1}{3}n^3$
ScaLAPCK	n^2	n^2	$\frac{2}{3}m_0 n^2$	$\frac{1}{3}n^3$	$\frac{1}{3}n^3$

Table 1: Time complexity of our LU decomposition algorithm, in comparison with the ScaLAPACK algorithm, for $n \times n$ matrix on m_0 compute nodes, where $m_0 = f_1 \times f_2$ and $l = \frac{1}{4}(m_0 + 2f_1 + 2f_2)$.

that after obtaining $\mathbf{L}_2^{'}$ and $\mathbf{P}_2$, $\mathbf{L}_2$ can be easily obtained by permuting $\mathbf{L}_2^{'}$ based on the permutation matrix $\mathbf{P}_2$. Therefore in our implementation, $\mathbf{L}_2$ is constructed only as it is read from HDFS.

The time complexity of our LU decomposition is shown in Table 1. We also present the time complexity of the algorithm used in ScaLAPACK (which we compare to in Section 7.5). Since the data in MapReduce is stored in a distributed environment, the amount of data transferred is one of the main bottlenecks. Therefore, we also show the total data transfer in this table. In our algorithm, all data is written to HDFS, which means that the amount of data read from HDFS is the same as the amount of data transferred between compute nodes. In the MPI implementation of ScaLAPACK, the data is read only once (better than our algorithm), but large amounts of data are transferred over the network between the master and workers (worse than our algorithm).

5.4 Triangular Matrix Inversion and Final Output

One MapReduce job is used to compute the inverses of the triangular matrices $\mathbf{L}$ and $\mathbf{U}$ and the product of their inverses. In the map phase, the inverses of the triangular matrices are computed using Equation 4. Half of the mappers compute the inverse of $\mathbf{L}$ and the other half compute the inverse of $\mathbf{U}$. In order to balance load, the i-th node is used to compute the $(k \times m_0 + i)$-th column of $\mathbf{L}^{-1}$ if i is less than $\frac{n}{2}$. If $i \geq \frac{n}{2}$, the node computes the $(k \times \frac{n}{m_0} + i - \frac{n}{2})$-th row of $\mathbf{U}^{-1}$, where k is an integer ($0 \leq k < \frac{n-i}{m_0}$ for $i < \frac{n}{2}$ or $0 \leq k < \frac{3n-2i}{2m_0}$ for $i \geq \frac{n}{2}$). Thus, each mapper computes an equal number of non-contiguous columns of $\mathbf{L}^{-1}$, which is designed to ensure balanced load. For example, if there are 4 map-

Algorithm	Write	Read	Transfer	Mults	Adds
Our Algorithm	$2n^2$	ln^2	$(l+2)n^2$	$\frac{2}{3}n^3$	$\frac{2}{3}n^3$
ScaLAPCK	n^2	m_0n^2	m_0n^2	$\frac{2}{3}n^3$	$\frac{2}{3}n^3$

Table 2: Time complexity of our triangular matrix inversion and final matrix inversion, in comparison with the ScaLAPACK algorithm, for $n \times n$ matrix on m_0 compute nodes, where $m_0 = f_1 \times f_2$ and $l = \frac{1}{2}(m_0 + f_1 + f_2)$.

pers, *Mapper0* computes columns $0, 4, 8, 12, \ldots$, *Mapper1* computes columns $1, 5, 9, 13, \ldots$, and so on.

In the reduce phase, the product of these two inverses $\mathbf{U}^{-1}\mathbf{L}^{-1}$ is computed. Each reducer reads a number of columns of $\mathbf{L}^{-1}$ and a number of rows of $\mathbf{U}^{-1}$, and multiples these two parts. In order to reduce read I/O, block wrap is used for matrix multiplication (Section 6.2). In order to balance load, instead of partitioning the final matrix into $f_1 \times f_2$ blocks (see Section 6.2), each of which contains consecutive rows and consecutive columns, the matrix is partitioned into grid blocks, each of which contains discrete rows and discrete columns. Worker j computes the product of row $\frac{m_0}{f_1}k_1 + j_1$ of $\mathbf{U}^{-1}$ and column $\frac{m_0}{f_2}k_2 + j_2$ of $\mathbf{L}^{-1}$, where $j_1 = \frac{j}{f_1}$, $j_2 = j \bmod f_1$. Here k_1 is any of the non-negative integers that satisfy $\frac{m_0}{f_1}k_1 + j < m_0$, and k_2 is any of the non-negative integers that satisfy $\frac{m_0}{f_2}k_2 + j < m_0$. As in the implementation of LU decomposition, all outputs, $\mathbf{L}^{-1}, \mathbf{U}^{-1}$ and $\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$, are written to HDFS. The map function only emits integers (j, j) to control the reduce tasks, and does not emit outputs. Table 2 shows the time complexity and data transfer of our matrix inversion algorithm, and the corresponding values in ScaLAPACK.

6. OPTIMIZATIONS OF THE MAPREDUCE IMPLEMENTATION

In this section, we present optimizations that we use to improve the performance of our implementation. Two of these optimizations aim to reduce read and write I/O, namely storing intermediate data in separate files (Section 6.1) and block wrap (Section 6.2). A third optimization aims to improve memory access locality (Section 6.3).

6.1 Storing Intermediate Data in Separate Files

In order to reduce the amount read and write I/O in different MapReduce jobs, we do not combine the results, such as $\mathbf{L}_1$, $\mathbf{L}_2$, and $\mathbf{L}_3$, in any stage. The results are located in many different files as shown in Section 5. Algorithm 2 writes all outputs into HDFS as separate files and skips lines 11–13. The total number of files for the final lower triangular or upper triangular matrix is $N(d) = 2^d + \frac{m_0}{2}(2^d - 1)$, where m_0 is the number of compute nodes, and d is the recursive depth that is constrained by the matrix order n, i.e., $d = \lceil \log_2 \frac{n}{n_b} \rceil$. For example, given a square matrix $\mathbf{A}$ with $n = 2^{15}$, $n_b = 2^{11} = 2048$, and $m_0 = 64$, the recursive depth d is 4 and the final lower triangular matrix $\mathbf{L}$ is stored in $N(d) = 496$ files. In our implementation, these files are read into memory recursively.

Because combining intermediate files can only happen on one compute node, such as the master node, and other compute nodes have to wait until combination is completed, combining intermediate files significantly increases the running time, and this optimization significantly improves performance, as shown in Section 7.3.

6.2 Block Wrap for Matrix Multiplication

Our algorithm requires multiplying two matrices at different stages, for example $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$, or $\mathbf{U}^{-1}$ and $\mathbf{L}^{-1}$. A simple and easy-to-implement way to multiply two matrices while reducing the amount of data read is to use the block method for matrix multiplication. In general, in order to compute $\mathbf{L}_2^{'}\mathbf{U}_2$, each compute node can read a number of rows, e.g., i-th to j-th rows, of $\mathbf{L}_2^{'}$ and the entire matrix $\mathbf{U}_2$. This compute node can then compute the i-th to j-th rows of $\mathbf{L}_2^{'}\mathbf{U}_2$. If the number of compute nodes is m_0, the amount of data read in each node is $(1 + \frac{1}{m_0})n^2$ and the total data read is $(m_0 + 1)n^2$.

There is a better method to multiply two matrices, called the *block wrap* method [6], which reduces the amount of data read. In this method, $\mathbf{L}_2^{'}$ is divided into f_1 blocks, each of which contains $\frac{n}{f_1}$ consecutive rows, while $\mathbf{U}_2$ is divided into f_2 blocks, each of which contains $\frac{n}{f_2}$ consecutive columns. Using this partitioning, every block of $\mathbf{L}_2^{'}$ will need to be multiplied by every block of $\mathbf{U}_2$, and the final matrix is partitioned into $f_1 \times f_2$ blocks. Each of these blocks is computed by one compute node. That is, each compute node reads $\frac{n}{f_1}$ rows of $\mathbf{L}_2^{'}$ and $\frac{n}{f_2}$ columns of $\mathbf{U}_2$ (one block from each matrix) and computes the product of these two block. f_1 and f_2 are chosen so that $m_0 = f_1 \times f_2$. The data read in each compute node is $(\frac{1}{f_1} + \frac{1}{f_2})n^2$, and the total data is $(f_1 + f_2)n^2$, which is significantly less than $(m_0 + 1)n^2$. In order to obtain the minimum data read, we compute f_1 and f_2 from n such that $|f_1 - f_2|$ is as small as possible. That is, we choose $f_2 \leq f_1$, and there is no other factor of m_0 between f_1 and f_2. For example, given 64 nodes, in the naive algorithm each node reads data of size $\frac{65}{64}n^2$, and the total data read for all 64 nodes is $65n^2$. Using the block wrap method and $f_1 = f_2 = 8$, each node reads data of size $\frac{1}{4}n^2$, and the total data read for all nodes is $16n^2$, much better than the naive algorithm.

6.3 Storing Transposed U Matrices

In general, matrices $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ are linearized in row-major order both in memory and in HDFS. The product of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ is computed as follows:

$$[\mathbf{L}_2^{'}\mathbf{U}_2]_{ij} = \sum_{k=1}^{n} [\mathbf{L}_2^{'}]_{ik} \times [\mathbf{U}_2]_{kj} \tag{7}$$

However, when the order of the matrices n is large, each read of an element from $\mathbf{U}_2$ will access a separate memory page, potentially generating a TLB miss and a cache miss. If a page can hold k data items, this access pattern can generate up to $n^3 + \frac{k+1}{k}n^2$ misses for data read and matrix multiplication.

In our implementation, the upper triangular matrix is always stored in a transposed fashion, i.e., we store $\mathbf{U}^T$ instead of $\mathbf{U}$. The product of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2^T$ can be computed as follows:

$$[\mathbf{L}_2^{'}\mathbf{U}_2]_{ij} = \sum_{k=1}^{n} [\mathbf{L}_2^{'}]_{ik} \times [\mathbf{U}_2^T]_{jk} \tag{8}$$

The number of misses can be reduced to $\frac{n^3}{k} + \frac{2n^2}{k}$, which is significantly less than the unoptimized implementation and can substantially improve performance.

7. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our technique. We study the scalability of our algorithm and implementation, the impact of our proposed optimizations, and we compare to a state-of-the-art alternative, namely ScaLAPACK.

7.1 Experimental Environment

We implemented our algorithm on Hadoop 1.1.1, the latest stable version at the time the work was done. All experiments were

Matrix	Order	Elements (Billion)	Text (GB)	Binary (GB)	Number of Jobs
$\mathbf{M}_1$	20480	0.42	8	3.2	9
$\mathbf{M}_2$	32768	1.07	20	8	17
$\mathbf{M}_3$	40960	1.68	40	16	17
$\mathbf{M}_4$	102400	10.49	200	80	33
$\mathbf{M}_5$	16384	0.26	5	2	9

Table 3: Five matrices used for the experiments.

performed on medium instances of Amazon's Elastic Compute Cloud (EC2) [11], except for the largest matrix $\mathbf{M}_4$, for which large instances were used. Each medium instance has 3.7 GB of memory and 1 virtual core with 2 EC2 compute unit, where each EC2 compute unit has performance similar to a 2007-era 1.0–1.2 GHz AMD Opteron or Xeon processor.

We use five matrices in our experiments. We were not able to find large, real-world, public benchmark matrices for which matrix inversion is required, so all of our test matrices were randomly generated using the Random class in Java. Note that the performance of our algorithm depends on the order of the input matrix and not on the data values in this matrix, so performance is not expected to change if we use real-world matrices instead of synthetic ones.

Details about the matrices used are shown in Table 3, which shows the order of each matrix, the number of elements (data type double), the size of the matrix in text format, and the size in binary format. Recall that the bound value n_b used in our experiments is 3200. Table 3 shows, for this value of n_b, the total number of MapReduce jobs required for inverting each matrix. The matrices are stored in HDFS with the default replication factor of 3.

7.2 Algorithm Scalability

In this section, we investigate the scalability of our proposed algorithm, focusing on strong scalability. The running time versus the number of EC2 instances for three matrices is shown in Figure 6. One ideal scalability line (i.e., running time proportional to 1 over the number of nodes) has been over-plotted on this figure in order to demonstrate the scalability of our algorithm. We can see that our algorithm has very good strong scalability, with a minor deviation from ideal scalability when the number of nodes is high. This deviation is due to the constant launch time of MapReduce jobs, since our algorithm uses multiple MapReduce jobs. However, we also note that the number of MapReduce jobs is proportional to the matrix order n, while the running time is proportional to n^3. Therefore we can expect that the larger the matrix, the better the algorithm scalability, which can be seen in Figure 6.

We investigated improving scalability by using systems that support iterative MapReduce computations, such as HaLoop [5]. However, we found that HaLoop and similar systems do not reduce the launch time of MapReduce jobs. HaLoop maintains intermediate state between MapReduce jobs, which is not useful for our algorithm. There are techniques for reducing the overhead of launching MapReduce jobs, such as having pools of worker processes that are reused by map and reduce tasks. These techniques can definitely benefit our work, but they do not require any changes to the matrix inversion MapReduce pipeline. Specifically, our analysis of how finely to decompose the computation holds even under faster job launching.

In order to verify the correctness of our implementation and check whether the data type double is precise enough, we compute $\mathbf{I}_n - \mathbf{M}\mathbf{M}^{-1}$ for matrices $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$, and $\mathbf{M}_5$. We find that every element in the computed matrices is less than 10^{-5}, which validates our implementation and shows that the data type double is sufficiently precise.

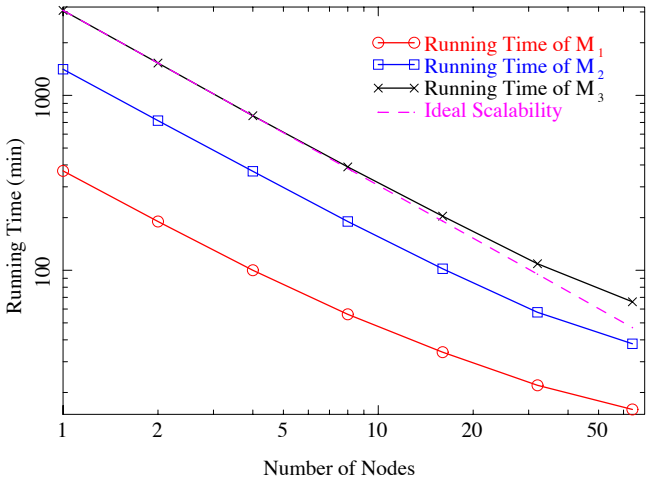

Figure 6: The scalability of our algorithm, in comparison with ideal scalability (purple line), which is defined as $T(n) = T(1)/n$, where $T(n)$ is the running time on n medium EC2 instances.

The largest matrix $\mathbf{M}_4$ is used to further test the scalability limits of our algorithm (Section 7.4), and the smallest matrix $\mathbf{M}_5$ is used to evaluate our optimizations in the next section.

7.3 Evaluating the Optimizations

Our first proposed optimization is storing intermediate data in separate files. Without this optimization, we combine all separate files of $\mathbf{L}$ and $\mathbf{U}$ in each MapReduce job in our iterative MapReduce process. The combination happens in the master node and the combined file is written by that node into HDFS. Since the combination is a serial process done on one node, it takes a constant time to combine the files independent of the number of the compute nodes. Therefore, we can expect that the benefit of keeping separate intermediate files increases as the number of compute nodes increases, since the running time gets smaller and the overhead of combination remains constant.

To validate this expectation, we conduct an experiment with matrix $\mathbf{M}_5$ in which we compare the time taken by our optimized algorithm to the time taken by the algorithm that combines $\mathbf{L}$ and $\mathbf{U}$ files at each step. The ratio of the unoptimized running time to the optimized running time for 4–64 nodes is shown in Figure 7. The figure shows the unoptimized version to be close to 30% slower in some cases, demonstrating the importance of this optimization.

As mentioned in Section 6.2, the block wrap method can significantly reduce read I/O, thereby improving performance. In this section, we use matrix $\mathbf{M}_5$ to evaluate the effect of using block wrap on performance. As before, we measure the running time without this optimization on 4–64 compute nodes and compare this to the running time with the optimization. The improvement is shown in Figure 7. The figure shows that the larger the number of compute nodes, the larger the improvement in performance.

We did not systematically evaluate the third optimization (transpose storing) in our experiments. It is a simple and intuitive optimization, and our experience is that it greatly improves the performance of our algorithm, by a factor of 2–3.

7.4 Scaling to a Very Large Matrix

In this section, we study the ability of our algorithm to invert a very large matrix, namely $\mathbf{M}_4$, which is a matrix of order 102400. We measure the running time on 128 Amazon EC2 large instances, each of which has two medium CPU cores, for a total of 256

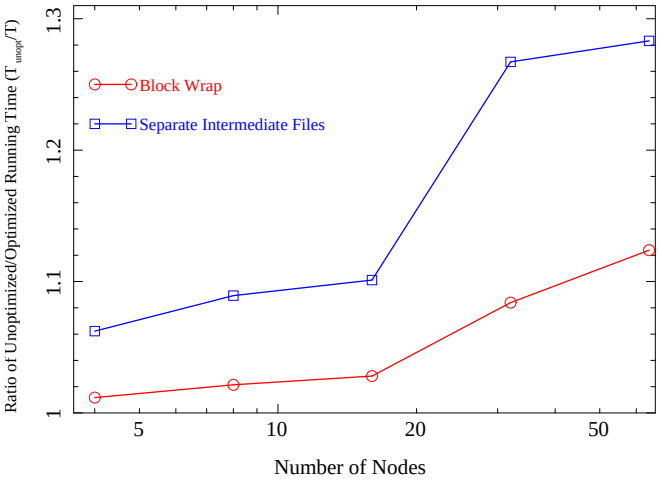

Figure 7: The running time of the optimized algorithm compared to the algorithm without the separate intermediate files optimization (blue), and without the block wrap optimization (red).

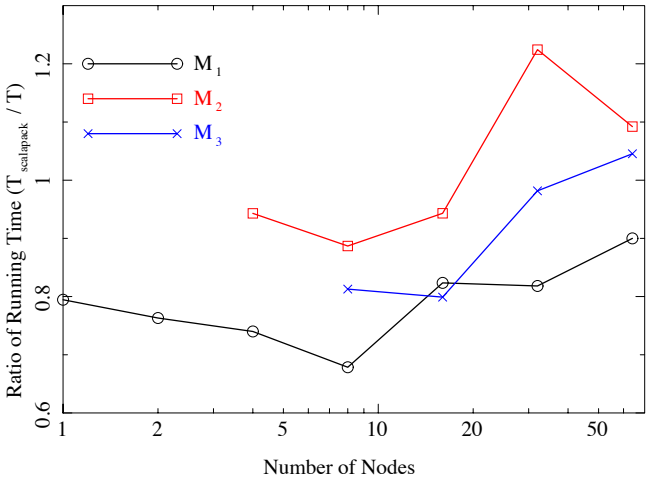

Figure 8: The ratio of the running time of ScaLAPACK to the running time of our algorithm.

medium CPU cores. A medium CPU core has performance similar to two 2007-era 1.0–1.2 GHz AMD Opteron or Xeon processors.

We executed two runs of our algorithm to invert this large matrix. In the first run, it took about 8 hours to solve the problem. During this run, one mapper computing the inverse of a triangular matrix failed and this mapper did not restart until one of the other mappers finished. This increased the running time. However, this failure recovery is a good demonstration of the benefit of using a fault tolerant framework like MapReduce for large scale problems. In the second run, there were no failures and it took about 5 hours to invert the matrix.

The large matrix is about 80 GB in size in binary representation. Our algorithm on the EC2 large instances writes more than 500 GB of data and reads more than 20 TB of data in the 33 MapReduce jobs required to invert the matrix.

We also used 64 medium EC2 instances to invert this matrix. It took about 15 hours in this case to invert the matrix. Analyzing the scalability of the medium instances compared to the large instances, we see that the medium instances show better scalability, based on a simple calculation as follows. Assume for simplicity that each medium instance core has similar compute performance to a large instance core. When we used 128 large EC2 instances we were using 256 cores, whereas when we used 64 medium instances, we were using 64 cores. Thus, we have four times as many cores when using large instances. Therefore, if our algorithm has ideal scalability, the running time in large instances should be $15/4 = 3.8$ hours (four times the cores should result in $\frac{1}{4}$ the running time). However, the running time we observed on large instances (5 hours) is longer than this time that assumes ideal scalability. There are two possible reasons related to EC2 for the running time being longer than expected. The first reason is that we found that the performance variance between different large EC2 instances is high, even though the instances are supposed to have similar performance. The second reason is that the data read speed on some large instances is less than the speed on the medium instances. We found that the speed of copying files between large instances is around 30–60 MB/s, while the speed of copying files between medium instances is around 60 MB/s. The main point of the experiment is that we are able to scale to such large scales in terms of both input size and number of nodes, and that this scalability holds in different runs on different cluster sizes, even in the presence of failures.

7.5 Comparison to ScaLAPACK

ScaLAPACK is a popular library of high-performance linear algebra routines for distributed memory message passing computers and clusters. ScaLAPACK is an extension of LAPACK [2], and it has been shown to have good scalability and performance. More details about ScaLAPACK can be found in [4]. In this section, we compare our matrix inversion technique to ScaLAPACK, to see how we stack up against a state-of-the-art competitor.

In this experiment, the package libscalapack-mpi-dev in Ubuntu is used. The version of MPI used is MPICH [20]. The drive routines PDGETRF and PDGETRI in ScaLAPACK are used to compute the LU decomposition and the triangular matrix inverse respectively. In order to reduce the data transfer between compute nodes in ScaLAPACK, we use an optimization similar to our *block wrap* optimization. In particular, we set the process grid to $f_1 \times f_2$, where $m_0 = f_1 \times f_2$ is the number of compute nodes, $f_1 < f_2$, and there is no factor of m_0 between f_1 and f_2, which means that the matrix is partitioned into $f_1 \times f_2$. The matrix is first partitioned into blocks of dimension 128×128, since we found that this size provides the best performance in our experiments. Next, these blocks are assigned to the process grid. In order to improve load balancing, the blocks are assigned as follows: the block in row $f_1 \times m_1 + i$ and column $f_2 \times m_2 + j$ is assigned to the $(f_2 \times j + i)$-th compute node, where m, n, i and j are integers that are constrained by following inequalities: $f_1 \times m_1 + i < \frac{n}{128}$, $f_2 \times m_2 + j < \frac{n}{128}$, $i < f_1$, and $j < f_2$, where n is the order of the matrix. In our ScaLAPACK implementation, all intermediate data is stored in memory, such that the matrix is read only once and written only once.

The ratio of the running time of ScaLAPACK to the running time of our algorithm on medium EC2 nodes for matrices $\mathbf{M}_1$ to $\mathbf{M}_3$ is shown in Figure 8. The figure shows that for these small matrices, there is a slight performance penalty for using our algorithm compared to ScaLAPACK. We expect ScaLAPACK to perform well since it is a highly optimized state-of-the-art library. The point of Figure 8 is to show that the performance of our MapReduce matrix inversion algorithm is comparable to ScaLAPACK. The advantages of using the popular and widely deployed MapReduce framework, such as scalability, fault tolerance, ease of programming, and rich software ecosystem, do not come at a high cost in terms of performance.

Figure 8 shows that our algorithm approaches or outperforms ScaLAPACK for larger matrices and a larger number of nodes. That is, our algorithm has better scalability than ScaLAPACK. To

better demonstrate the scalability of our algorithm compared to ScaLAPACK, we run another experiment in which we use ScaLAPACK on 128 large EC2 instances (256 CPU cores) and 64 medium EC2 instances (64 CPU cores) to compute the inverse of the largest matrix, $\mathbf{M}_4$. These are the same cluster sizes that we used with our algorithm for this matrix in the previous section. ScaLAPACK took 8 hours on the large instances and more than 48 hours on the medium instances to invert matrix $\mathbf{M}_4$, both of which are significantly longer than our results reported in Section 7.4 (5 hours on large instances and 15 hours on medium instances). It should be noted that the running time of our algorithm on the large instances with a failure (8 hours) is similar to the running time of ScaLAPACK without failure. Thus, while our algorithm may have a small performance penalty compared to ScaLAPACK at low scale, it has better scalability and performance than ScaLAPACK at high scale.

The reason that our algorithm is better than ScaLAPACK at high scale is that ScaLAPACK transfers large amounts of data over the network (Tables 1 and 2). At low scale, the network can accommodate the required bandwidth, but it becomes a bottleneck at high scale. In addition, MapReduce scheduling is more effective than ScaLAPACK at keeping the workers busy. The scalability of ScaLAPACK scheduling is not an issue at low scale, but it is a limitation at high scale.

8. CONCLUSIONS AND FUTURE WORK

We presented a scalable and fault tolerant algorithm for matrix inversion using MapReduce. Our algorithm relies on recursive block LU decomposition to partition the computation in a way that is suitable for parallelization in a cluster environment using MapReduce. The algorithm uses a pipeline of MapReduce jobs to partition the input matrix, compute the LU decomposition of this matrix, and compute the inverse of the input matrix from its LU decomposition. We implemented this algorithm in Hadoop, and presented optimizations to improve its I/O efficiency and memory locality. Our experimental evaluation on Amazon EC2 shows that our algorithm has better scalability than the state-of-the-art ScaLAPACK package, inverting a large matrix of order 102400 in 5 hours on 128 Amazon EC2 large instances.

Many of the lessons and techniques in this paper may be applicable to matrix operations other than matrix inversion, such as finding eigenvalues or principal component analysis. For example, we have identified LU decomposition from among different matrix decompositions as suitable for MapReduce. The block computation style in Algorithm 2 may be useful for other matrix operations. Other matrix operations may also benefit from the matrix partitioning style in Figure 3 and Algorithm 3, and the corresponding file layout in Figure 4. The scheduling of tasks into MapReduce jobs and the optimizations in Section 6 may also be useful.

One promising direction for future work is to implement our matrix inversion technique on the Spark system [34]. In our implementation using Hadoop, all intermediate data, such as $\mathbf{L}_1$ and $\mathbf{U}_1$, is written to HDFS files by one MapReduce job and read from these HDFS files by the next MapReduce job in the pipeline. Therefore, the amount of read I/O in our algorithm is much larger than other algorithms that keep intermediate data in memory, such as the MPI implementation of ScaLAPACK (see Table 1). In Hadoop, it is not easy to keep intermediate data in memory and still preserve fault tolerance. On the other hand, Spark provides parallel data structures that allow users to explicitly keep data in memory with fault tolerance. Therefore, we expect that implementing our algorithm in Spark would improve performance by reducing read I/O. What is promising is that our technique would need minimal changes (if any) in order to be implemented on Spark, and the scalability of our technique would be even better on Spark than it is on Hadoop.

A relatively recent development in the Hadoop ecosystem is the use of resource managers such as Hadoop YARN [32] and Mesos [16]. These resource managers make it easier to run MPI-based packages such as ScaLAPACK alongside MapReduce on the same cluster. Thus, it would be interesting to investigate the conditions under which to use ScaLAPACK or MapReduce for matrix inversion, and to implement a system to adaptively choose the best matrix inversion technique for an input matrix. An important question would be if the performance benefit obtained from ScaLAPACK is high enough to warrant the administrative overhead and programming complexity of installing and operating two frameworks (MPI and MapReduce) on the same cluster.

9. REFERENCES

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. In *Proc. ACS/IEEE Int. Conf. on Comp. Systems and Applications (AICCSA)*, 2011.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Proc. Conf. on Supercomputing (SC)*, 1990.

[3] P. Bientinesi, B. Gunter, and R. A. v. d. Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Mathematics Software*, 35(1):3:1–3:22, 2008.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1997.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow. (PVLDB)*, 3(1-2):285–296, 2010.

[6] K. Dackland, E. Elmroth, B. Kågström, and C. V. Loan. Design and evaluation of parallel block algorithms: LU factorization on an IBM 3090 VF/600J. In *Proc. SIAM Conf. on Parallel Proc. for Scientific Computing*, 1991.

[7] B. De Schutter and B. De Moor. The QR decomposition and the singular value decomposition in the symmetrized max-plus algebra. In *Proc. European Control Conf. (ECC)*, 1997.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Comm. ACM*, 51(1):107–113, 2008.

[9] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. High performance matrix inversion based on LU factorization for multicore architectures. In *Proc. ACM Int. Workshop on Many Task Computing on Grids and Supercomputers*, 2011.

[10] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:2003–2016, 2003.

[11] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[12] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2011.

[13] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2011.

[14] G. H. Golub and C. F. Van Loan. *Matrix computations*. Johns Hopkins University Press, third edition, 1996.

[15] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library: or MAD skills, the SQL. *Proc. VLDB Endow. (PVLDB)*, 5(12):1700–1711, 2012.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, 2011.

[17] K. K. Lau, M. Kumar, and S. Venkatesh. Parallel matrix inversion techniques. In *Proc. IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing*, 1996.

[18] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, 2013.

[19] S. Lupke. LU-decomposition on a massively parallel transputer system. In *Proc. Int. Conf. on Parallel Architectures and Languages*, 1993.

[20] E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2010.

[22] D. Marks, L. Colwell, R. Sheridan, T. Hopf, A. Pagnani, R. Zecchina, and C. Sander. Protein 3d structure computed from evolutionary sequence variation. *PLoS One*, 6(12):e28766–e28766, 2011.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[24] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. MadLINQ: Large-scale distributed matrix computation for the cloud. In *Proc. ACM European Conf. on Computer Systems (EuroSys)*, 2012.

[25] E. S. Quintana, G. Quintana, X. Sun, and R. vande Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2000.

[26] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng. HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment. In *Proc. IEEE Int. Conf. on Cluster Computing and Workshops*, 2009.

[27] M. Wall, A. Rechtsteiner, and L. Rocha. Singular value decomposition and principal component analysis. In *A Practical Approach to Microarray Data Analysis*. Springer, 2003.

[28] R. J. Warp, D. J. Godfrey, and J. T. Dobbins III. Applications of matrix inversion tomosynthesis. *Physics of Medical Imaging*, 3977:376–383, 2000.

[29] J. Xiang. Scalable scientific computing algorithms using MapReduce. Master's thesis, University of Waterloo, 2013.

[30] J. Xiang, S. N. Zhang, and Y. Yao. Probing the spatial distribution of the interstellar dust medium by high angular resolution X-ray halos of point sources. *The Astrophysical J.*, 628:769–779, 2005.

[31] J. Xinag, H. Meng, and A. Aboulnaga. Source code of "Scalable Matrix Inversion Using MapReduce". https://github.com/JingenXiang/MatrixInversion.

[32] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. Symp. on Operating Systems Design and Implementation (OSDI)*, 2008.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, 2012.

[35] Y. Zhang and J. Yang. Optimizing I/O for big array analytics. *Proc. VLDB Endow. (PVLDB)*, 5(8):764–775, 2012.

[36] Q. Zheng, X. Wu, M. Fang, H. Wang, S. Wang, and X. Wang. Application of HPMR in parallel matrix computation. *Comp. Engineering*, 36(08):49–52, 2010.