This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

Spark-based Large-scale Matrix Inversion for Big Data Processing

JUN LIU*, (Member, IEEE), YANG LIANG*, NIRWAN ANSARI†, (Fellow, IEEE)
*Center for Data Science, Beijing University of Posts and Telecommunications, Beijing, 100876, China
†Electrical and Computer Engineering Department, New Jersey Institute of Technology, New Jersey, 07102, US

***Abstract*—Matrix inversion is a fundamental operation for solving linear equations for many computational applications, especially for various emerging big data applications. However, it is a challenging task to invert large-scale matrices of extremely high order (several thousands or millions), which are common in most web-scale systems such as social networks and recommendation systems. In this paper, we present a LU decomposition-based block-recursive algorithm for large-scale matrix inversion. We present its well-designed implementation with optimized data structure, reduction of space complexity and effective matrix multiplication on the Spark parallel computing platform. The experimental evaluation results show that the proposed algorithm is efficient to invert large-scale matrices on a cluster composed of commodity servers and is scalable for inverting even larger matrices. The proposed algorithm and implementation will become a solid foundation for building a high-performance linear algebra library on Spark for big data processing and applications.**

***Index Terms*—matrix inversion, LU decomposition, linear algebra, parallel algorithm, distributed computing, Spark**

I. Introduction

Theoretically, a set of linear equations can be represented as $Ax = b$, where A is an $n \times n$ matrix, x and b are $n \times 1$ vectors. This equation can be solved by computing the inverse of the matrix A, denoted by A^{-1}, to get $x = A^{-1} \times b$. Therefore, matrix inversion is an essential computation task in many data scientific applications, such as signal processing, complex network analysis and collaborative recommendation. For general matrices, there exist some commonly available matrix inversion algorithms like Gaussian elimination, Gauss-Jordan [1], Cholesky decomposition [2] and LU Decomposition [3]. However, these algorithms are computation-intensive that require a cubic number of operations. Therefore, they are not applicable to invert large-scale matrices of dimension in the order of thousands or millions, often required in emerging big data applications. For example, in real-life social networks (like Facebook and Twitter), e-commerce websites (like Amazon and Ebay) and online-video service providers (like Youtube and Netflix), various types of matrices, including following matrices, transaction matrices and rating matrices, contain millions of distinct users and items. Inversion of these large-scale matrices is a fundamental operation for proximity measurement, link prediction and personalized recommendation tasks. This calls for the application of parallel computing techniques in the inversion of large-scale matrices. Message Passing Interface (MPI) is proved to be an effective programming model to support parallel matrix inversion jobs [4]. In recent years, a number of new distributed computing technologies have emerged as platforms for data intensive computing tasks. MapReduce [5] and Spark [6] are two most popular ones owing to their outstanding scalability and fault-tolerance capabilities. Xiang, *et al.* [7] proposed and implemented a scalable matrix inversion algorithm based on MapReduce. As compared to MapReduce, Spark provides a novel data abstraction called resilient distributed datasets (RDDs) based on distributed memory for efficient data reuse, which improves the performance of iterative computing jobs. However, Spark only supports relatively coarse-grained transformation on RDDs. It leads to challenges in the design and implementation of complex matrix operation algorithms, and thus calls for new ideas and solutions.

In this paper, we present a LU decomposition based algorithm for large-scale matrix inversion and its implementation on Spark. We carry out a block-recursive approach to break down the huge inversion computation on the original large-scale matrix into a set of small tasks, which can be executed as a pipeline of Spark tasks on a cluster. The Spark implementation of the proposed algorithm as well as the MapReduce-based algorithm proposed in [7] and an MPI-based program are evaluated on clusters with different configurations. The comparison with the MapReduce-based algorithm shows that our algorithm achieves remarkable performance improvement, and that to the MPI-based program demonstrates that the Spark implementation is more robust on heterogenous and unreliable clusters. However, it is noted that this paper is not trying to prove that Spark is fundamentally superior to MPI or MapReduce. Our goal is to introduce: 1) a novel matrix inversion algorithm along with its well-designed Spark implementation, which will become a fundamental computational component to build a linear algebra library for big data science, 2) the underlying mathematical principles for high-performance matrix inversion like block-oriented data structure for efficient accessing matrix elements and solving L^{-1} and U^{-1} instead of L and U to reduce the computation and space complexity, and 3) the experimental performance evaluation and analysis of Spark-based, MapReduce-based and MPI-based matrix inversion algorithms on local clusters composed of commodity servers under different scenarios. This paper is an elaborated and extended version of a conference paper presented at IEEE INFOCOM 2016 workshop of Big Data Sciences, Technologies and Applications [8]. The key additions of this version include: (1) we propose and illustrate the theories and pseudo-codes of optimized algorithms in

Section II, (2) we detail the key points of implementation on Spark in Section III, (3) we perform an additional experiment on an extra large-scale matrix sized 102400 and present the execution times on two clusters in Section IV, and (4) we extend the survey of related works including the researches of using Graphics Processing Unit to accelerate the matrix computation in Section V.

The rest of this paper is structured as follows. In Section II, we present the basic and the optimized LU decomposition based block-recursive matrix inversion algorithm. Section III introduces the key points of its implementation on Spark. We demonstrate the performance and the scalability of the proposed algorithm, substantiated with experimental evaluation results, in Section IV. The related work is discussed in Section V. In Section VI, we conclude and point out future works.

II. Related Work

Owing to its fundamental role in scientific computing, matrix inversion is widely supported in several numerical analysis softwares like Matlab, R, LINPACK [9] and LAPACK [10]. Although these softwares provide basic matrix inversion capabilities for solving linear equations, they have performance issue when the order of the matrix to be inverted becomes huge. Therefore, developing parallel algorithms for inverting large-scale matrix is always an important issue in data science research community.

Lau *et al.* [11] proposed two algorithms based on Gaussian elimination to invert sparse, symmetric and positive definite matrices on parallel computers, and implemented these two algorithms on SIMD and MIMD computers. However, these algorithms do not work for general matrices. ScaLAPACK [4] extended LAPACK to perform large-scale dense matrix computation on shared-memory supercomputers. It achieved the goal of scalability to keep the computing task efficient when the number of processors increases, but it does not provision fault-tolerance capability. Bientinesi *et al.* [12] proposed an algorithm to invert a symmetric positive definite matrix based on Cholesky factorization. Although the experimental evaluation results showed that the proposed algorithm outperforms ScaLAPACK by improving load-balance on a distributed memory environment, this algorithm cannot be used to invert general matrices. Agullo, *et al.* [13] proposed an efficient and scalable tile algorithm to invert a symmetric positive definite matrix. They used a dynamic scheduler to orchestrate the tasks in the process of inverting a matrix for fine granularity parallelism and asynchronous scheduling, but it does not work for general matrices. Dongarra, *et al.* [14] designed a LU factorization based algorithm for inverting general square matrices on multicore computer architecture. The implementation shows good performance, but it is not suitable for clusters. Yang, *et al.* [15] proposed a parallel algorithm for matrix inversion based on Gauss-Jordan elimination with pivoting. However, the implementation relies on some specific DSP hardware, which limits its applications.

In recent years, Graphics Processing Unit (GPU) has been proved having potentials to perform computationally intensive tasks. Peter *et al.* [16] proposed to accelerate the computation of matrix inversion by connecting a GPU to general-purpose multi-core processors and implemented a matrix inversion algorithm based on Gauss-Jordan elimination on a hybrid architecture consisting of one (or more) multi-core general processors connected to several GPUs. The evaluation results showed that the proposed architecture can achieve remarkable high performance. Sharma *et al.* [17] modified the Gauss-Jordan algorithm for matrix inversion by leveraging the large scale parallelization capability of a massively multithreaded GPU. The algorithm was implemented on a Computer Unified Device Architecture (CUDA) platform. Although the above works have demonstrated that GPU can considerably reduce the computational time of matrix inversion, they are non-scalable centralized methods and need special hardwares.

To break the resource limitation of single server, Caron and Utard [18] designed a LU factorization based parallel out-of-core algorithm to invert huge matrices and implemented the algorithm based on the ScaLAPACK library in cluster environment. However, owing to the heavy communications and I/O overheads, the algorithm is slow to invert huge matrices. For example, it needs 3 days to invert a matrix of order 10,000 on a cluster of 16 servers. As cloud computing technologies emerges, several new data processing platforms have been developed for large-scale data processing. MapReduce and Spark are two outstanding technologies, which have been proved to be efficient for big data processing on clusters composed of commodity servers with scalability and fault-tolerance [19]. Building linear algebra functionality on these platforms becomes a problem of great interest. HAMA [20] and linalg [21] are such kind of efficient matrix computation software packages based on MapReduce and Spark, respectively. However, they do not provide matrix inversion function. The work of [7] was the first MapReduce-based matrix inversion algorithm, but the inefficient hard disk based intermediated data mechanism limits its performance.

III. Algorithm Design and Optimization

A. Matrix Inversion based on LU Decomposition

The inverse of a square matrix $A = [a_{ij}]_{1 \leqslant i,j \leqslant n}$ is denoted as A^{-1} such that $AA^{-1} = I_n$, where I_n is the identity matrix of dimension $n \times n$. The LU decomposition method computes the inverse of a matrix A by factorizing the original matrix into two matrices $L = [l_{ij}]_{1 \leqslant i,j \leqslant n}$ and $U = [u_{ij}]_{1 \leqslant i,j \leqslant n}$ such that $A = LU$, where L is a lower triangular matrix (i.e., $l_{ij} = 0 \ for \ 1 \leqslant i < j \leqslant n$) and U is an upper triangular matrix (i.e., $u_{ij} = 0 \ for \ 1 \leqslant j < i \leqslant n$). From this decomposition, the equation $AA^{-1} = I_n$ can be transformed to $LUA^{-1} = I_n$, and the inverse matrix A^{-1} can be simply computed by $A^{-1} = U^{-1}L^{-1}$.

In some cases, the LU factorization of the original matrix may fail to materialize. In order to make the factorization numerically stable, the LU decomposition is always computed using partial pivoting in practice, which decomposes the row permuted matrix PA instead of the original matrix A. P is a square binary matrix that has exactly one entry of 1 in each row and each column, and 0s elsewhere. Then, we can solve the inverse matrix A^{-1} by computing $U^{-1}L^{-1}P$. Algorithm

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

Algorithm 1 : LU decomposition on a single node

Input:

$A = [a_{ij}]_{1 \leqslant i,j \leqslant n} \rightarrow$ The input matrix

Output:

$L \rightarrow L$

$U \rightarrow U$

$P \rightarrow$ The pivot matrix

```
1:  function LUDecompose(A)
2:      n = rows(A)
3:      for k = 1 to n do
4:          (j, k) = argmax(|A_{k,k}|, |A_{k+1,k}|, ..., |A_{n,k}|)
5:          add j to P
6:          swap i-th row with j-th row
7:          for i = k + 1 to n do
8:              A_{i,k} = A_{i,k} / A_{k,k}
9:              for j = k + 1 to n do
10:                 A_{i,j} = A_{i,j} − A_{i,k} A_{k,j}
11:             end for
12:         end for
13:     end for
14:     return (A,P)  /*(L,U,P)*/
15: end function
```

1 shows the pseudo-code of the in-place LU decomposition by the partial pivoting method, which is only workable for a small matrix that can be loaded into the memory of a single server. Return values are matrix A and matrix P. The upper triangular portion of A resembles U whereas the lower one resembles L.

B. Basic Block-recursive Matrix Inversion Algorithm

We develop a block-recursive algorithm based on LU decomposition to compute the inverse of a large-scale matrix that cannot be loaded into the memory of a single server. To illustrate the basic idea of the proposed algorithm, we take a matrix with 16 **blocks**, which is shown as the first matrix composed by $B_{11} - B_{44}$ in Figure 1, as an example to show the overall process. Each **block** is a square matrix of dimension $b \times b$. The order of the block, b, is around 10^3 or less, and thus the block is small enough to fit in the memory of a computing server and be decomposed efficiently. Matrix M denotes the input matrix of the algorithm, and $M^{(i)}$ denotes the input matrix of the *i*-th step. The steps to invert the sample matrix are as follows:

Step 1: Take the original matrix A as the input of computing $M^{(1)}$. If the input matrix $M^{(1)}$ is not small enough, e.g., cannot be decomposed on a single server very quickly, $M^{(1)}$ is partitioned into 4 small sub-matrices. The first sub-matrix, the matrix composed by $B_{11}, B_{12}, B_{21}, and B_{22}$ at the top-left corner, is selected as the new input matrix $M^{(2)}$.

Step 2: The input matrix $M^{(2)}$ is examined again to check if it is small enough. If not, $M^{(2)}$ is recursively partitioned into 4 small sub-matrices. The sub-matrix at the top-left corner is selected as the new input matrix $M^{(3)}$.

Step 3: Now $M^{(3)}$ is small enough, i.e., only the sub-matrix B_{11} is left in this example. It could be efficiently decomposed on a single server. Unlike the traditional LU decomposition algorithm, we calculate the L^{-1}, U^{-1} and P instead of L, U and P for further processes. We will explain how this approach can improve the performance in following sections.

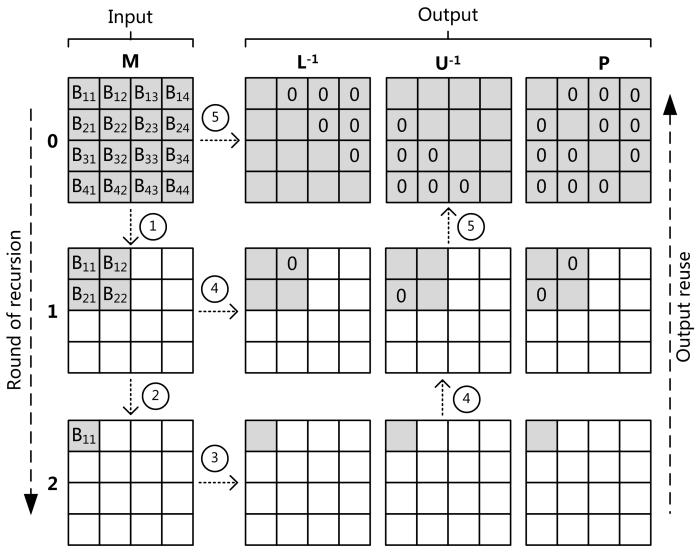

Fig. 1. Process of the proposed algorithm

Step 4: The results of L^{-1}, U^{-1} and P calculated in the function in the second round of recursion are returned to the caller in the first round of recursion. They are used together with the input matrix $M^{(2)}$ to solve the L^{-1}, U^{-1} and P in the first round of recursion.

Step 5: The results of L^{-1}, U^{-1} and P calculated in the first round of recursion are returned to the caller, and are used together with the original input matrix $M^{(1)}$ to compute the final L^{-1}, U^{-1} and P.

For a single block, L^{-1}, U^{-1} and P can be easily calculated by Algorithm 1 on a single server. So, the key point of the proposed algorithm to achieve high performance is how to calculate L^{-1}, U^{-1} and P when the input matrix is too big to be computed on a single server. We introduce a block-based approach to decompose the matrices M, L, U and P such that $PM = LU$. Without loss of generality, we assume the order of the square matrix M is $2^k b$, where k is a natural number and b is the order of the block matrix that can be decomposed on a single server. We will describe how to handle the matrices that do not meet this criterion at the end of this section. Let the matrices M, L, U and P be partitioned as blocks with equal size:

$$\begin{pmatrix} P_1 & O \\ O & P_2 \end{pmatrix} \begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix} = \begin{pmatrix} L_1 & O \\ L_2 & L_3 \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ O & U_3 \end{pmatrix}.$$

Performing the matrix multiplication in the above equation results in

$$\begin{pmatrix} P_1 M_1 & P_1 M_2 \\ P_2 M_3 & P_2 M_4 \end{pmatrix} = \begin{pmatrix} L_1 U_1 & L_1 U_2 \\ L_2 U_1 & L_2 U_2 + L_3 U_3 \end{pmatrix}.$$

This leads us to the following equations:

$$L_1 U_1 = P_1 M_1 \tag{1}$$

$$L_1 U_2 = P_1 M_2 \tag{2}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

Algorithm 2 : Basic Block-based LU Decomposition

Input:
$M = [m_{ij}]_{1 \leqslant i,j \leqslant n}$ → The input matrix
Output:
L^{-1} → Inverse of L
U^{-1} → Inverse of U
P → The pivot matrix

1: **function** $BlockLUDecompose(M)$
2: **if** M is small enough **then**
3: $(L, U, P) = LUDecompose(M)$
4: $L^{-1} = Inverse(L)$
5: $U^{-1} = Inverse(U)$
6: **else**
7: $\begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix} = M$
8: $(L_1^{-1}, U_1^{-1}, P_1) = BlockLUDecompose(M_1)$
9: $U_2 = L_1^{-1}(P_1 M_2)$
10: $\tilde{L_2} = M_3 U_1^{-1}$
11: $\tilde{M} = M_4 - \tilde{L_2} U_2$
12: $(L_3^{-1}, U_3^{-1}, P_2) = BlockLUDecompose(\tilde{M})$
13: $L_2 = P_2 \tilde{L_2}$
14: $L^{-1} = \begin{pmatrix} L_1^{-1} & O \\ -L_3^{-1} L_2 L_1^{-1} & L_3^{-1} \end{pmatrix}$
15: $U^{-1} = \begin{pmatrix} U_1^{-1} & -U_1^{-1} U_2 U_3^{-1} \\ O & U_3^{-1} \end{pmatrix}$
16: $P = \begin{pmatrix} P_1 & O \\ O & P_2 \end{pmatrix}$
17: **end if**
18: **return** (L^{-1}, U^{-1}, P)
19: **end function**

$$L_2 U_1 = P_2 M_3 \tag{3}$$

$$L_2 U_2 + L_3 U_3 = P_2 M_4. \tag{4}$$

If M_1 is small enough, i.e., it can be efficiently calculated on a single server, we can decompose M_1 to get L_1, U_1 and P_1 from Eq. (1) by Algorithm 1:

$$(L_1, U_1, P_1) = LUDecompose(M_1). \tag{5}$$

We can get L_1^{-1} by inverting the small matrix L_1. From Eq. (2), we have

$$U_2 = L_1^{-1} P_1 M_2. \tag{6}$$

If we define

$$\tilde{L_2} U_1 = M_3, \tag{7}$$

we can get

$$\tilde{L_2} = M_3 U_1^{-1}, \tag{8}$$

where U_1^{-1} is the inverse of small matrix U_1. Considering both Eqs. (3) and (8), we have

$$L_2 = P_2 \tilde{L_2}. \tag{9}$$

Algorithm 3 : Basic Block-recursive Matrix Inversion

Input:
$A = [a_{ij}]_{1 \leqslant i,j \leqslant n}$ → The input matrix
Output:
A^{-1} → Inverse Matrix of A

1: **function** $BlockInverse(A)$
2: (L^{-1}, U^{-1}, P) = $BlockLUDecompose(A)$
3: A^{-1}=$U^{-1} L^{-1} P$
4: **return** (A^{-1})
5: **end function**

Then, by substituting Eq. (9) into Eq. (4), we get

$$L_3 U_3 = P_2 M_4 - L_2 U_2 = P_2 (M_4 - \tilde{L_2} U_2).$$

Let $\tilde{M} = M_4 - \tilde{L_2} U_2$. If $\tilde{M}$ is small enough, we can get L_3, U_3 and P_2 by decomposing $\tilde{M}$:

$$(L_3, U_3, P_2) = LUDecompose(\tilde{M}). \tag{10}$$

After obtaining P_2, we can get L_2 by substituting P_2 and Eq. (8) into Eq. (9):

$$L_2 = P_2 \tilde{L_2} = P_2 M_3 U_1^{-1}. \tag{11}$$

Thus far, we get all L_1, L_2, L_3, U_1, U_2, U_3, P_1 and P_2 from Eqs. (5), (6), (10) and (11). So, we finally have:

$$L^{-1} = \begin{pmatrix} L_1^{-1} & O \\ -L_3^{-1} L_2 L_1^{-1} & L_3^{-1} \end{pmatrix} \tag{12}$$

$$U^{-1} = \begin{pmatrix} U_1^{-1} & -U_1^{-1} U_2 U_3^{-1} \\ O & U_3^{-1} \end{pmatrix} \tag{13}$$

$$P = \begin{pmatrix} P_1 & O \\ O & P_2 \end{pmatrix}.$$

M_1 and $\tilde{M}$ are assumed to be small enough to be decomposed on a single server according to Eqs. (5) and (10). If any of them is too big to be solved on a single server, we can keep recursively partitioning it into smaller sub-matrices until the sub-matrix is small enough. Algorithm 2 illustrates the basic block-recursive LU decomposition algorithm. Based on Algorithm 2, we have the basic block-recursive matrix inversion algorithm as described in Algorithm 3.

Note that we assume the order of the square matrix M and n is $2^k b$, where k is a natural number. If n does not equal to $2^k b$, we can perform the LU decomposition below. We define the block with dimensions $b \times b$ as the basic block. For each round of the recursion, the input matrix M is partitioned into $N \times N$ basic blocks, where $N = \lceil n/b \rceil$. The order of the blocks in the last row and column is $n - b \times (N-1)$. The order of other blocks is b. If $N = 2^k$, where k is a natural number, the matrix M can be decomposed directly by using the proposed algorithm. Otherwise, let $k = \lceil log_2 N \rceil$. In line 7 of Algorithm 2, we can set the size of M_1 as $2^{k-1} \times 2^{k-1}$ blocks. Then, we get the size of M_2, M_3 and M_4 as $2^{k-1} \times (N - 2^{k-1})$, $(N - 2^{k-1}) \times 2^{k-1}$ and $(N - 2^{k-1}) \times (N - 2^{k-1})$ blocks, respectively. Finally, the $BlockLUDecompose()$ function can be executed recursively on M_1 and $\tilde{M}$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

Algorithm 4 : Optimized Block-based LU Decomposition v1

Input:

$M = [m_{ij}]_{1 \leqslant i,j \leqslant n}$ → The input matrix

Output:

L^{-1} → Inverse of L

U^{-1} → Inverse of U

P → The pivot matrix

1: **function** $BlockLUDecompose(M)$
2: **if** M is smal **then**
3: $(L, U, P) = LUDecompose(M)$
4: $L^{-1} = Inverse(L)$
5: $U^{-1} = Inverse(U)$
6: **else**
7: $\begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix} = M$
8: $(L_1^{-1}, U_1^{-1}, P_1) = BlockLUDecompose(M_1)$
9: $T = U_1^{-1}L_1^{-1}$
10: $\tilde{S} = M_3T$
11: $\tilde{M} = M_4 - \tilde{S}(P_1M_2)$
12: $(L_3^{-1}, U_3^{-1}, P_2) = BlockLUDecompose(\tilde{M})$
13: $S = P_2\tilde{S}$
14: $L^{-1} = \begin{pmatrix} L_1^{-1} & O \\ -L_3^{-1}S & L_3^{-1} \end{pmatrix}$
15: $U^{-1} = \begin{pmatrix} U_1^{-1} & -T(P_1M_2)U_3^{-1} \\ O & U_3^{-1} \end{pmatrix}$
16: $P = \begin{pmatrix} P_1 & O \\ O & P_2 \end{pmatrix}$
17: **end if**
18: **return** (L^{-1}, U^{-1}, P)
19: **end function**

Algorithm 5 : Optimized Block-based LU Decomposition v2

Input:

$M = [m_{ij}]_{1 \leqslant i,j \leqslant n}$ → The input matrix

Output:

$LUofM$ → intermediate matrix of LU

1: **function** $BlockLUDecompose(M)$ ▷ *actual* L^{-1}, U^{-1}
2: $\begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix} = M$
3: **if** $M_1$is small **then**
4: $(L_1, U_1, P_1) = LU(M_1)$
5: $L_1^{-1} = Inverse(L_1)$
6: $U_1^{-1} = Inverse(U_1)$
7: $T = U_1^{-1}L_1^{-1}$
8: $\tilde{S} = M_3T$
9: $\tilde{M} = M_4 - \tilde{S}(P_1M_2)$
10: $(L_3, U_3, P_2) = LUDecompose(\tilde{M})$
11: $L_3^{-1} = Inverse(L_3)$
12: $U_3^{-1} = Inverse(U_3)$
13: **else**
14: $LUofM_1 = BlockLUDecompose(M_1)$
15: $(L_1^{-1}, U_1^{-1}, P_1) = getLU(LUofM_1)$
16: $T = U_1^{-1}L_1^{-1}$
17: $\tilde{S} = M_3T$
18: $\tilde{M} = M_4 - \tilde{S}(P_1M_2)$
19: $LUof\tilde{M} = BlockLUDecompose(\tilde{M})$
20: $(L_3^{-1}, U_3^{-1}, P_2) = getLU(LUof\tilde{M})$
21: **end if**
22: $LUofM = (L_1^{-1}, \tilde{S}, L_3^{-1}, U_1^{-1}, T, P_1M_2, U_3^{-1}, P_1, P_2)$
23: **return** $LUofM$
24: **end function**

C. Algorithm Optimization

Note that there are a number of matrix multiplications in Algorithm 2. If we can eliminate some of these multiplications, we will save tremendous computation resources and time. From Eqs. (2), (3), (12) and (13), we have:

$$L^{-1} = \begin{pmatrix} L_1^{-1} & O \\ -L_3^{-1}P_2M_3U_1^{-1}L_1^{-1} & L_3^{-1} \end{pmatrix}$$

$$U^{-1} = \begin{pmatrix} U_1^{-1} & -U_1^{-1}L_1^{-1}P_1M_2U_3^{-1} \\ O & U_3^{-1} \end{pmatrix}.$$

In the above equations, there is a duplicate multiplication $U_1^{-1}L_1^{-1}$, which is performed in each round of recursion. We introduce two new matrices, $T = U_1^{-1}L_1^{-1}$ and $\tilde{S} = M_3T$. Thus, Algorithm 2 is optimized and results in Algorithm 4. By comparing line 14 of Algorithm 2 with line 14 of Algorithm 4, we can see that the multiplication of three matrices $L_3^{-1}L_2L_1^{-1}$ have been optimized and have become the multiplication of two matrices $-L_3^{-1}S$.

The last step of Algorithm 3 takes a long time to compute $A^{-1} = U^{-1}L^{-1}P$, in which U^{-1}, L^{-1} and P have equal sizes with the large-scale input matrix A. From Algorithm 4, we have

$$M^{-1} = \begin{pmatrix} [M^{-1}]_1 & [M^{-1}]_2 \\ [M^{-1}]_3 & [M^{-1}]_4 \end{pmatrix},$$

where:

$$[M^{-1}]_1 = U_1^{-1}L_1^{-1}P_1 + U_1^{-1}L_1^{-1}P_1M_2U_3^{-1}L_3^{-1}P_2M_3U_1^{-1}L_1^{-1}P_1$$

$$[M^{-1}]_2 = -U_1^{-1}L_1^{-1}P_1M_2U_3^{-1}L_3^{-1}P_2$$

$$[M^{-1}]_3 = -U_3^{-1}L_3^{-1}P_2M_3U_1^{-1}L_1^{-1}P_1$$

$$[M^{-1}]_4 = -U_3^{-1}L_3^{-1}P_2.$$

There are a number of duplicate matrix multiplications in the four equations above, such as $U_1^{-1}L_1^{-1}$, $U_3^{-1}L_3^{-1}$ and P_2M_3. To eliminate these duplicate matrix multiplications, we define the following new variables:

$$\begin{aligned} T &= U_1^{-1}L_1^{-1} \\ X_1 &= TP_1 \\ X_2 &= U_3^{-1}L_3^{-1} \\ Y_1 &= X_1M_2 \\ Y_2 &= X_2M_3. \end{aligned}$$

Then, we have:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

Algorithm 6 : Compose LU

Input:
$LUofM \rightarrow$ intermediate matrix of LU
Output:
$L^{-1} \rightarrow$ Inverse of L
$U^{-1} \rightarrow$ Inverse of U
$P \rightarrow$ The pivot matrix

function $getLU(LUofM)$
  $(L_1^{-1}, \tilde{S}, L_3^{-1}, U_1^{-1}, T, P_1M_2, U_3^{-1}, P_1, P_2) = LUofM$
  $S = P_2\tilde{S}$
  $L^{-1} = \begin{pmatrix} L_1^{-1} & O \\ -L_3^{-1}S & L_3^{-1} \end{pmatrix}$
  $U^{-1} = \begin{pmatrix} U_1^{-1} & -T(P_1M_2)U_3^{-1} \\ O & U_3^{-1} \end{pmatrix}$
  $P = \begin{pmatrix} P_1 & O \\ O & P_2 \end{pmatrix}$
  return (L^{-1}, U^{-1}, P)
end function

$$[M^{-1}]_4 = X_2$$
$$[M^{-1}]_3 = -Y_2X_1$$
$$[M^{-1}]_2 = -Y_1X_2$$
$$[M^{-1}]_1 = X_1 - Y_1[M^{-1}]_3.$$

Based on the above equations, Algorithm 4 can be optimized and results in Algorithm 5. The pseudo-code of function $getLU$ in Algorithm 5 is shown in Algorithm 6. Finally, we have the optimized block-recursive matrix inversion algorithm as described in Algorithm 7, which achieves high-performance by avoiding the large-scale matrix multiplication $A^{-1} = U^{-1}L^{-1}P$.

IV. Key Points of Implementation on Spark

A. The Reason for Choosing Spark

The growing success of new distributed computing technologies like MapReduce [5] has made it possible to achieve large-scale matrix operations on clusters composed of commodity servers [7] [20]. However, the nature of MapReduce makes it inefficient for iterative computations. To address this issue, a number of alternative technologies have been proposed. Spark [6] is such a parallel computing platform, which supports efficient execution of iterative algorithms on a distributed memory abstraction named Resilient Distributed Datasets (RDDs). Choosing Spark to implement the proposed algorithm has several advantages. First, our algorithm benefits from the scalability and fault-tolerance capabilities of Spark, which provides low-overhead fault-tolerance to release the programmers from check-pointing and rollbacks during the long process of data computing. Second, the performance of our algorithm can be improved by the persistence feature of RDDs by avoiding to repeatedly save and load the intermediate data to and from hard-disks. Finally, as compared to the simple Map and Reduce programming model in MapReduce, the rich and flexible APIs of Spark provide us a relaxed space to control and optimize our algorithm in a fine-grained manner.

Algorithm 7 : Optimized Block-recursive Matrix Inversion

Input:
$A = [a_{ij}]_{1 \leqslant i,j \leqslant n} \rightarrow$ The input matrix
Output:
$A^{-1} \rightarrow$ Inverse of A

function $BlockInverse(A)$
  if A is small **then**
    return A^{-1}
  else
    $\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} = A$
    $LUofA = BlockLUDecompose(A)$
    $(L_1^{-1}, \tilde{S}, L_3^{-1}, U_1^{-1}, T, P_1A_2, U_3^{-1}, P_1, P_2) = LUofA$
    $X_1 = TP_1$
    $X_2 = U_3^{-1}L_3^{-1}$
    $Y_1 = X_1A_2$
    $Y_2 = X_2A_3$
    $[A^{-1}]_4 = X_2$
    $[A^{-1}]_3 = -Y_2X_1$
    $[A^{-1}]_2 = -Y_1X_2$
    $[A^{-1}]_1 = X_1 - Y_1[A^{-1}]_3$
    $A^{-1} = \begin{pmatrix} [A^{-1}]_1 & [A^{-1}]_2 \\ [A^{-1}]_3 & [A^{-1}]_4 \end{pmatrix}$
    return A^{-1}
  end if
end function

B. Data Structure Design on Spark

In each round of the recursion, the Spark-based implementation of the proposed algorithm requires distributing the huge input matrix and the intermediate matrices across servers in the distributed computing environment. Therefore, the representation of matrix data is a key point for effective implementation. In Spark, data are represented and managed as RDDs, which are a distributed memory abstraction to perform in-memory computations on clusters in a fault-tolerant manner. A RDD is a collection of elements that are partitioned across servers of a cluster. In this condition, we introduce two different distributed matrix representations, *IndexRowMatrix* and *BlockMatirx* to implement the proposed algorithm.

IndexRowMatrix is a row-oriented distributed matrix with meaningful indices. It is a suitable matrix data format for the input of our algorithm. For example, $A = [a_{ij}]_{1 \leqslant i,j \leqslant n}$ can be stored as a RDD $\{a_{11}, a_{12}, ..., a_{1n}, a_{21}, a_{22}, ..., a_{2n}, ..., a_{n1}, a_{n2}, ..., a_{nn}\}$ in row-major order. If a matrix is stored in this format, it requires two steps to access a partition of the matrix composed of a set of blocks, such as B_{11}, B_{12}, B_{21} and B_{22} in Figure 1. The first step is positioning the elements in these blocks by transforming the block indices into a number of element indices, such as $B_{11} = \{a_{11}, ..., a_{1b}, a_{21}, ..., a_{2b}, ..., a_{b1}, ..., a_{bb}\}$. The second step is reading the elements based on the transformed element indices. These elements may dispersively stored in a number of servers. It incurs a longer disk IO time and heavy overheads of network communications to compose a new RDD by these elements for further processing. Therefore,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

IndexRowMatrix is not suitable to store intermediate matrices. Towards this end, we introduce the block-oriented data structure *BlockMatirx*. It is a distributed matrix where a block is a $< Key, Value >$ pair $< BlockID, BlockValue >$. *BlockID* is the index of the block and *BlockValue* is the sub-matrix at the given $BlockID$ with size $b \times b$. For example, the block B_{11} in Figure 1 is organized as a RDD $< 11, \{a_{11}, ..., a_{1b}, a_{21}, ..., a_{2b}, ..., a_{b1}, ..., a_{bb}\} >$. A matrix is stored as a RDD that is composed by a number of blocks. In this condition, a block can be accessed directly by its index. Elements in a block are stored contiguous in memory, which dramatically reduces the time of disk IOs and overheads of network communications. Note that converting a distributed matrix to a different format, like from *IndexRowMatrix* to *BlockMatrix*, may require a global *shuffle* with network overhead. It is important to covert the format of matrix data as less as possible, especially for an iterative algorithm.

C. Solving L^{-1} and U^{-1} Instead of L and U

Previous LU decomposition based matrix inversion algorithms, such as [7], [18], [22], follow three steps: factorizing A as $PA = LU$, solving L^{-1} and U^{-1}, and obtaining $A^{-1} = U^{-1}L^{-1}P$. In contrast, we directly calculate L^{-1} and U^{-1} instead of L and U in each round of the recursion. In this section, we illustrate the necessity and benefit of this approach.

We have seen that if we calculate L and U at line 8 and 12 in Algorithm 2, it is required to solve two linear equations below to obtain two sub-matrices U_2 and L_2 with known matrix U_1, M_3, L_1, P_1 and M_2:

$$\begin{pmatrix} [\tilde{L_2}]^1 \\ ... \\ [\tilde{L_2}]^n \end{pmatrix} U_1 = \begin{pmatrix} [M_3]^1 \\ ... \\ [M_3]^n \end{pmatrix}$$

$$L_1 \begin{pmatrix} [U_2]_1 & ... & [U_2]_n \end{pmatrix} = \begin{pmatrix} [P_1M_2]_1 & ... & [P_1M_2]_n \end{pmatrix}.$$

where $[*]^i$ and $[*]_i$ are the i_{th} row and column of a matrix, respectively. If we follow the above process, we have to access the matrix by row manner and column manner, which will need to convert the BlockMatrix to IndexRowMatrix in each iteration. However, if we calculate L^{-1} and U^{-1} at line 8 and 12 in Algorithm 2, we can directly obtain $\tilde{L_2}$ and U_2 by computing $\tilde{L_2} = M_3U_1^{-1}$ and $U_2 = L_1^{-1}P_1M_2$ with matrix multiplication.

Another problem of traditional LU decomposition matrix inversion resides in the last step of solving A^{-1} by $A^{-1} = L^{-1}U^{-1}P$. If we only get L and U, it is required to invert large matrices to obtain L^{-1} and U^{-1}. For example, we can partition the large scale L as

$$L = \begin{pmatrix} L_1 & O \\ L_2 & L_3 \end{pmatrix}$$

to solve for L^{-1} by leveraging the lower triangular matrix characteristic of L:

$$L^{-1} = \begin{pmatrix} L_1^{-1} & O \\ -L_3^{-1}L_2L_1^{-1} & L_3^{-1} \end{pmatrix}.$$

For a large scale L, it requires a number of recursions to obtain the whole L^{-1}, which is a compute-intensive task. Obviously, we can avoid this step by directly obtain L^{-1} and U^{-1}.

D. High-performance Matrix Multiplication

Matrix multiplication operations are the major part of the computations of the proposed algorithm. Therefore, implementing a high-performance matrix multiplication is one of the key points for large-scale matrix inversion. Towards this end, we use a 1-D block-based approach to achieve efficient matrix multiplication with low network communications overhead. Suppose that we will solve $C = AB$, where A and B are both $n \times n$ matrices. We transform the matrices A and B from the original 2-D representation to an 1-D representation as:

$$\{A_{11}, A_{12}, ..., A_{1N}\}_{\times N \ times}, ..., \{A_{N1}, A_{N2}, ..., A_{NN}\}_{\times N \ times}$$
$$\{B_{11}, B_{21}, ..., B_{N1}, ..., B_{1N}, B_{2N}, ..., B_{NN}\}_{\times N \ times}$$

The 1-D representations of A and B are organized as two RDDs, $FlatA$ and $FlatB$, respectively. Each element in the RDD has a sequence number Seq. We perform a *join* operation between these two RDDs to get a new RDD, named $FlatAFlatBPair$. Each element in $FlatAFlatBPair$ is a key value pair $< Seq, (A_{i,k}, B_{k,j}) >$. Each map task multiplies two sub-matrices as its output value and *BlockID* $< Seq/N^2, \ Seq/N\%N >$ as its output key. Each reduce task sums the matrices to get the final result $C_{i,j} = \sum_{k=1}^{N} A_{i,k}B_{k,j}$. With customized partitioning functions, we can set the number of parallel tasks as N^2. In general, it is larger than the total number of cores; these parallel tasks could achieve better dynamic load balancing and speed up recovery when a server in the cluster fails or shows slow response.

E. Effective Permutation of Matrix

In the proposed algorithm, there are plenty of permutation operations on matrices to control the round-off error of matrix inversion. We concentrate on two points to achieve effective permutation of matrix: (1) an optimized data structure to manage the permutation matrix, and (2) effectively permuting rows and columns of a given matrix.

The permutation matrix P sized $n \times n$ has precisely one entry whose value is 1 in each row and each column, and its other entries are zero. Based on this characteristic, we utilize an array-based data structure $ArrayP = [arrayP_i]_{1 \leqslant i \leqslant n}$, where $arrayP_i = j$ if $p_{i,j} = 1$, to store P. This array-based data structure saves the space cost by using only n elements to store an $n \times n$ matrix, as well as facilitates the permutation operations below.

As mentioned before, the intermediate matrices to be computed in the proposed algorithm are stored in a block-based format. It requires three steps to perform the matrix permutation in the traditional way: (1) transforming the block-based format data to row-major/column-major order data, (2) updating the row/column indices for interchanging row/column, and (3) transforming the row-major/column-major order data to block-based format data. In the above

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

three steps, there are several rounds of *shuffle* operations, which will cause too many overheads of disk IOs and network communications for permuting a large-scale matrix. To improve performance, we introduce a method to perform the row permutation $B = PA$ directly on the block-based data structure. We compose a RDD RA to store blocks of the original matrix on row-major order. Each block is represented as a $< Key, Value >$ pair $< BlockID, BlockValue >$. Then, we apply a *flatMap* transformation on RA to obtain another RDD $\tilde{A}$. The first step in *flatMap* is splitting each block into a set of rows represented as $< Key, Value >$ pairs $< BlockID, (RowIndex, ValueOfRowPiece) >$, where $RowIndex$ is the index r of a row and $ValueOfRowPiece$ is the set of elements in the row. The second step is finding the index $\tilde{r}$ in $ArrayP$ such that $arrayP_{\tilde{r}} = r$, and updating the index of each row from r to $\tilde{r}$. After obtaining the new index of each row, we can get the $newBlockID$ of the row piece. The column index of $newBlockID$ is the same as previous $BlockID$ because the row permutation does not change the column index of each element. The row index $newBlockID$ can be easily obtained by $\tilde{r}/b$. Finally, we can obtain B by performing *groupByKey* transformation on $\tilde{A}$ to combine these row pieces to a new $BlockValue$. In the above process, there is only one *shuffle* in the *groupByKey* transformation.

From the equation of column permutation $B = AP = (P^T A^T)^T$, we can use the row permutation to achieve column permutation. The only thing we need to consider is obtaining P^T from P. From the array representation of P, $ArrayP = [arrayP_i]_{1 \leqslant i \leqslant n}$, we can simply get P^T by swapping the index i and the value j. After this, we can follow the above row permutation process to get the result of column permutation.

F. Reduction of Space Complexity

In general, most of traditional methods of parallelizing the LU factorization are based on a *blocked right-looking* approach, which is based on partitioning a large-scale matrix M that cannot be computed on a single server to blocks as:

$$\begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix} = \begin{pmatrix} L_1 & O \\ L_2 & L_3 \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ O & U_3 \end{pmatrix}.$$

To make it computable, the size of M_1 is set to be $b \times b$, where b is the size of a basic block that can be decomposed on a single server. This block partition approach leads to a recursive algorithm: (1) solving L_1 and U_1 by factorizing $M_1 = L_1 U_1$, (2) obtaining L_2 and U_2 by solving $M_3 = L_2 U_1$ and $M_2 = L_1 U_2$, respectively, and (3) defining a new matrix $\tilde{M}$ as $\tilde{M} = M_4 - L_2 U_2$ and recursively factorizing $\tilde{M}$ to obtain L_2 and U_2. For simplicity, we assume that the order of M is $2^k b$, where k is a natural number. In each iteration, we will obtain a new matrix $\tilde{M}$. If it is small enough, $\tilde{M}$ can be updated in the memory. However, in the process of inverting a large-scale matrix, $\tilde{M}$ is so large that has to be stored into the hard disk. It makes the size of $\tilde{M}$ to be an important factor of algorithm performance. The size of $\tilde{M}$ is

$$S = \sum_{r=1}^{N-1} r^2 b^2 = \frac{(N-1)N(2N-1)}{6} b^2,$$

where $N = 2^k$. It means that the space complexity of the *blocked right-looking* based method is $O(N^3)$.

In contrast, in our proposed Algorithm 2, the space cost of $\tilde{M}$ to invert a matrix M sized $2^k b \times 2^k b$ is

$$\begin{aligned} S(k) &= S(k-1) + (2^{k-1})^2 b^2 + S(k-2) + (2^{k-2})^2 b^2 \\ &\quad + ... + S(2) + (2^2)^2 b^2 + S(1) + 2^2 b^2 + b^2 \\ &= 2S(k-1) + 4^{k-1} b^2. \end{aligned}$$

From the above equation, we have

$$S(k) - \frac{1}{2} 4^k b^2 = 2(S(k-1) - \frac{1}{2} 4^{k-1} b^2).$$

In the above equation, $S(1)$ is the size of the intermediate data to invert matrix M sized $2b \times 2b$, which equals to b^2. If we let

$$T(k) = S(k) - \frac{1}{2} 4^k b^2,$$

we have

$$T(1) = S(1) - \frac{1}{2} 4b^2 = -b^2.$$

Then, we have:

$$T(k) = -2^{k-1} b^2$$

$$S(k) = (\frac{1}{2} 4^k b^2 - 2^{k-1}) b^2 = \frac{1}{2}(N^2 - N) b^2.$$

It indicates that the space complexity of our proposed algorithm is $O(N^2)$, which is much smaller than that of the blocked right-looking based method and thus helps improve the performance.

V. Experimental Evaluation

A. Experimental Environment

We implement our algorithm (later called SparkInverse) on Spark 1.3, the program of MapReduce-based algorithm based on the source code provided in [7] (later called MRInverse) on Hadoop 2.6, and a program (later called MPIInverse) based on ScaLAPACK library and MPICH2 MPI platform. All experiments were performed on a cluster composed of commodity servers connected by Gigabit switches, which are deployed in a data center with well designed architecture and optimized network [23]. Each server has 64GB memory, two 2.1GHz Intel Xeon CPUs with 12 physical cores, eight 7200 RPM hard disks, and one Gigabit ethernet card.

TABLE I
Input Matrices

Matrix	Order	CSV File Size	Binary File Size
M1	20480	7.6GB	3.2GB
M2	32768	20GB	8.1GB
M3	40960	31GB	14GB
M4	102400	189GB	-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2016.2546544, IEEE Access

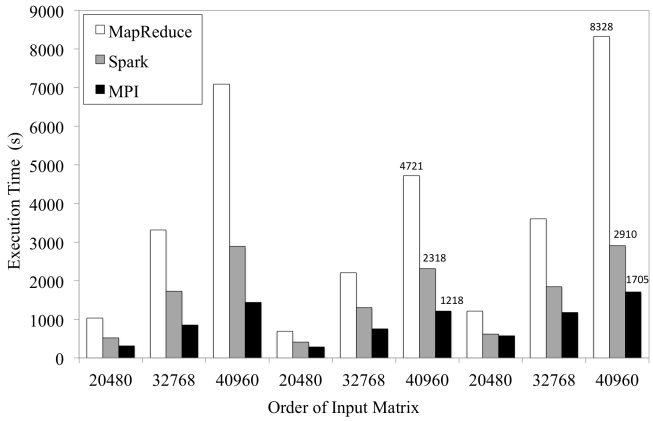

Fig. 2. Performance evaluation

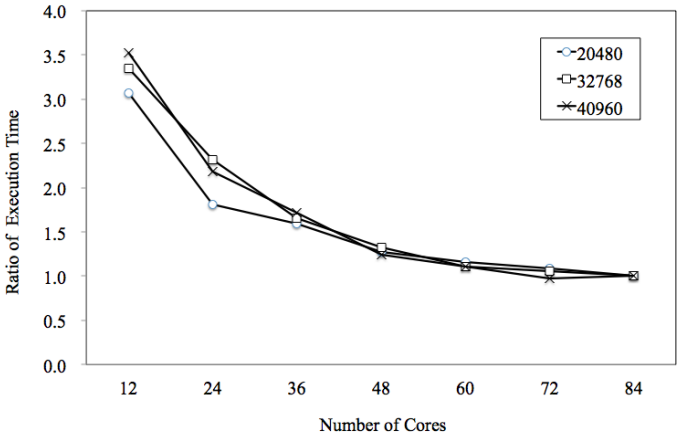

Fig. 3. The scalability of the proposed algorithm

To investigate the performance of the algorithms, we generated 4 large-scale matrices of uniformly distributed random numbers between 0 and 1. Details of the matrices are presented in Table I, which shows the order of each matrix, the size of the CSV file used as the input for SparkInverse, and the size of the binary file used as the input for the comparative MRInverse and MPIInverse. All files are stored in HDFS with the replication factor of 3.

B. Performance Comparison

Before evaluating the performance of algorithms, we verify the precision of MRInverse, SparkInverse and MPIInverse by computing $R = I_n - MM^{-1}$ for M_1, M_2 and M_3. The result shows that all elements in R are less than 10^{-7}, i.e., this validates the implementations with sufficient precision.

After validating the algorithms, we execute MRInverse, SparkInverse and MPIInverse on the cluster with different conditions to compare the performance. Figure 2 shows the results. The white, gray and dark bars are the results of execution time of MRInverse, SparkInverse, and MPIInverse, respectively. The first and second groups of three bars are the results of programs running on a small cluster with 4 servers and a large cluster with 7 servers, respectively. All servers in both small and large clusters are fully occupied by a single program to be evaluated during the testing. The results show that the SparkInverse outperforms the MRInverse for both small and large clusters. The SparkInverse performs even better when the input matrix becomes larger. This is attributed to the fact that MRInverse generates a large amount of intermediate data, and has to input from and output to hard disks during the computing. In the condition of fully occupied cluster environment, MPIInverse exhibits better performance as compared to MRInverse and SparkInverse. As compared to MPIInverse, our algorithm achieves comparable performance. It means that we can take advantages of fault-tolerance, convenient programming and enhanced software ecosystem by using the Spark framework with acceptable performance degradation.

In practice, a cluster composed of commodity servers cannot be occupied by a single program and usually experiences unpredictable failures. To evaluate this situation, we create a simulated environment by generating background network traffic in the large cluster and computing workload on one server. The third group of three bars in Figure 2 shows the results in this situation. The percentages of increased execution time on M_3, 76%, 40% and 26%, show the significant performance degradations of MRInverse and MPIInverse as compared to SparkInverse. The MPI program has to transfer a large amount of data on the network and cannot reschedule the computing tasks if any server crashes or shows slow responses. The MRInverse algorithm uses a static data partitioning approach to distribute tasks on servers, and cannot adapt the dynamic workloads on servers. In contrast, SparkInverse utilizes the capability of the speculative execution mechanism of the Spark framework that automatically handles the straggler of network transmitting and data computing.

For the matrix M_4 of order 102400, the MRInverse failed to invert it on the 7-node cluster with a 'Java heap space' error when it runs the last 'LU Inverse' job. Meanwhile, the SparkInverse spent 8 hours 30 minutes and 5 hours 42 minutes to solve the inversion of M_4 with precision of 10^{-5} on 4 and 7 servers, respectively. This demonstrates the advantage of our algorithm for inverting extra-large matrices on distributed computing environment.

C. Scalability and Bottleneck Analysis

To evaluate the scalability of the proposed algorithm, we run the Spark program to invert M_1, M_2 and M_3 with a varied number of servers. We configured the number of executors from 1 to 7, implying that the program can use a range of physical cores from 12 to 84. Figure 3 shows the result. The *x* axis is the number of cores and the *y* axis is the ratio of the execution time of each test to that of the test with the maximum number of cores. We can see that the execution time decreases when the available computing resource increases. Results show that the algorithm achieves good scalability when the number of cores is smaller than 48. However, we also note that there is a deviation from the expected line when the number of cores is larger than 48. To investigate the cause of this deviation, we broke down the execution time of the whole process of inverting the matrix. As we know, the computing

time, the disk IO time and the network IO time are the major parts of the total execution time of big data processing jobs. For Spark, the distributed memory data abstraction reduces the impact of the disk IO time on performance. So, we detailed the data size and the time of reading remote data on the shuffle process of computing the multiplication of two matrices of order 20480. The results are shown in Table II, in which the columns from left to right are the number of nodes, the accumulated CPU time, the time of pulling data from remote nodes in shuffle, the size of data read from remote nodes, the ratio of the remote data to the total shuffled data, and the ratio of time of remote pulling to the total shuffle time. Note that the accumulated CPU time is not the execution time. It is the accumulated CPU time of each task to get the shuffle data for the next reduce operation. We can see that the accumulated CPU time increases when the number of nodes increases because there are more data to be read from the remote nodes when more nodes are involved in the computing. In other words, the time of reading data from the remote nodes occupies the most part of the total shuffle time, and thus degrades the scalability of our algorithm when the number of nodes increases.

VI. Conclusion and Future Work

Large-scale matrix inversion is a fundamental operation for big data processing tasks, such as web-scale graph analysis and personalized recommendation. In this paper, we have presented a scalable and efficient matrix inversion algorithm for large-scale matrices and its implementation on Spark. We use a block-recursive method to breakdown the huge computation into a set of small partitions to compute the LU decomposition. Unlike traditional LU decomposition based matrix inversion algorithms, we solve L^{-1} and U^{-1} instead of L and U to reduce the computation and space complexity. The experimental evaluation has demonstrated that the proposed algorithm remarkably outperforms the state-of-the-art MapReduce-based implementation and exhibits the reliability and fault-tolerance capabilities as compared to the MPI program. The analysis of evaluation results on clusters with varied sizes has also showed that our algorithm achieves good scalability.

In terms of the future work, we consider two directions: (1) optimizing the proposed algorithm and implementation by decreasing the communications cost with new technologies like Tachyon [24], and (2) designing and implementing new linear algebra software libraries based on the algorithm implementation for large-scale matrix computation including singular value decomposition and solving eigenvectors and eigenvalues.

TABLE II
Impact of pull remote data

Node	Time		Remote Data	Shuffle Read Ratio	
	Accumulated CPU	Shuffle Read		Size	Time
1	4301s	0	0	0	0
2	8030s	5714s	49.6GB	49.6%	71.2%
3	7848s	5878s	65.6GB	65.6%	74.9%
4	7919s	6221s	72.4GB	72.4%	78.6%
7	10437s	8906s	85.1GB	85.1%	85.3%

References

[1] S. C. Althoen and R. Mclaughlin, "Gauss-jordan reduction: A brief history," *The American mathematical monthly*, vol. 94, no. 2, pp. 130–142, 1987.

[2] A. Burian, J. Takala, and M. Ylinen, "A fixed-point implementation of matrix inversion using cholesky decomposition," *IEEE 46th Midwest Symposium on Circuits and Systems*, vol. 3, pp. 1431-1434, 2003.

[3] W. H. Press, *Numerical Recipes 3rd edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[4] L. S. Blackford, J. Choi, *et al.*, *ScaLAPACK Users' Guide*. 1997.

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

[6] M. Zaharia, M. Chowdhury, *et al.*, "Spark: cluster computing with working sets," *2nd USENIX Conference on Hot Topics in Cloud Computing*, vol. 10, 2010.

[7] J. Xiang, H. Meng, and A. Aboulnaga, "Scalable matrix inversion using mapreduce," *23rd International Symposium on High-performance Parallel and Distributed Computing*, ACM, pp. 177-190, 2014.

[8] J. Liu, Y. Liang, C. Fang, and N. Ansari, "Spark-based Large-scale Matrix Inversion for Big Data Processing," *IEEE INFOCOM Workshop of Big Data Sciences, Technologies, and Applications (BDSTA), accepted*, 2016.

[9] J. Dongarra and P. Luszczek, "Linpack benchmark," *Encyclopedia of Parallel Computing*, Springer, pp. 1033-1036, 2011.

[10] E. Anderson, Z. Bai, *et al.*, "Lapack: A portable linear algebra library for high-performance computers," *ACM/IEEE conference on Supercomputing*, pp. 2-11, 1990.

[11] K. Lau, M. Kumar, and S. Venkatesh, "Parallel matrix inversion techniques," *IEEE Second International Conference on Algorithms & Architectures for Parallel Processing*, pp. 515-521, 1996.

[12] P. Bientinesi, B. Gunter, and R. A. Geijn, "Families of algorithms related to the inversion of a symmetric positive definite matrix," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 1, 2008.

[13] E. Agullo, H. Bouwmeester, *et al.*, "Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures," *High Performance Computing for Computational Science–VECPAR*, Springer, pp. 129-138, 2011.

[14] J. Dongarra, M. Faverge, *et al.*, "High performance matrix inversion based on lu factorization for multicore architectures," *ACM Workshop on Many Task Computing on Grids and Supercomputers*, pp. 33-42, 2011.

[15] K. Yang, Y. Li, and Y. Xia, "A parallel method for matrix inversion based on gauss-jordan algorithm," *Journal of Computational Information Systems*, vol. 9, no. 14, pp. 5561-5567, 2013.

[16] P. Ezzatti, E. S. *et al.*, "High performance matrix inversion on a multicore platform with several gpus," *19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. IEEE, pp. 87-93, 2011.

[17] G. Sharma, A. Agarwala, and B. Bhattacharya, "A fast parallel gauss jordan algorithm for matrix inversion using cuda," *Computers & Structures*, vol. 128, pp. 31–37, 2013.

[18] E. Caron and G. Utard, "Parallel out-of-core matrix inversion," *IEEE Parallel and Distributed Processing Symposium*, 2001.

[19] J. Liu, F. Liu, and N. Ansari, "Monitoring and analyzing big traffic data of a large-scale cellular network with Hadoop," *IEEE Network*, vol. 28, no. 4, pp. 32-39, 2014.

[20] S. Seo, E. J. Yoon, *et al.*, "Hama: An efficient matrix computation with the mapreduce framework," *IEEE Cloud Computing Technology and Science*, pp. 721-726, 2010.

[21] R. B. Zadeh, X. Meng, *et al.*, "linalg: Matrix computations in apache spark," *arXiv preprint arXiv:1509.02256*, 2015.

[22] L. Karlsson, "Computing explicit matrix inverses by recursion," MS Thesis, Umea University, Sweden, 2006.

[23] Y. Zhang and N. Ansari, "On architecture design, congestion notification, TCP incast and power consumption in data centers," IEEE Communications Surveys and Tutorials, vol. 15, no. 1, pp. 39-64, 2013.

[24] H. Li, A. Ghodsi, *et al.*, "Tachyon: Memory throughput i/o for cluster computing frameworks," *ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware*, vol. 18, 2013.