ICIC Express Letters
Part B: Applications
Volume **7**, Number **9**, September **2016**

 ISSN 2185-2766
pp. **1963–1968**

AN EFFICIENT MATRIX INVERSION ALGORITHM ON SPARK

Wei Lu, Xiangyu Zhao, Ergude Bao* and Weiwei Xing

School of Software Engineering
Beijing Jiaotong University
No. 3, Shangyuancun, Haidian District, Beijing 100044, P. R. China
*Corresponding author: baoe@bjtu.edu.cn

Received January 2016; accepted April 2016

Abstract. *Matrix inversion operation is a fundamental building block of many computational tasks in many fields. The operation has been successfully parallelized on Hadoop, while it is challenging to parallelize it on Spark, a more up-to-date parallel computation framework than Hadoop. In this paper, we address this problem by presenting an efficient and scalable parallel algorithm on Spark by splitting the input matrix into independent blocks for parallel LU decomposition and splitting the LU matrices into largely independent blocks for parallel triangular matrix inversion. Experimental results show our algorithm is faster than the previous Hadoop based algorithm with sufficient scalability.*
Keywords: Matrix inversion, Spark, Linear algebra, MapReduce

1. **Introduction.** Matrix inversion operation is a fundamental building block for many computational tasks in various fields such as image processing, wireless communication, and computer graphics. Basically, matrix inversion is used in these applications to solve linear equations: given an equation $Ax = B$, where A is a square matrix of order n, x and B are both vectors with n elements, x could be computed with $x = A^{-1}B$.

The inverse of a matrix can be computed using many methods, such as LU decomposition [1], QR decomposition [2], SVD decomposition [3] and Gauss-Jordan elimination [4]. The matrices processed are usually very large and become even larger in the current big data period. Therefore, how to parallelize the matrix inversion operation has become an important research topic.

To parallelize the matrix inversion operation, two things should be decided: (1) what matrix inversion method to parallelize, and (2) what programming framework to use. For the first thing, Xiang et al. [5] explained that using the LU decomposition method for matrix inversion is advantageous in parallelization over other methods. For the second thing, currently, Hadoop is the most widely used programming framework for parallel computation, while Spark is a much newer framework for this task. The major difference between Spark and Hadoop is that Spark provides in-memory primitives while Hadoop needs a large amount of disk operations, so Spark could be tremendously faster than Hadoop. Another difference between Spark and Hadoop is the fault tolerance policy. Spark's Resilient Distributed Dataset (RDD) model could rebuild lost data for fault tolerance without the need to replicate data as Hadoop. MLlib [6] and SystemML [7] are machine learning libraries for Spark and Hadoop, respectively. Although they had implemented various parallel matrix operations on Spark and Hadoop, none includes matrix inversion or LU decomposition. Sparkler [8] is an extension of Spark to support the decomposition of large size but low rank matrices. Therefore, it is important to parallelize matrix inversion with LU decomposition on Spark.

In this paper, we design the Spark based parallel matrix inversion algorithm by reducing the input matrix into a set of independent blocks which can be fit into memory and iteratively processing the blocks. In each iteration, the block is processed in parallel. Note

that if the primitives of Spark could be further enriched to allow more user control upon the data processing, our algorithm can be easily extended to support parallel computation among blocks, since the blocks are independent. Therefore, our algorithm and this paper would thus be a pioneer before the enrichment of the Spark primitives. More specifically, the algorithm has three steps: (1) decompose the input matrix into two triangular matrices first, (2) invert the two triangular matrices, and (3) compute the inverse of the input matrix with the two inverted triangular matrices. In the first two steps, independent blocks no larger than the memory size are obtained and processed one by one.

We organize the rest of this paper as follows. In Section 2, we present some basic matrix knowledge. In Section 3, we introduce the parallel matrix inversion algorithm on Spark. In Section 4, we will describe how to tune the performance on Spark. In Section 5, we present the experimental results. The conclusion is discussed in Section 6.

2. **Preliminaries.** A square matrix is the matrix which has the same number of rows and columns. The order of a square matrix is the number of rows or columns. The inverse of a square matrix A of order n is another matrix B such that $AB = BA = I$, where matrix I is the $n \times n$ identity matrix. A matrix A is invertible if and only if A is non-singular, that is, A is of full rank n. The element of matrix A in the ith row and jth column is denoted as A_{ij}. Below are the main idea about how to invert a matrix with LU decomposition.

The LU decomposition method [1] factors a matrix A into two triangular matrices by $A = LU$, where L is a lower triangular matrix, and U is an upper triangular matrix. After the LU decomposition, the inverse of A can be computed as $A^{-1} = U^{-1}L^{-1}$. Therefore, the problem of computing the inverse of A can be reduced to computing the inverses of the triangular matrices L and U.

The LU decomposition algorithm can be found in many references, so we will not give more introduction. Equation (1) shows how to invert a lower triangular matrix. Because $(A^T)^{-1} = (A^{-1})^T$, an upper matrix can be transposed to a lower triangular matrix to compute its inverse.

$$[A^{-1}]_{ij} = \begin{cases} 0 & \text{for } i < j \\ \frac{1}{[A]_{ii}} & \text{for } i = j \\ -\frac{1}{[A]_{ii}} \sum_{k=j}^{i-1} [A]_{ik} \left[A^{-1}\right]_{kj} & \text{for } i > j \end{cases} \tag{1}$$

3. **Parallel Matrix Inversion.** The algorithm to invert a matrix on Spark in parallel has three steps: (1) LU decomposes the input matrix into a lower triangular matrix and an upper triangular matrix, (2) inverts the two triangular matrices, and (3) computes the inverse of the input matrix with the two inverted triangular matrices. Each step can be parallelized, and are discussed below.

3.1. **Parallel LU decomposition.** As discussed above, we need to split the input matrix into blocks for LU decomposition, Equation (2) and subgraph (a) in Figure 1 illustrate the blocking strategy. The input matrix is split into 4 matrices: A_{11}, A_{12}, A_{21} and A_{22} of orders $r \times r$, $r \times (n-r)$, $(n-r) \times r$, and $(n-r) \times (n-r)$, respectively, where n is the order of the input matrix and $r << n$. Accordingly, each of the two output matrices are also composed of 3 blocks: L_{11}, L_{21} and L_{22} of orders $r \times r$, $(n-r) \times r$ and $(n-r) \times (n-r)$, respectively for the lower triangular matrix, and U_{11}, U_{12} and U_{22} of orders $r \times r$, $r \times (n-r)$ and $(n-r) \times (n-r)$, respectively for the upper triangular matrix. The relationships between the blocks in the input matrix and the blocks in the output matrix are given in Equation (3). Therefore, L_{11}, L_{21}, L_{22}, U_{11}, U_{12} and U_{22} can be computed as below.

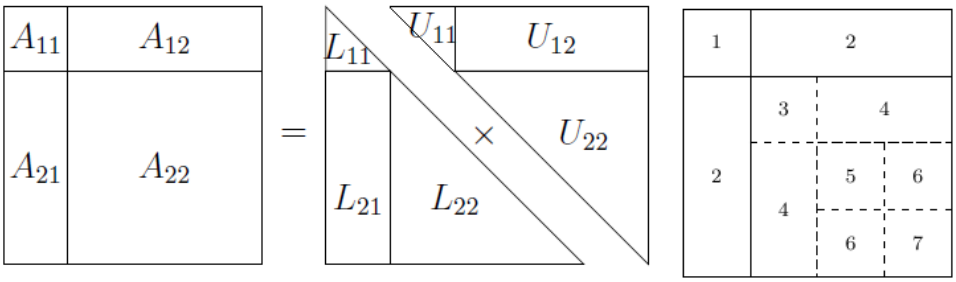
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} \tag{2}$$

$$\begin{cases} A_{11} = L_{11}U_{11} \\ A_{12} = L_{11}U_{12} \\ A_{21} = L_{21}U_{11} \\ A_{22} = L_{21}U_{12} + L_{22}U_{22} \end{cases} \tag{3}$$

- L_{11} and U_{11}: Since $r << n$, A_{11} can be loaded into memory and L_{11} and U_{11} can be computed directly with LU decomposition algorithm on single node.
- L_{21} and U_{12}: Given L_{11} and U_{11}, L_{21} and U_{12} can be computed with Equation (4). Since the rows of L_{21} are independent upon each other according to the equation, the computation of L_{21} can be parallelized on Spark. Similarly, since the columns of U_{12} are also independent upon each other, the computation of U_{12} can also be parallelized.

$$\begin{cases} [L_{21}]_{ij} = \frac{1}{[U_{11}]_{ii}} \left([A_{21}]_{ij} - \sum_{k=1}^{i-1} [L_{21}]_{ik} [U_{11}]_{kj} \right) \\ [U_{12}]_{ij} = \frac{1}{[L_{11}]_{ii}} \left([A_{12}]_{ij} - \sum_{k=1}^{i-1} [L_{11}]_{ik} [U_{12}]_{kj} \right) \end{cases} \tag{4}$$

- L_{22} and U_{22}: A_{22} is split recursively into 4 matrices A'_{11}, A'_{12}, A'_{21} and A'_{22} and decomposed as discussed above. When A^k_{22} in the kth recursion is small enough to fit into memory, L^k_{22} and U^k_{22} are computed from $A^k_{22} - L_{21}U_{12}$ as discussed in Section 2. Subgraph (b) in Figure 1 shows the steps about the recursion.



(a) Block LU method

(b) Process of block LU method

FIGURE 1. Block method for LU decomposition

Algorithm 1 gives the pseudo code for the parallel LU decomposition.

Algorithm 1 Parallel LU decomposition

```
1: function ParaLUDecom(A)
2:     if order of A < 2K then
3:         LUDecomposition(A)
4:     else
5:         split matrix into A11, A12, A21, A22
6:         L11 and U11 = LUDecomposition(A11)
7:         L21 = parallel process A21 and L11 with Equation (4)
8:         U12 = parallel process A12 and U11 with Equation (4)
9:         L22 and U22 = ParaLUDecom(A22 − L21U12)
10:        L = combine L11, L21 and L22
11:        U = combine U11, U12 and U22
12:    end if
13: end function
```

3.2. **Parallel triangular matrix inversion.** As discussed in Section 2, the inverse of the upper triangular matrix U can be computed with its transposed lower triangular matrix U^T, so here we only discuss how to compute the inverse of the lower triangular matrix L. Equation (5) and subgraph (a) in Figure 2 illustrate the blocking strategy for L: it is split into 3 matrices L_{11}, L_{21} and L_{22} of orders $r \times r$, $(n-r) \times r$ and $(n-r) \times (n-r)$, respectively. Its inverse is also a lower triangular matrix with three matrices B_{11}, B_{21} and B_{22} of orders $r \times r$, $(n-r) \times r$ and $(n-r) \times (n-r)$, respectively. Among these matrices, L_{11}, L_{22}, B_{11} and B_{22} are all lower triangular matrices. Equation (6) gives the relationships between these matrices, where I is the identity matrix of order n. Therefore, B_{11}, B_{21} and B_{22} can be computed as below.

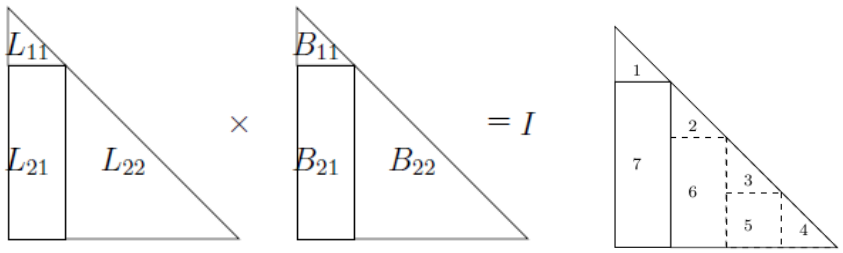
$$\begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} B_{11} & \\ B_{21} & B_{22} \end{bmatrix} = I \tag{5}$$

$$\begin{cases} L_{11}B_{11} = I \\ L_{21}B_{11} + L_{22}B_{21} = 0 \\ L_{22}B_{22} = I \end{cases} \tag{6}$$

- B_{11}: Since the order of L_{11} is small enough, B_{11} can be computed directly as discussed in Section 2.
- B_{21}: The computation of B_{21} is dependent on $L_{22}^{-1} = B_{22}$. Given B_{22}, as well as L_{21} and B_{11}, it can be computed with Equation (7). Since the computation of B_{21} requires only multiplications, it can be easily parallelized on Spark.

$$B_{21} = B_{22}(-L_{21}B_{11}) \tag{7}$$

- B_{22}: Since the order of L_{22} is large, it has to be computed recursively similar to Section 3. L_{22} is recursively split into 3 matrices L'_{11}, L'_{21} and L'_{22}. When L^k_{22} in the k^{th} recursion is small enough to fit into memory, B^k_{11} and B^k_{21} can be computed as discussed above. In this way, B^{k-1}_{22}, B^{k-2}_{22}, ... and B_{22} can be computed one by one. Subgraph (b) in Figure 2 shows the steps about the recursion.



(a) Block triangular matrix inversion

(b) Process to invert triangular matrix

FIGURE 2. Block method for triangular matrix inversion.

Algorithm 2 gives the pseudo code for the parallel lower triangular matrix inversion.

4. **Performance Tuning.** We do the following optimizations on Spark to tune the performance of our algorithm in its implementation stage.

- *Keep intermediate data in memory.* As discussed in Section 3, the computation of L_{21}, U_{12} and B_{21} require L_{11}, U_{11} and B_{11}, respectively, so we keep them in memory rather than writing to disk for further computation.
- *Serialize data.* After the blocks of the input matrix are all computed, we need to shuffle and combine them. In order to reduce the network load for this operation, we serialize the data to reduce the data size. Though it takes some time for data serialization and deserialization, the total computation time can be reduced. The

framework kryo [10] is used for serialization rather than Spark's built-in serializer due to its better performance.

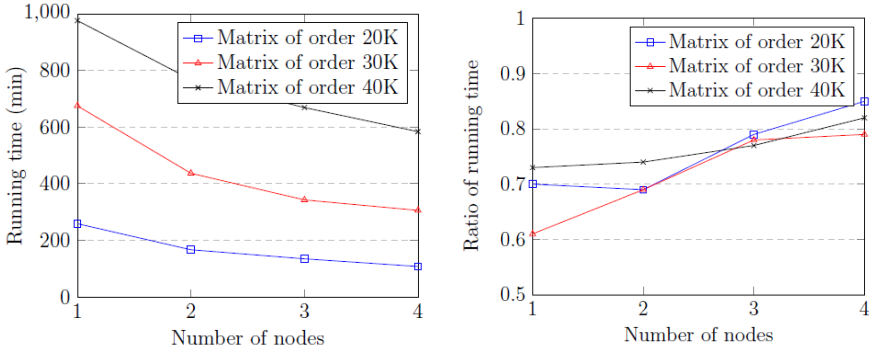
- *Tune parallelization scale.* The parallelization scale is a key factor affecting our algorithm's runtime. In order to fit the number of computing nodes and achieve a suitable parallelization scale, we set a relatively small number of partitions in the beginning and let Spark dynamically increase the number. Before we trigger the shuffle event, we redistribute the RDD to have fewer partitions.

Algorithm 2 Parallel triangular matrix inversion

```
function ParaInvertTrian(L)
    if order of L < 2000 then
        invert L with Equation (1)
    else
        split L into L11, L21, L22
        B11 = invert L11 with Equation (1)
        B22 = ParaInvertTrian(L22)
        B21 = parallel process B22, L21 and B11 with Equation (7)
        B = combine B11, B21 and B22
    end if
end function
```

5. **Experimental Results.** In the experiments, we investigated the scalability of our algorithm on Spark in cluster mode with multiple computing nodes, we also compared our algorithm to the matrix inversion algorithm on Hadoop as discussed in [5]. The computer we used had 4 nodes with 8 cores of 2GHz and 8GB memory in each node. The version of Spark was 1.3.1 with Scala 2.11.7. The version of Hadoop was 2.4.1 with Java 1.7. For the test with Spark, three matrices were used for orders 20K, 30K and 40K respectively.

5.1. **Scalability of cluster mode.** First test was Spark's cluster mode with the YARN [11] cluster manager. In this mode, Spark runs the tasks with multiple computing nodes from 1 to 4. YARN was chosen among a variety of cluster managers (e.g., Mesos [12] and Spark's cluster manager itself), because of its stronger support in task scheduling. Subgraph (a) in Figure 3 shows the test results: blue, red and black lines represent the runtime for matrices of orders 20K, 30K and 40K, respectively. The runtime of our algorithm generally decreases in linear with the increment of computing nodes, though it is not as linear as in local mode due to the overhead of data transfers between nodes. In general, our algorithm is scalable in cluster mode with various data sizes.



(a) Cluster mode (b) Comparison with [5]

FIGURE 3. Test results in cluster mode

5.2. **Comparison with existing method.** Since there has been no other matrix inversion algorithm implemented on Spark, we compared our algorithm to the matrix inversion algorithm on Hadoop [5]. Both were run in cluster mode with YARN on 1 to 4 nodes. For each test, the ratio of the runtime with our algorithm and with [5] was calculated and subgraph (b) in Figure 3 shows the results. In general, all the ratios for different tests are below 1 with the average 0.75, meaning that our algorithm is faster. The reason for our algorithm's fast speed comes from both our algorithm's parallelization and Spark's in-memory operations.

6. **Conclusion and Future Work.** In this paper we present a scalable matrix inversion algorithm on Spark. The algorithm is composed of three steps: block LU decomposition, block triangular matrix inversion and computing the inverse of the input matrix with the two inverted triangular matrices. Our experimental evaluation shows that our algorithm has good scalability and performance. One promising future work is to improve the LU decomposition step by adding parallelization among blocks and reducing the communication between each block.

Acknowledgment. This work is partially supported by National Natural Science Foundation of China (No. 61100143, 61272353, 61370128), Program for New Century Excellent Talents in University (NCET-13-0659), Beijing Higher Education Young Elite Teacher Project (YETP0583), and Fundamental Research Funds for the Central Universities (2014JBZ004, 2015RC045). The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers.

REFERENCES

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

[2] S. Lüpke, Lu-decomposition on a massively parallel transputer system, *PARLE'93 Parallel Architectures and Languages Europe*, pp.692-695, 1993.

[3] M. E. Wall, A. Rechtsteiner and L. M. Rocha, Singular value decomposition and principal component analysis, *A Practical Approach to Microarray Data Analysis*, pp.91-109, 2003.

[4] B. De Schutter and B. De Moor, The QR decomposition and the singular value decomposition in the symmetrized max-plus algebra, *SIAM Journal on Matrix Analysis and Applications*, vol.19, no.2, pp.378-406, 1998.

[5] J. Xiang, H. Meng and A. Aboulnaga, Scalable matrix inversion using mapreduce, *Proc. of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pp.177-190, 2014.

[6] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen et al., Mllib: Machine learning in apache spark, *arXiv Preprint arXiv:1505.06807*, 2015.

[7] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian and S. Vaithyanathan, Systemml: Declarative machine learning on mapreduce, *IEEE the 27th International Conference on Data Engineering*, pp.231-242, 2011.

[8] B. Li, S. Tata and Y. Sismanis, Sparkler: Supporting large-scale matrix factorization, *Proc. of the 16th International Conference on Extending Database Technology*, pp.625-636, 2013.

[9] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim and S. Maeng, HAMA: An efficient matrix computation with the mapreduce framework, *IEEE the 2nd International Conference on Cloud Computing Technology and Science*, pp.721-726, 2010.

[10] H. Karau, A. Konwinski, P. Wendell and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*, O'Reilly Media, Inc., 2015.

[11] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth et al., Apache Hadoop YARN: Yet another resource negotiator, *Proc. of the 4th Annual Symposium on Cloud Computing*, p.5, 2013.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker and I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center, *NSDI*, vol.11, pp.22-22, 2011.