SPIN: A Fast and Scalable Matrix Inversion Method in Apache Spark

Chandan Misra
Indian Institute of Technology Kharagpur
West Bengal, India
chandan.misra1@gmail.com

Swastik Haldar
Indian Institute of Technology Kharagpur
West Bengal, India
swastik.haldar@iitkgp.ac.in

Sourangshu Bhattacharya
Indian Institute of Technology Kharagpur
West Bengal, India
sourangshu@cse.iitkgp.ernet.in

Soumya K. Ghosh
Indian Institute of Technology Kharagpur
West Bengal, India
skg@iitkgp.ac.in

ABSTRACT

The growth of big data in domains such as Earth Sciences, Social Networks, Physical Sciences, etc. has lead to an immense need for efficient and scalable linear algebra operations, e.g. Matrix inversion. Existing methods for efficient and distributed matrix inversion using big data platforms rely on LU decomposition based block-recursive algorithms. However, these algorithms are complex and require a lot of side calculations, e.g. matrix multiplication, at various levels of recursion. In this paper, we propose a different scheme based on Strassen's matrix inversion algorithm (mentioned in Strassen's original paper in 1969), which uses far fewer operations at each level of recursion. We implement the proposed algorithm, and through extensive experimentation, show that it is more efficient than the state of the art methods. Furthermore, we provide a detailed theoretical analysis of the proposed algorithm, and derive theoretical running times which match closely with the empirically observed wall clock running times, thus explaining the U-shaped behaviour w.r.t. block-sizes.

CCS CONCEPTS

•**Computing methodologies → MapReduce algorithms;**

KEYWORDS

Linear Algebra, Matrix Inversion, Strassen's Algorithm, Apache Spark

1 INTRODUCTION

Dense matrix inversion is a basic procedure used by many applications in Data Science, Earth Science, Scientific Computing, etc, and has become an essential component of many such systems. It is an expensive operation, both in terms of computational and space complexity, and hence consumes a large fraction of resources in

many of the workloads. In the big data era, many of these applications have to work on huge matrices, possibly stored over multiple servers, and thus consuming huge amounts of computational resources for matrix inversion. Hence, designing efficient large scale distributed matrix inversion algorithms, is an important challenge.

Since its release in 2012, *Spark* [19] has been adopted as a dominant solution for scalable and fault-tolerant processing of huge datasets in many applications, e.g., machine learning [11], graph processing [7], climate science [12], social media analytics [2], etc. Spark has gained its popularity for its in-memory distributed data processing ability, which runs interactive and iterative applications faster than Hadoop MapReduce. It's close intergration with Scala / Java, and the flexible structure for RDDs allow distributed recursive algorithms to be implemented efficiently, without compromising on scalability and fault-tolerance. Hence, in this paper we focus on Spark for implementation of large scale distributed matrix inversion.

There are a variety of existing inversion algorithms, e.g. methods based on QR decomposition [13], LU decomposition [13], Cholesky decomposition [5], Gaussian Elimination [3], etc. Most of them require $O(n^3)$ time (where n denotes the order of the matrix), and main speed-ups in shared memory settings come from architecture specific optimizations (reviewed in section 2). Surprisingly, there are not many studies on distributed matrix inversion using big-data frameworks, where jobs could be distributed over machines with a diverse set of architectures. LU decomposition is the most widely used technique for distributed matrix inversion, possibly due to it's efficient block-recursive structure. Xiang et al. [17] proposed a Hadoop based implementation of inverting a matrix relying on computing the LU decomposition and discussed many Hadoop specific optimizations. Recently, Liu et al. [10] proposed several optimized block-recursive inversion algorithms on Spark based on LU decomposition. In the block recursive approach [10], the computation is broken down into subtasks that are computed as a pipeline of Spark tasks on a cluster. The costliest part of the computation is the matrix multiplication and the authors have given a couple of optimized algorithms to reduce the number of multiplications. However, in spite of being optimized, the implementation requires 9 $O(n^3)$ operations on the leaf node of the recursion tree, 12 multiplications at each recursion level of LU decomposition and an additional 7 multiplication after the LU decomposition to invert the matrix, which makes the implementation perform slower.

In this paper, we use a much simpler and less exploited algorithm, proposed by Strassen in his 1969 multiplication paper [16]. The algorithm follows similar block-recursion structure as LU decompostion, yet providing a simpler approach to matrix inversion. This approach involves no additional matrix multiplication at the leaf level of recursion, and requires only 6 multiplications at intermediate levels. We propose and implement a distributed matrix inversion algorithm based on Strassen's original serial inversion scheme. We also provide a detailed analysis of wall clock time for the proposed algorithm, thus revealing the 'U'-shaped behaviour with respect to block size. Experimentally, we show comprehensively, that the proposed approach is superior to the LU decomposition based approaches for all corresponding block sizes, and hence overall. We also demonstrate that our analysis of the proposed approach matches with the empirically observed wall clock time, and similar to ideal scaling behaviour. In summary:

(1) We propose and implement a novel approach (SPIN) to distributed matrix inversion, based on an algorithm proposed by Strassen [16].
(2) We provide a theoretical analysis of our proposed algorithm which matches closely with the empirically observed wall clock time.
(3) Through extensive experimentation, we show that the proposed algorithm is superior to the LU decomposition based approach.

2 RELATED WORK

The literature on parallel and distributed matrix inversion can be divided broadly into three categories: 1) HPC based approach, 2) GPU based approach and 3) Hadoop and Spark based approach. Here, we briefly review them.

2.1 HPC based approach

LINPACK, LAPACK and ScaLAPACK are some of the most robust linear algebra software packages that support matrix inversion. LINPACK was written in Fortran and used on shared-memory vector computers. It has been superseded by LAPACK which runs more efficiently on modern cache-based architectures. LAPACK has also been extended to run on distributed-memory MIMD parallel computers in ScaLAPACK package. However, these packages are based on architectures and frameworks which are not fault tolerant and MapReduce based matrix inversion are more scalable than ScaLAPACK as shown in [17]. Lau et al. [8] presented two algorithms for inverting sparse, symmetric and positive definite matrices on SIMD and MIMD respectively. The algorithm uses Gaussian elimination technique and the sparseness of the matrix to achieve higher performance. Bientinesi et al. [4] presented a parallel implementation of symmetric positive definite matrix on three architechtures — sequential processors, symmetric multiprocessors and distributed memory parallel computers using Cholesky factorization technique. Yang et al. [18] presented a parallel algorithm for matrix inversion based on Gauss-Jordan elimination with partial pivoting. It used efficient mechanism to reduce the communication overhead and also provides good scalability. Bailey et al. presented techniques to compute inverse of a matrix using

an algorithm suggested by Strassen in [16]. It uses Newton iteration method to increase its stability while preserving parallelism. Most of the above works are based on specialized matrices and not meant for general matrices. In this paper, we concentrate on any kind of square positive definite and invertible matrices which are distributed on large clusters which the above algorithms are not suitable for.

2.2 Multicore and GPU based approach

In order to fully exploit the multicore architecture, tile algorithms have been developed. Agullo et al. [1] developed such a tile algorithm to invert a symmetric positive definite matrix using Cholesky decomposition. Sharma et al. [15] presented a modified Gauss-Jordan algorithm for matrix inversion on CUDA based GPU platform and studied the performance metrics of the algorithm. Ezzatti et al. [6] presented several algorithms for computing matrix inverse based on Gauss-Jordan algorithm on hybrid platform consisting of multicore processors connected to several GPUs. 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.

2.3 MapReduce based approach

MadLINQ [14] offered a highly scalable, efficient and fault tolerant matrix computation system with a unified programming model which integrates with DryadLINQ, data parallel computing system. However, it does not mention any inversion algorithm explicitly. Xiang et al. [17] implemented first LU decomposition based matrix inversion in Hadoop MapReduce framework. However, it lacks typical Hadoop shortcomings like redundant data communication between map and reduce phases and inability to preserve distributed recursion structure. Liu et al. [10] provides the same LU based distributed inversion on Spark platform. It optimizes the algorithm by eliminating redundant matrix multiplications to achieve faster execution. Almost all the MapReduce based approaches relies on LU decomposition to invert a matrix. The reason is that it partitions the computation in a way suitable for MapReduce based systems. In this paper, we show that matrix inversion can be performed efficiently in a distributed environment like Spark by implementing Strassen's scheme which requires less number of multiplications than the earlier providing faster execution.

3 ALGORITHM DESIGN

In this section, we discuss the implementation of *SPIN* on Spark framework. First, we describe the original Strassen's inversion algorithm [16] for serial matrix inversion in section 3.1. Next, in section 3.2, we describe the *BlockMatrix* data structure from *MLLib* which is used in our algorithm to distribute the large input matrix into the distributed file system. Finally, section 3.3 describes the distributed inversion algorithm, and its implementation strategy using *Blockmatrix*.

3.1 Strassen's Algorithm for Matrix Inversion

Strassen's matrix inversion algorithm appeared in the same paper in which the well known Strassen's matrix multiplication was published. This algorithm can be described as follows. Let two matrices A and $C = A^{-1}$ be split into half-sized sub-matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{-1} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Then the result C can be calculated as shown in Algorithm 1. Intuitively, the steps involved in the algorithm are difficult to be performed in parallel. However, for input matrices which are too large to be fit into the memory on a single server, each such step is required to be processed distributively. These steps include breaking a matrix into four equal size sub-matrices, multiplication and subtraction of two matrices, multiplying a matrix to a scalar and arranging four half-sized sub-matrices into a full matrix. All these steps are done by splitting the matrix into blocks which act as execution unit of the spark job. A brief description of the block data structure is given below.

Algorithm 1: Strassen's Serial Inversion Algorithm

---

function Inverse();
Input : Matrix A (input matrix of size $n \times n$), int *threshold*
Output: Matrix C (invert of matrix A
begin
  if *n=threshold* **then**
    | invert A in any approach (e.g., LU, QR, SVD
    | decomposition);
  else
    | Compute $A_{11}, B_{11}, ..., A_{22}, B_{22}$ by computing $n = \frac{n}{2}$;
    | $I \leftarrow A_{11}^{-1}$
    | $II \leftarrow A_{21}.I$
    | $III \leftarrow I.A_{12}$
    | $IV \leftarrow A_{21}.III$
    | $V \leftarrow IV - A_{22}$
    | $VI \leftarrow V^{-1}$
    | $C_{12} \leftarrow III.VI$
    | $C_{21} \leftarrow VI.II$
    | $VII \leftarrow III.C_{21}$
    | $C_{11} \leftarrow I - VII$
    | $C_{22} \leftarrow -VI$
  end
  return C
end

3.2 Block Matrix Data Structure

In order to distribute the matrix in the HDFS (Hadoop Distributed File System), we create a distributed matrix called BlockMatrix, which is basically an RDD of MatrixBlocks spread in the cluster. Distributing the matrix as a collection of Blocks makes them easy to be processed in parallel and follow divide and conquer approach. *MatrixBlock* is a block of matrix represented as a tuple *((rowIndex, columnIndex), Matrix).* Here, *rowIndex* and *columnIndex* are the row and column index of a block of the matrix. *Matrix* refers to a one-dimensional
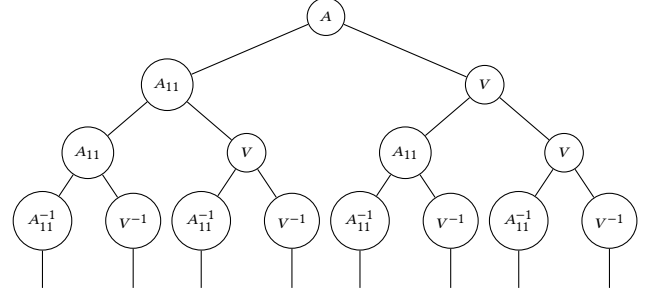

Figure 1: Recursion tree for Algorithm 2

array representing the elements of the matrix arranged in a column major fashion.

3.3 Distributed Block-recursive Matrix Inversion Algorithm

The distributed block-recursive algorithm can be visualized as Figure 1, where upper left sub-matrix is divided recursively until it can be inverted serially on a single machine. After the leaf node inversion, the inverted matrix is used to compute intermediate matrices, where each step is done distributively. Another recursive call is performed for matrix VI until leaf node is reached. Like A_{11}, it is also inverted on a single node when the leaf node is reached. The core inversion algorithm (described in Algorithm 2) takes a matrix (say A) represented as *BlockMatrix*, as input as shown in Figure 1. The core computation performed by the algorithm is based on six distributed methods, which are as follows:

- *breakMat*: Breaks a matrix into four equal sized sub-matrices
- *xy*: Returns one of the four sub-matrices after the breaking, according to the index specified by x and y.
- *multiply*: Multiplies two *BlockMatrix*
- *subtract*: Subtracts two *BlockMatrix*
- *scalarMul*: Multiples a scalar with a *BlockMatrix*
- *arrange*: Arranges four equal quarter *BlockMatrices* into a single full *BlockMatrix*.

Below we describe the methods in a little bit more details and also provide the algorithm for each.

breakMat method breaks a matrix into four sub-matrices but does not return four sub-matrices to the caller. It just prepare the input matrix to a form which help filtering each part easily. As described in Algorithm 3 it takes a *BlockMatrix* and returns a *PairRDD* of *tag* and *Block* using a *mapToPair* transformation. First, the *BlockMatrix* is converted into an RDD of *MatrixBlocks*. Then, each *MatrixBlock* of the RDD is mapped to tuple of *(tag, MatrixBlock)*, resulting a *pairRDD* of such tuples. Inside the *mapToPair* transformation, we carefully tag each *MatrixBlock* according to which quadrant it belongs to.

xy method is a generic method signature for four methods used for accessing one of the four sub-matrices of size 2^{n-1} from a matrix of size 2^n. Each method consists of two transformation — *filter* and *map*. *filter* takes the matrix as a *pairRDD* of *(tag, MatrixBlock)* tuple which was the output of *breakMat* method and filters the appropriate portion against the tag associated with the *MatrixBlock*.

Algorithm 2: Spark Algorithm for Strassen's Inversion Scheme

function Inverse();
begin

    Input : BlockMatrix A, int $size$, int $blockSize$

    Output : BlockMatrix $AInv$

    $size$ = Size of matrix A or B;

    $blockSize$ = Size of a single matrix block;

    $n = \frac{size}{blockSize}$;

    if $n = 1$ **then**

        $RDD < Block > invA \leftarrow A.toRDD()$

        Map();

        begin

            Input : Block block

            Output : Block block

            $block.matrix \leftarrow locInverse(block.matrix)$

            return $block$

        end

        $blockAInv \leftarrow invA.toBlockMatrix()$

        return $blockAInv$

    else

        $size \leftarrow size/2$

        $pairRDD \leftarrow breakMat(A, size)$

        $A11 \leftarrow \_11(pairRDD, blockSize)$

        $A12 \leftarrow \_12(pairRDD, blockSize)$

        $A21 \leftarrow \_21(pairRDD, blockSize)$

        $A22 \leftarrow \_22(pairRDD, blockSize)$

        $I \leftarrow Inverse(A11, size, blockSize)$

        $II \leftarrow multiply(A21, I)$

        $III \leftarrow multiply(I, A12)$

        $IV \leftarrow multiply(A21, III)$

        $V \leftarrow subtract(IV, A22)$

        $VI \leftarrow Inverse(V, size, blockSize)$

        $C12 \leftarrow multiply(III, VI)$

        $C21 \leftarrow multiply(VI, II)$

        $VII \leftarrow multiply(III, C21)$

        $C11 \leftarrow subtract(I, VII)$

        $C22 \leftarrow scalerMul(VI, -1, blockSize)$

        $C \leftarrow arrange(C11, C12, C21, C22, size, blockSize)$

        return C

    end

end

Algorithm 3: Spark Algorithm for breaking a *BlockMatrix*

function breakMat();
begin

    Input : BlockMatrix A, int size

    Output : PairRDD brokenRDD

    $ARDD \leftarrow A.toRDD$

    MapToPair();

    begin

        Input : block of ARDD

        Output : tuple of brokenRDD

        $ri \leftarrow block.rowIndex$

        $ci \leftarrow block.colIndex$

        if $ri/size = 0$ & $ci/size = 0$ **then**

            $tag \leftarrow$ "A11"

        end

        else if $ri/size = 0$ & $ci/size = 1$ **then**

            $tag \leftarrow$ "A12"

        end

        else if $ri/size = 1$ & $ci/size = 0$ **then**

            $tag \leftarrow$ "A21"

        end

        else

            $tag \leftarrow$ "A22"

        end

        $block.rowIndex \leftarrow ri\%size$

        $block.colIndex \leftarrow ci\%size$

        return $Tuple2(tag, block)$

    end

    return $brokenMat$

end

Algorithm 4: Spark Algorithm for multiplying a scalar to a distributed matrix

function xy();
begin

    Input : PairRDD $brokenRDD$

    Output : BlockMatrix xy

    filter();

    begin

        Input : PairRDD $brokenRDD$

        Output : PairRDD $filteredRDD$

        return $brokenRDD.tag =$ "A_{xy}"

    end

    map();

    begin

        Input : PairRDD $filteredRDD$

        Output : RDD rdd

        return $filteredRDD.block$

    end

    $xy \leftarrow rdd.toBlockMatrix()$

    return xy

end

Then it converts the *pairRDD* into *RDD* using the *map* transformation.

    multiply method multiplies two input sub-matrices and returns another sub-matrix of *BlockMatrix* type. Multiply method in our algorithm uses naive block matrix multiplication approach, which replicates the blocks of matrices and groups the blocks together to be multiplied in the same node. It uses co-group to reduce the communication cost.

    subtract method subtracts two *BlockMatrix* and returns the result as *BlockMatrix*.

scalarMul method (as described in Algorithm 5), takes a *Block-Matrix* and returns another *BlockMatrix* using a *map* transformation. The *map* takes blocks one by one and multiply each element of the block with the scalar.

Algorithm 5: Spark Algorithm for multiplying a scalar to a distributed matrix

---

function scalarMul();
begin
    Input : BlockMatrix A, double scalar, int blockSize
    Output : BlockMatrix productMat
    $ARDD \leftarrow A.toRDD()$
    Map();
    begin
        Input : block of ARDD
        Output : block of productRDD
        $product \leftarrow block.matrix.toDoubleMatrix$
        $block.matrix \leftarrow product.toMatrix$
        return *block*
    end
    $productMat \leftarrow product.toBlockMatrix()$
    return *productMat*
end

---

arrange method (as described in Algorithm 6), takes four sub-matrices of size 2^{n-1} which represents four co-ordinates of a full matrix of size 2^n and arranges them in later and returns it as *Block-Matrix*. It consists of four *map*s, each one for a separate *BlockMatrix*. Each *map* maps the block index to a different block index that provides the final position of the block in the result matrix.

4 PERFORMANCE ANALYSIS

In this section, we attempt to estimate the performances of the proposed approach, and state-of-the-art approach using LU decomposition for distributed matrix inversion. In this work, we are interested in the *wall clock running time* of the algorithms for varying number of nodes, matrix sizes and other algorithmic parameters e.g., partition / block sizes. This is because we are interested in the practical efficiency of our algorithm which includes not only the time spent by the processes in the CPU, but also the time taken while waiting for the CPU as well as data communication during shuffle. The wall clock time depends on three independently analyzed quantities: total *computational complexity* of the sub-tasks to be executed, total *communication complexity* between executors of different sub-tasks on each of the nodes, and *parallelization factor* of each of the sub-tasks or the total number of processor cores available.

Later, in section 5, we compare the theoretically derived estimates of *wall clock time* with empirically observed ones, for validation. We consider only square matrices of dimension 2^p for all the derivations. The key input and tunable parameters for the algorithms are:

- $n = 2^p$: number of rows or columns in matrix A
- b = number of splits for square matrix
- $2^q = \frac{n}{b}$ = block size in matrix A
- *cores* = Total number of physical cores in the cluster

Algorithm 6: Spark Algorithm for rearranging four sub-matrices into single matrix

---

function arrange();
begin
    Input : BlockMatrix C11, BlockMatrix C12, BlockMatrix
            C21, BlockMatrix C22, int size, int blockSize
    Output : BlockMatrix arranged
    $C11RDD \leftarrow C11.toRDD()$
    $C12RDD \leftarrow C12.toRDD()$
    $C21RDD \leftarrow C21.toRDD()$
    $C22RDD \leftarrow C22.toRDD()$
    Map();
    begin
        Input : block of C12RDD
        Output : block of C1
        $block.colIndex \leftarrow block.colIndex + size$
        return *block*
    end
    Map();
    begin
        Input : block of C21RDD
        Output : block of C2
        $block.rowIndex \leftarrow block.rowIndex + size$
        return *block*
    end
    Map();
    begin
        Input : block of C22
        Output : block of C3
        $block.rowIndex \leftarrow block.rowIndex + size$
        $block.colIndex \leftarrow block.colIndex + size$
        return *block*
    end
    $unionRDD \leftarrow C11RDD.union(C1.union(C2.union(C3)))$
    $C \leftarrow unionRDD.toBlockMatrix()$
    return *C*
end

- i = current processing level of algorithm in the recursion tree.
- m = total number of levels of the recursion tree.

Therefore,

- Total number of blocks in matrix A or $B = b^2$
- $b = 2^{p-q}$

LEMMA 4.1. *The proposed distributed block recursive strassen's matrix inversion algorithm or SPIN (presented in Algorithm 2) has a complexity in terms of wall clock execution time requirement, where n is the matrix dimension, b is the number of splits, and cores is the actual number of physical cores available in the cluster, as*

$$Cost_{SPIN} = \left(\frac{n^3}{b^2}\right) + \frac{10b^2 - 6b}{min\left[\frac{b^2}{4^i}, cores\right]} + \frac{(b-1) + \left(9b^2 + n^2(b+1)\right)}{b \times min\left[\frac{b^2}{4^{i+1}}, cores\right]}$$
$$+ \frac{n^2(b^2 n + b^2 - 2n)}{b^2 \times min\left[\frac{n^2}{4^{i+1}}, cores\right]} \tag{1}$$

Proof. Before going into details of the analysis, we give the performance analysis of the methods described in section 3.3. A summary of the independently analyzed quantities is given in Table 1.

There are two primary part of the algorithm — *if* part and *else* part. *If* part does the calculation of the leaf nodes of the leaf nodes of the recursion tree as shown in Figure 1, while *else* part does the computation for internal nodes. It is clearly seen from the figure that, at level i, there are 2^i nodes and the leaf level contains 2^{p-q} nodes.

There is only one transformation in *if* part which is *map*. It calculates the inverse of a matrix block in a single node using serial matrix inversion method. The size of each block is n/b and we need $\approx (n/b)^3$ time to perform each such method. Therefore, the computation cost to process all the leaf nodes is

$$Comp_{leafNode} = 2^{p-q} \times \left(\frac{n}{b}\right)^3 = \frac{n^3}{b^2} \tag{2}$$

In *SPIN*, leaf nodes processes one block on a single machine of the cluster. In spite of being small enough to be accommodated in a single node, we do not collect them in the master node for the communication cost. Instead, we do a *map* which takes the only block of the RDD, do the calculation and return the RDD again.

breakMat method takes a *BlockMatrix* and returns a *PairRDD* of *tag* and *Block* using a *mapToPair* transformation. If the method is executed for m levels, the computation cost of *breakMat* is

$$Comp_{breakMat} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^i}\right) = 2b(b-1) \tag{3}$$

Note that, i^{th} level contains 2^i nodes. Here each block is consumed in parallel giving parallelization factor as

$$PF_{breakMat} = min\left[\left(\frac{b^2}{4^i}\right), cores\right] \tag{4}$$

The total number of blocks processed in *filter* and *map* are $\left(\frac{b^2}{4^i}\right)$ and $\left(\frac{b^2}{4^{i+1}}\right)$ for i^{th} level respectively. Consequently, the parallelization factor of both of them are $min\left[\left(\frac{b^2}{4^i}\right), cores\right]$ and $min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right]$ respectively. Therefore, the computation cost for xy is

$$Comp_{xy} = \left[\frac{\sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^i}\right)}{min\left[\left(\frac{b^2}{4^i}\right), cores\right]} + \frac{\sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^{i+1}}\right)}{min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right]}\right]$$
$$= \left[\frac{8b^2 - 4b}{min\left[\left(\frac{b^2}{4^i}\right), cores\right]} + \frac{2b^2 - 2b}{min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right]}\right] \tag{5}$$

multiply method multiplies two *BlockMatrices*, the computation cost of which can be derived as

$$Comp_{multiply} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{n^3}{8^{i+1}}\right) = \frac{n^3(b^2 - 1)}{6b^2} \tag{6}$$

and the parallelization factor will be

$$PF_{multiply} = min\left[\frac{n^2}{4^{i+1}}, cores\right] \tag{7}$$

subtract method subtracts two *BlockMatrices* using a *map* transformation. There are two subtraction in each recursion level. Therefore,

$$Comp_{subtract} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{n^2}{4^{i+1}}\right) = \frac{n^2(b-1)}{2b} \tag{8}$$

and the parallelization factor will be

$$PF_{subtract} = min\left[\frac{n^2}{4^{i+1}}, cores\right] \tag{9}$$

scalarMul method (as described in Algorithm 5), takes a *BlockMatrix* and returns another *BlockMatrix* using a *map* transformation. The *map* takes blocks one by one and multiply each each element of the block with the scalar. Therefore, the computation cost of *scalarMul* is

$$Comp_{scalarMul} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^{i+1}}\right) = \frac{b}{2}(b-1) \tag{10}$$

Again, here each block is consumed in parallel giving parallelization factor as

$$PF_{scalarMul} = min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right] \tag{11}$$

arrange method (as described in Algorithm 6), takes four submatrices of size 2^{n-1} which represents four co-ordinates of a full matrix of size 2^n and arranges them in later and returns it as *BlockMatrix*. It consists of four *map*s, each one for a separate *BlockMatrix*. Each *map* maps the block index to a different block index that provides the final position of the block in the result matrix. The computation cost and parallelization factor for *map*s are same as *scalarMul*, which can be found in equation 10.

SPIN requires 4 xy method calls, 6 multiplications, and 2 subtractions for each recursion level. When summed up it will give equation 1.

□

Lemma 4.2. *The proposed distributed block recursive LU decomposition based matrix inversion algorithm or SPIN (presented in Algorithm 5, 6, and 7 in [10]) has a complexity in terms of wall clock execution time requirement, where n is the matrix dimension, b is the number of splits, and cores is the actual number of physical cores available in the cluster, as below*

Table 1: Summary of the cost analysis of *LU* and *SPIN*

Method	Computation Cost		Parallelization Factor	
	LU	SPIN	LU	SPIN
leafNode	$9 \times \frac{n^3}{b^2}$	$\frac{n^3}{b^2}$	—	—
breakMat	$\frac{2}{3}\left(b^2 - 3b + 2\right)$	$2b^2 - 2b$	$min\left[\frac{b^2}{4^i}, cores\right]$	$min\left[\frac{b^2}{4^i}, cores\right]$
xy (filter)	$\frac{2}{3}\left(b^2 - 3b + 2\right)$	$8b^2 - 4b$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$	$min\left[\frac{b^2}{4^i}, cores\right]$
xy (map)	$\frac{1}{6}\left(b^2 - 3b + 2\right)$	$2b^2 - 2b$	$min\left[\frac{b^2}{4^{i+2}}, cores\right]$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$
multiply (large)	$\frac{16n^3}{21b^3}(b^3 - 7b + 6)$	$\frac{n^3}{6b^2}(b^2 - 1)$	$min\left[\frac{n^2}{4^i}, cores\right]$	$min\left[\frac{n^2}{4^{i+1}}, cores\right]$
multiply Communication (large)	$\frac{8n^2(b^2-1)(8b^2-112)}{105b^2}$	$\frac{n^2(b^2-1)}{6b}$	$min\left[\frac{b^2}{4^i}, cores\right]$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$
multiply (small)	$\frac{8n^3}{42b^3}(b^3 - 7b + 6)$	—	$min\left[\frac{n^2}{4^{i+1}}, cores\right]$	—
multiply Communication (small)	$\frac{n^2(b^2-1)(8b^2-112)}{105b^2}$	—	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$	—
subtract	$\frac{2n^2}{3b^2}(b^2 - 3b + 2)$	$\frac{n^2}{2b}(b - 1)$	$min\left[\frac{n^2}{4^i}, cores\right]$	$min\left[\frac{n^2}{4^{i+1}}, cores\right]$
scalarMul	$\frac{4}{3}\left(b^2 - 3b + 2\right)$	$\frac{b}{2}(b - 1)$	$min\left[\frac{b^2}{4^i}, cores\right]$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$
arrange	—	$\frac{b}{2}(b - 1)$	—	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$
Additional Cost	$7 \times \left(\frac{n}{2}\right)^3$	—	$min\left[\frac{n^2}{4}, cores\right]$	—

$$Cost_{LU} = \frac{9n^3}{b^2} + \frac{(b-1)[210b^2(b-2) + 64n^2(b+1)(b^2-14)]}{105b^2 \times min\left[\frac{b^2}{4^i}, cores\right]}$$

$$+ \frac{(b-1)[70b^2(b-2) + 8n^2(b+1)(b^2-14)]}{105b^2 \times min\left[\frac{b^2}{4^{i+1}}, cores\right]}$$

$$+ \frac{(b-1)(b-2)}{105b^2 \times min\left[\frac{b^2}{4^{i+2}}, cores\right]}$$

$$+ \frac{2n^2(b-1)[8n(b^2+b+6) + 7b(b-2)]}{21b^3 \times min\left[\frac{n^2}{4^i}, cores\right]}$$

$$+ \frac{8n^3(b-1)(b^2+b-6)}{42b^3 \times min\left[\frac{n^2}{4^{i+1}}, cores\right]} + \frac{7n^3}{8 \times min\left[\frac{n^2}{4}, cores\right]}$$

$$(12)$$

Proof. Liu et al. in [10] has described several algorithms for distributed matrix inversion using LU decomposition. We are referring the most optimized one (stated as Algorithm 5, 6 and 7 in the paper) for the performance analysis. The core computation of the algorithm is done with 1) a call to a recursive method *LU* which basically decomposes the input matrix recursively until leaf nodes of the tree where the size of the matrix reaches the block size and 2) the computation after LU decomposition. The matrix inversion algorithm performs 7 additional multiplications (as given in Algorithm 5 in [10]) of size $\left(\frac{n}{2}\right)$, providing additional cost of, which is basically 7 matrix multiplications of dimension $\left(\frac{n}{2}\right)$. We call this as *Additional Cost* and can be obtained as follows

$$Comp_{AdditionalCost} = \frac{7 \times \left(\frac{n}{2}\right)^3}{min\left[\frac{n^2}{4}, cores\right]} \qquad (13)$$

There are two primary parts of the *LU* method − *if* part and *else* part. *If* part does the LU decomposition at the leaf nodes of the recursion tree while *else* part does for the internal nodes. *If* part requires 2 LU decomposition, 4 matrix inversion and 3 matrix multiplications and there are 2^{p-q} number of leaf nodes in the recursion tree. Each of these processing requires $O\left(\left(\frac{n}{b}\right)^3\right)$ time for a matrix of n dimension. Therefore, the total cost of the *if* part is

$$Comp_{leagNode} = 9 \times 2^{p-q} \times \left(\frac{n}{b}\right)^3 = 9 \times \left(\frac{n^3}{b^2}\right) \qquad (14)$$

The *else* part requires 4 *multiply*, 1 *subtraction* and 2 calls to *getLU* method. *getLU* method compose the LU of a matrix by taking 9 matrices of dimension 2^k and arranges them to return 3 matrices of size 2^{k+1}. It requires 4 *multiply* and 2 *scalarMul* methods of matrices of dimension 2^k.

The recursion scheme of LU decomposition is little bit different from *SPIN*. Here the number of *LU* call at level i is $2^i - 1$ instead of 2^i of *SPIN*. The computation and communication costs for the methods (summarized in Table 1) can be summed up to get equation 12.   □

5   EXPERIMENTS

In this section, we perform experiments to evaluate the execution efficiency of our implementation *SPIN* comparing it with the distributed LU decomposition based inversion approach (to be mentioned as *LU* from now) and scalability of the algorithm compared to ideal scalability. First, we select the fastest wall clock execution

Table 2: Summary of Test setup components specifications

Component Name	Component Size	Specification
Processor	2	Intel Xeon 2.60 GHz
Core	6 per processor	NA
Physical Memory	132 GB	NA
Ethernet	14 Gb/s	Infini Band
OS	NA	CentOS 5
File System	NA	Ext3
Apache Spark	NA	2.1.0
Apache Hadoop	NA	2.6.0
Java	NA	1.7.0 update 79

Figure 2: Fastest running time of LU and Strassen's based inversion among different block sizes

time among different partition size for each approach and compare them. Second, we conduct a series of experiments to individually evaluate the effect of partition size and matrix size of each competing approach. At last we evaluate the scalability of our implementation.

5.1 Test Setup

All the experiments are carried out on a dedicated cluster of 3 nodes. Software and hardware specifications are summarized in Table 2. Here *NA* means *Not Applicable.*

For block level multiplications both the implementation uses JBlas [9], a linear algebra library for Java based on *BLAS* and *LA-PACK*. We have tested the algorithms on matrices with increasing cardinality from (16×16) to (16384×16384). All of these test matrices have been generated randomly using Java Random class.

Resource Utilization Plan. While running the jobs in the cluster, we customize three parameters — the number of executors, the executor memory and the executor cores. We wanted a fair comparison among the competing approaches and therefore, we ensured jobs should not experience *thrashing* and none of the cases tasks should fail and jobs had to be restarted. For this reason, we restricted ourselves to choose the parameters value which provides good utilization of cluster resources and mitigating the chance of task failures. By experimentation we found that, keeping executor memory as 50 GB ensures successful execution of jobs without "*out of memory*" error or any task failures for all the competing approaches. This includes the small amount of overhead to determine the full request to YARN for each executor which is equal to 3.5 GB. Therefore, the executor memory is 46.5 GB. Though the physical memory of each node is 132 GB, we keep only 100 GB as YARN resource allocated memory for each node. Therefore, the total physical memory for job execution is 100 GB resulting 2 executors per node and a total 6 executors. We reserve, 1 core for operating system and hadoop daemons. Therefore, available total core is 11. This leaves 5 cores for each executor. We used these values of the run time resource parameters in all the experiments except the scalability test, where we have tested the approach with varied number of executors.
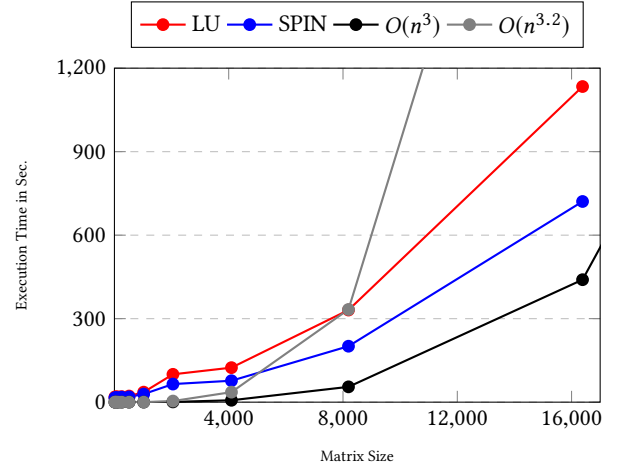
5.2 Comparison with state-of-the-art distributed systems

In this section, we compare the performance of *SPIN* with *LU*. We report the running time of the competing approaches with increasing matrix dimension in Figure 2. We take the best wall clock time (fastest) among all the running time taken for different block sizes. It can be seen that, *SPIN* takes the minimum amount of time for all matrix dimensions. Also, as expected the wall clock execution time increases with the matrix dimension, non-linearly (roughly as $O(n^3)$). Also, the gap in wall clock execution time between both *SPIN* and *LU* increases monotonically with input matrix dimension. As we shall see in the next section, both *LU* and *SPIN* follow a *U* shaped curve as a function of block sizes, hence allowing us to report the minimum wall clock execution time over all block sizes.

5.3 Variation with partition size

In this experiment, we examine the performance of *SPIN* with *LU* with increasing partition size for each matrix size. We report the wall clock execution time of the approaches when partition size is increased within a particular matrix size. For each matrix size (from (4096×4096) to (16384×16384)) we increase the partition size until we get a intuitive change in the results as shown in Figure. 3.

It can be seen that both *LU* and *SPIN* follows a *U* shape curve. However, *SPIN* outperforms *LU* when they have the same partition size, for all the matrix sizes. The reason of this is manifold. First of all, *LU* requires 9 times more $O\left(\frac{n}{b}\right)^3$ operations compared to a single operation of *SPIN*. For small partition sizes, where *leafNode* dominates the overall wall clock execution time, this cost is responsible for *LU*'s slower performance.

Additionally, when the partition size increases, the number of recursion level also increases and consequently the cost of *multiply* method increases which is the costliest method call. Though there is a difference between the number of recursion level for any partition size (), the additional matrix multiplication cost (as shown in Table 1) provides enough cost to slowdown *LU*'s performance.
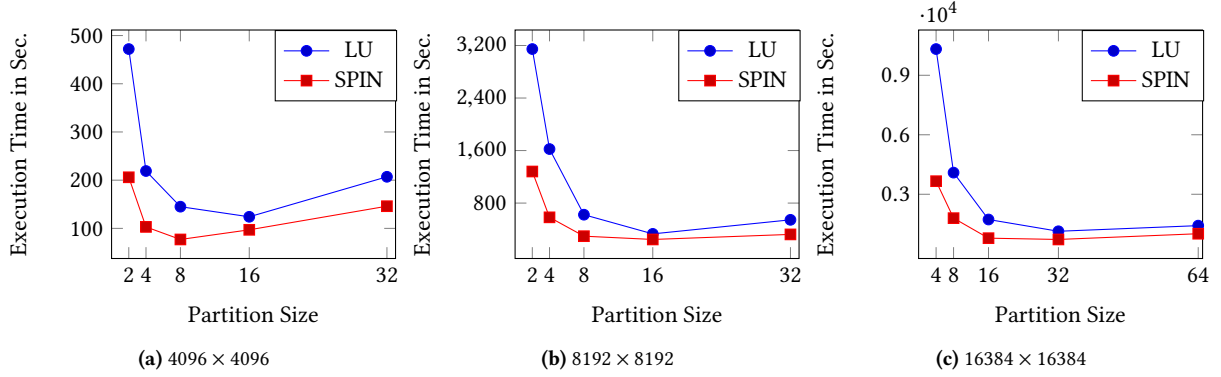
(a) 4096 × 4096   **(b)** 8192 × 8192   **(c)** 16384 × 16384

Figure 3: Comparing running time of LU and SPIN for matrix size (4096 × 4096), (8192 × 8192), (16384 × 16384) **for increasing partition size**

5.4 Comparison between theoretical and experimental result

In this experiment, we compare the theoretical cost of *SPIN* with the experimental wall clock execution time to validate our theoretical cost analysis. Figure 4 shows the comparison for three matrix sizes (from (4096 × 4096) to (16384 × 16384)) and for each matrix size with increasing partition size.

As expected, both theoretical and experimental wall clock execution time shows a *U* shaped curve with increasing partition size. The reason is that, for smaller partition sizes, the block size becomes very large for large matrix size. As a result, the single node matrix inversion shares most of the execution time and subdues the effect of matrix multiplication execution time which are processed distributedly. That is why we find large execution time at beginning, which is also depicted in Table 3, where experimental wall clock execution time is tabulated for different methods used in the algorithm for matrix of dimension 4096. It is seen that for $b = 2$, the *leafNode* cost is far more than matrix *multiply* method.

Later, when partition size further increases, the leaf node cost drops sharply as the cost depends on $\frac{n^3}{b^2}$, which decreases the cost by square of partition size. On the other hand, the number of *multiply* becomes large for enhanced recursion level, and thus the effective cost which subdues the effect of *leafNode* cost. As in Table 3, for $b = 8$ onwards the *multiply* cost becomes more and more dominating resulting further increase in wall clock execution time.

5.5 Scalability

In this section, we investigate the scalability of *SPIN*. For this, we generate three test cases, each containing a different set of two matrices of sizes equal to (4096 × 4096), (8192 × 8192) and (16384 × 16384). The running time vs. the number of spark executors for these 3 pairs of matrices is shown in Figure 5. The ideal scalability line (i.e. $T(n) = T(1)/n$ - where n is the number of executors) has been over-plotted on this figure in order to demonstrate the scalability of our algorithm. We can see that *SPIN* has a good scalability, with a minor deviation from ideal scalability when the size of the matrix is low (i.e. for (4096 × 4096) and (8192 × 8192)).

Table 3: Experimental results of wall clock execution time of different methods in *SPIN*(The unit of execution time is millisecond)

Method	b = 2	b = 4	b = 8	b = 16
leafNode	43504	11550	5040	3980
breakMat	178	441	901	1764
xy	2913	1353	693	309
multiply	7836	13116	23256	37968
subtract	1412	1854	2820	5592
scalar	333	728	1308	2450
arrange	307	685	1510	3074
Total	56483	29727	35528	55137

6 CONCLUSION

In this paper, we have focused on the problem of distributed matrix inversion of large matrices using Spark framework. To make large scale matrix inversion faster, we have implemented Strassen's matrix inversion technique which requires six multiplications in each recursion step. We have given the detailed algorithm, called *SPIN*, of the implementation and also presented the details of the cost analysis along with the baseline approach using LU decomposition. By doing that, we discovered that the primary bottleneck of inversion algorithm is matrix multiplications and that *SPIN* is faster as it requires less number of multiplications compared to LU based approach.

We have also performed extensive experiments on wall clock execution time of both the approaches for increasing partition size as well as increasing matrix size. Results showed that *SPIN* outperformed LU for all the partition and matrix sizes and also the difference increases as we increase matrix size. We also showed the resemblance between theoretical and experimental findings of *SPIN*, which validated our cost analysis. At last we showed that *SPIN* has a good scalability with increasing matrix size.

REFERENCES

[1] Emmanuel Agullo, Henricus Bouwmeester, Jack Dongarra, Jakub Kurzak, Julien Langou, and Lee Rosenberg. 2010. Towards an Efficient Tile Matrix Inversion of
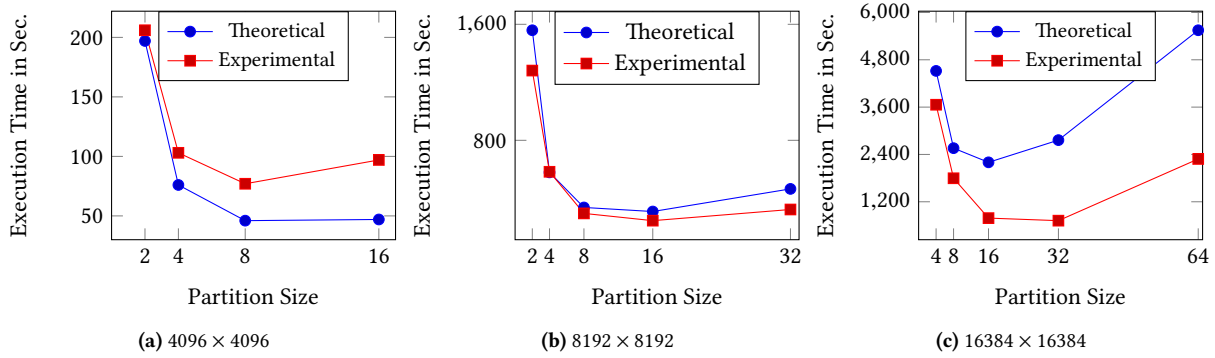
Figure 4: Comparing theoretical and experimental running time of *SPIN* for matrix size $(4096{\times}4096)$**,** $(8192{\times}8192)$**,** $(16384{\times}16384)$ **for increasing partition size**
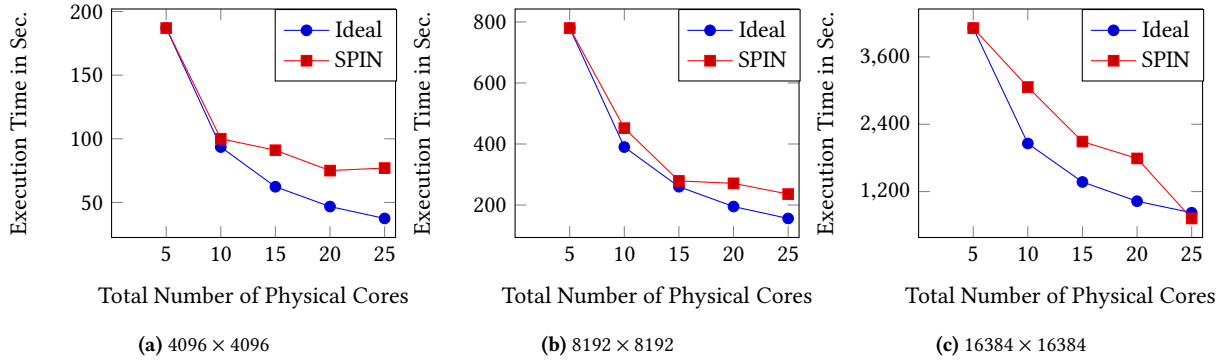

Figure 5: The scalability of *SPIN*, in comparison with ideal scalability (blue line), on matrix (4096×4096)**,** (8192×8192) **and** (16384×16384)

Symmetric Positive Definite Matrices on Multicore Architectures.. In *VECPAR*, Vol. 10. Springer, 129–138.

[2] Ali Al Essa and Miad Faezipour. 2017. MapReduce and Spark-Based Analytic Framework Using Social Media Data for Earlier Flu Outbreak Detection. In *Industrial Conference on Data Mining*. Springer, 246–257.

[3] Steven C Althoen and Renate Mclaughlin. 1987. Gauss-Jordan reduction: A brief history. *The American mathematical monthly* 94, 2 (1987), 130–142.

[4] Paolo Bientinesi, Brian Gunter, and Robert A Geijn. 2008. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software (TOMS)* 35, 1 (2008), 3.

[5] Adrian Burian, Jarmo Takala, and Mikko Ylinen. 2003. A fixed-point implementation of matrix inversion using Cholesky decomposition. In *Circuits and Systems, 2003 IEEE 46th Midwest Symposium on*, Vol. 3. IEEE, 1431–1434.

[6] Pablo Ezzatti, Enrique S Quintana-Orti, and Alfredo Remon. 2011. High performance matrix inversion on a multi-core platform with several GPUs. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE, 87–93.

[7] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*, Vol. 14. 599–613.

[8] KK Lau, MJ Kumar, and R Venkatesh. 1996. Parallel matrix inversion techniques. In *Algorithms & Architectures for Parallel Processing, 1996. ICAPP 96. 1996 IEEE Second International Conference on*. IEEE, 515–521.

[9] Linear Algebra for Java 2017. Linear Algebra for Java. http://jblas.org/. (2017). [Online; accessed 30-July-2017].

[10] Jun Liu, Yang Liang, and Nirwan Ansari. 2016. Spark-based large-scale matrix inversion for big data processing. *IEEE Access* 4 (2016), 2166–2176.

[11] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al.

2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[12] Rahul Palamuttam, Renato Marroquín Mogrovejo, Chris Mattmann, Brian Wilson, Kim Whitehall, Rishi Verma, Lewis McGibbney, and Paul Ramirez. 2015. SciSpark: Applying in-memory distributed computing to weather event detection and tracking. In *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2020–2026.

[13] William H Press. 2007. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.

[14] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. 2012. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 197–210.

[15] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. 2013. A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA. *Computers & Structures* 128 (2013), 31–37.

[16] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.

[17] Jingen Xiang, Huangdong Meng, and Ashraf Aboulnaga. 2014. Scalable matrix inversion using mapreduce. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 177–190.

[18] Kaiqi Yang, Yubai Li, and Yijia Xia. 2013. A parallel method for matrix inversion based on gauss-jordan algorithm. *Journal of Computational Information Systems* 9, 14 (2013), 5561–5567.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.