Improving Execution Concurrency of Large-Scale Matrix Multiplication on Distributed Data-Parallel Platforms

Rong Gu, *Student Member, IEEE*, Yun Tang, *Student Member, IEEE*, Chen Tian, Hucheng Zhou, Guanru Li, Xudong Zheng, and Yihua Huang

Abstract—Matrix multiplication is a dominant but very time-consuming operation in many big data analytic applications. Thus its performance optimization is an important and fundamental research issue. The performance of large-scale matrix multiplication on distributed data-parallel platforms is determined by both computation and IO costs. For existing matrix multiplication execution strategies, when the execution concurrency scales up above a threshold, their execution performance deteriorates quickly because the increase of the IO cost outweighs the decrease of the computation cost. This paper presents a novel parallel execution strategy *CRMM (Concurrent Replication-based Matrix Multiplication)* along with a parallel algorithm, Marlin, for large-scale matrix multiplication on data-parallel platforms. The CRMM strategy exploits higher execution concurrency for sub-block matrix multiplication with the same IO cost. To further improve the performance of Marlin, we also propose a number of novel system-level optimizations, including increasing the concurrency of local data exchange by calling native library in batch, reducing the overhead of block matrix transformation, and reducing disk heavy shuffle operations by exploiting the semantics of matrix computation. We have implemented Marlin as a library along with a set of related matrix operations on Spark and also contributed Marlin to the open-source community. For large-sized matrix multiplication, Marlin outperforms existing systems including Spark MLlib, SystemML and SciDB, with about $1.29\times$, $3.53\times$ and $2.21\times$ speedup on average, respectively. The evaluation upon a real-world DNN workload also indicates that Marlin outperforms above systems by about $12.8\times$, $5.1\times$ and $27.2\times$ speedup, respectively.

Index Terms—Parallel matrix multiplication, data-parallel algorithms, machine learning library

1 Introduction

Machine learning and data mining algorithms are critical for knowledge discovery from big data. Distributed data-parallel platforms, such as Hadoop MapReduce [1], [2] and Spark [3], [4], provide friendly development frameworks for users by automatically handling the underlying parallel computation details (e.g., task split and fail over). As a result, a number of prevalent machine learning and data mining algorithm libraries are developed on distributed data-parallel platforms [5], [6], [7]. Matrix computation, including matrix multiplication and factorization, is the core of many big data machine learning and data mining applications such as mining social networks, recommendation systems and natural language processing [8], [9], [10]. Therefore, many libraries or frameworks built on top of distributed data-parallel platforms, such as Spark MLlib [5] and SystemML [7] on Spark, start to provide native matrix computation interfaces. With these matrix abstractions, data scientists can write imperative programs without needing to worry about the underlying complicated details of distributed system implementations.

Among these matrix operations, multiplication is a very important but time-consuming computation required by many machine learning algorithms such as page rank, logistic regression and Deep Neural Network (DNN) [9]. Take the DNN algorithm as an example, 95 percent of GPU computation and 89 percent of CPU computation are occupied by matrix multiplications [11]. Also, many matrix factorization operations can be approximated by multiplication operations [7]. Thus, matrix multiplication is the major focus of performance optimization for many big data analytical algorithms or applications. In fact, large-scale matrix multiplication can hardly be handled by the single-node matrix computation libraries due to hardware resource limitation. Therefore, there is an ever-increasing need for scalable and efficient matrix computation systems.

The performance of a large-scale matrix multiplication execution strategy is basically determined by the total time costs of its computation and IO on distributed data-parallel platforms. A typical process of most large-scale matrix multiplication execution strategies works as follows: 1) each operand matrix is split to many sub-matrices and distributed to many cores; 2) partial multiplications are concurrently performed on different cores; 3) the intermediate

- *R. Gu, Y. Tang, C. Tian, and Y. Huang are with State Key Laboratory for Novel Software Technology, Nanjing University, Jiangsu Sheng 210000, China. E-mail: {gurong, tianchen, yhuang}@nju.edu.cn, tangyun@smail.nju.edu.cn.*
- *H. Zhou, G. Li, and X. Zheng are with Microsoft Research, Beijing 100084, China. E-mail: {guanrl, xuzhen, huzho}@microsoft.com.*

Manuscript received 8 July 2016; revised 17 Jan. 2017; accepted 1 Mar. 2017. Date of publication 22 Mar. 2017; date of current version 9 Aug. 2017. Recommended for acceptance by X. Gu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2686384

outputs are aggregated to generate the final result matrix. Generally speaking, the more cores that are used, the higher parallelism the computation can be achieved, but also the more IO traffic are introduced among cores. Here the IO bottleneck can be either disk access or network communication, or both. For a given execution strategy, the computation cost is a decreasing function of the execution concurrency, while the IO cost is an increasing function of the execution concurrency.

The performance of existing matrix multiplication execution strategies deteriorate quickly when execution concurrency scales up above a threshold. To parallelize large-scale matrix multiplication, Spark MLlib and SystemML adopt the *Replication-based Matrix Multiplication* (*RMM*) and *Cross-Product Matrix Multiplication* (*CPMM*) [10] execution strategies. Both *RMM* and *CPMM* have a relatively large IO cost. As a result, when increasing the number of cores, the increase of the IO cost may quickly outweigh the decrease of the computation cost (analysis details in Section 2.2).

In this paper, we present a novel parallel execution strategy along with a parallel algorithm, Marlin, for large-scale matrix multiplication on data-parallel platforms. Marlin exploits concurrency from perspectives of computation, IO and local data exchange. For the matrix multiplication execution strategy, we propose *Concurrent Replication-based Matrix Multiplication* (*CRMM*) strategy. Compared with existing execution strategies, *CRMM* significantly reduces the IO cost by exploiting execution concurrency of two operand matrices in all three dimensions. Consequently, *CRMM* can exploit higher concurrency for sub-block matrix multiplication with the same IO cost to reduce the overall execution cost. We also prove that the existing *RMM* and *CPMM* strategies are actually two special cases of *CRMM*.

Furthermore, we propose a number of novel system-level optimizations to further improve the performance of Marlin. First, when interfacing with native libraries on a node, the *Batch Calling Native Library (BCNL)* strategy is used to increase the concurrency of data exchange and amortize the overhead of system calls. Second, instead of using the inefficient coordination matrix transformation, we propose the *Slicing Matrix Transformation (SMT)* strategy to reduce the overhead of block matrix transformation. Third, instead of naively using the general shuffle mechanism provided by data-parallel platforms, Marlin adopts the *Light-Shuffle sub-Matrix co-Grouping (LSMG)* strategy, which reduces disk heavy shuffle operations by half by exploiting the semantics of matrix computation. Lastly, we use an existing heuristic algorithm to choose the appropriate partitioning strategy for various matrix scales from real-world scenarios.

We have implemented Marlin[1] as a library along with a set of related matrix operations on the widely-used distributed data-parallel system Spark. Marlin provides a group of easy-to-use matrix APIs to support easy programming for matrix model-based machine learning algorithms. With Marlin's APIs, an end-to-end DNN training algorithm can be programmed in less than 30 lines of code. Note that the execution strategy and optimizations proposed in this paper can also be applied to other distributed data-parallel platforms such as Apache Flink [12], Dryad [13] etc.

```
1  val layer1 = 28 * 28
2  val layer2 = 300
3  val layer3 = 10
4  val iterations = 300
5  val fraction = 0.01
6  val learningRate = 0.1
7
8  // initialize weights
9  val W = DenseMatrix.rand[Double](layer1, layer2)
10 val V = DenseMatrix.rand[Double](layer2, layer3)
11 // including partitionBy and cache operations
12 val (data, labels) = loadMnistImage(sc, path)
13
14 val batchSize = fraction * data.numRows()
15
16 for (i <- 0 until iterations){
17     // Propagate through the network
18     val (input, label) = loadBatches(data, fraction)
19     val wIn = input * W
20     val wOut = activate(wIn)
21     val vIn = wOut * V
22     val vOut = activate(vIn)
23     // Back Propagate the errors
24     val vDelta = dActivate(vIn) .* (vOut — label)
25     val wDelta = dActivate(wIn) .* (vDelta * V.t)
26     // update weight matrix
27     V -= learningRate .* wOut.t * vDelta / batchSize
28     W -= learningRate .* input.t * wDelta / batchSize
29 }
```

Fig. 1. A three-layer fully-connected DNN training program written with Marlin APIs

We have evaluated the performance of Marlin on a cluster of 20 physical server nodes with 320 cores in total, by comparing it with three cutting-edge alternatives: the built-in matrix computation library of Spark MLlib, the matrix-based programming platform SystemML [7], [10], [14] that uses Spark as the underlying execution engine, and the widely-used matrix library provided by SciDB [15]. Experimental results over various benchmarks show that for large-sized matrix multiplication, Marlin outperforms Spark MLlib, SystemML and SciDB with about $1.29\times$, $3.53\times$ and $2.21\times$ speedup, respectively; for multiplication of large matrix by small matrix, it achieves $2.68\times$, $1.72\times$ and $11.2\times$ speedup, respectively. The evaluation upon a real-world DNN workload also indicates that Marlin outperforms them by about $12.8\times$, $5.1\times$ and $27.2\times$ speedup, respectively.

The rest of paper is organized as follows. Section 2 introduces the background of this research. Section 3 presents the *CRMM* strategy in details. Section 4 discusses several important system-level optimizations. Section 5 provides the design and interface details of the framework. The performance evaluation results are presented in Section 6. We discuss related work in Section 7. Section 8 concludes the paper.

2 Background

2.1 Matrix Multiplication for Machine Learning

Matrix is an important mathematic model to represent many machine learning and data analytic algorithms or applications, such as page rank, logistic regression and DNN [9]. Further, matrix multiplication is a frequently-used but very time-consuming operation for many such algorithms. We take the DNN training as an example. A three-layer fully-connected neural network training process expressed by Marlin API[2] is present in Fig. 1. The goal of this algorithm is to optimize the model weights `W` (line 9) and `V` (line 10) among these three

1. Marlin is now available at https://github.com/PasaLab/marlin

2. Details of Marline programming API are presented in Section 5.

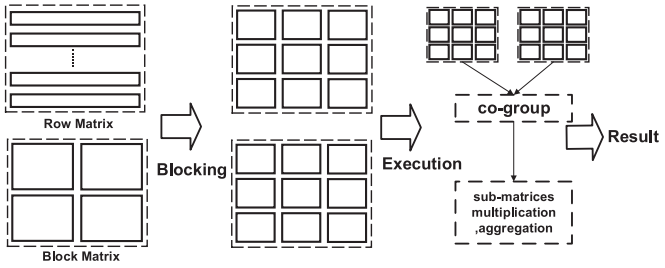


Fig. 2. The workflow of distributed matrix multiplication.

layers. All `input`, `W` and `V` are represented as matrices with sizes $input\text{-}num \times layer1$, $layer1 \times layer2$ and $layer2 \times layer3$, respectively. The training data are fed to the neural network batch by batch during each training iteration. At the beginning, The MNIST [16] training `data` and `labels` are generated by `loadMnistImage()` (line 12). Then, during each training iteration, the mini-batch data (`input`) is prepared by `loadBatches()` (line 18). There are two phases of computation in each iteration during the model training: the feed forward phase (line 19-22) starting from bottom layer, and the back propagate phase (line 24-25) starting from top layer. The activation function `activate` and its derivation `dActivate` function are adopted to calculate the deltas of model weights during these two computation phases. Lastly, the model weights are updated by subtracting the trained model deltas in each iteration (line 27, 28).

The computation of the training process is dominated by matrix multiplication (line 19, 21, 25, 27 and 28). For such a DNN program, it has been reported that 95 percent of GPU computation and 89 percent of CPU computation are occupied by matrix multiplications alone [11].

Therefore, optimizing the performance of large-scale matrix multiplication becomes a very important and fundamental research issue for many big data analytic applications. For this reason, similar to related works [17], [18], this paper focuses on optimizing large-scale matrix multiplication operation by utilizing distributed data-parallel computing models and platforms.

2.2 Distributed Matrix Multiplication Strategies

Large scale matrix multiplication has two categories: small matrix multiplying large matrix, and two large matrices multiply with each other. For small matrix multiplying large matrix, the commonly-used execution strategy is $Broadcast\ MM$ (Broadcast Matrix Multiplication), or $MapMM$ (Map-side Matrix Multiplication) [7], [10], which broadcasts the small matrix to each computing node with one blocked sub-matrix of the large matrix. We also adopt the $BroadcastMM$ strategy for this case in our design. In this paper, *we focus on optimizing the execution strategy of two-large-matrix multiplication on distributed data-parallel platforms.*

As shown in Fig. 2, to perform parallel execution on top of a parallel computation platform, a blocking strategy is used to split a large matrix to smaller pieces (i.e., sub-matrix). Consider two matrices A and B, the goal of a blocking strategy is to split A and B into sub-matrices of $M_b \times K_b$, $K_b \times N_b$ blocks respectively.[3]

3. Note that b is not a variable, it means that matrices A and B are represented in the blocked format after splitting.

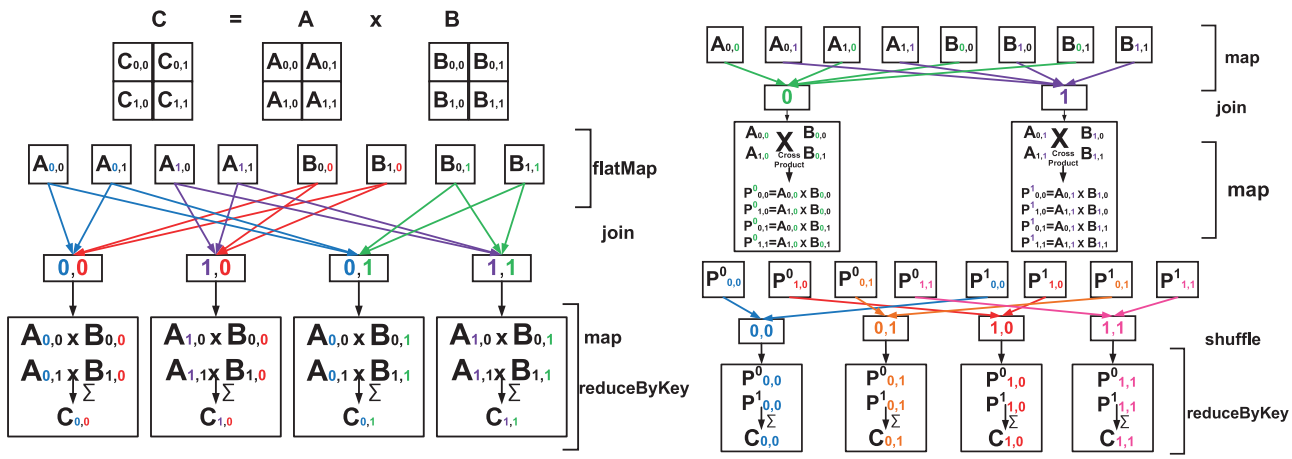
An execution strategy for large-scale matrix multiplication is responsible for scheduling the execution workflow of the blocked sub-matrices on the underlying parallel computing platforms. For an existing blocking scheme of a matrix multiplication, different execution strategies can have different performance. On the other hand, for a specific execution strategy, we can also derive the optimized blocking scheme if we want to use all available cores simultaneously.

There exist two widely-used large-matrix multiplication execution strategies on distributed data-parallel platforms [10], [14]: Replication-based Matrix Multiplication (RMM) and Cross-Product Matrix Multiplication ($CPMM$). Their detailed execution workflows are illustrated in Fig. 3a and 3b respectively [10]. Consider two matrices A and B split into sub-matrices with $M_b \times K_b$ blocks in A and $K_b \times N_b$ blocks in B. Then, the matrix multiplication can be represented as the blocked format: $C_{i,j} = \sum A_{i,k}B_{k,j}, i < M_b, k < K_b, j < N_b$.

RMM is used in both SystemML [7] and Spark MLlib [5]. As shown in Fig. 3a, there exists only one shuffle phase in RMM. In order to compute one result block $C_{i,j}$, the reduce stage should obtain all the required blocks from A and B. Also, $A_{i,k}$ and $B_{k,j}$ have to be replicated together to generate the multiplication result block $C_{i,j}$. In the *join* stage of RMM, each block of A and B is replicated N_b and M_b times respectively. As a result, the size of shuffled data during this phase is $N_b|A| + M_b|B|$. In the reduce stage, $M_b \times N_b$ tasks can run concurrently to execute sub-block multiplications. Among the $M_b \times N_b$ tasks, each task will execute K_b times of matrix multiplication and then the intermediate results are aggregated to generate the final result block $C_{i,j}$. *Thus, for a given execution parallelism degree* Par$= \boldsymbol{M_b \times N_b}$ *in* RMM, *the size of data written to/read from disk is* $\boldsymbol{N_b|A| + M_b|B|}$, *which is time-consuming when* Par *is large.*

$CPMM$ is one of the execution strategies adopted by SystemML [7]. As shown in Fig. 3b, different from RMM, $CPMM$ does not need to replicate data of matrix A and B multiple times. Instead, it implements a cross-product strategy that requires one *join* and one *shuffle* phase. In the *join* phase, the input matrices A and B are grouped into the input blocks $A_{i,k}$s and $B_{k,j}$s by the joint key k and then written to disk. The size of shuffled data during this phase is $|A| + |B|$. Then the next map stage performs a cross product to compute $P^k_{i,j} = A_{i,k}B_{k,j}$. As blocks are grouped by joint key k, there are K_b sub-block multiplication tasks running in parallel at most. In the shuffle phase, these intermediate results are then written to disk by grouping all the $P^k_{i,j}$s by the key (i, j). The size of shuffled data during this phase is $K_b|C|$. Finally in the reduce stage, the *reduceByKey* transformation computes $C_{i,j} = \sum P^k_{i,j}$. It is clear that *for a given execution parallelism degree* Par $= \boldsymbol{K_b}$ *in* CPMM, *the size of shuffled data for aggregation is* $\boldsymbol{K_b|C|}$ *which is huge when* Par *is large.*

To demonstrate the relationship between the parallelism degree and shuffle size of different execution strategies, we conduct an empirical experiment on a 12-node cluster with 16 cores each node, i.e., totally 192 cores in the cluster. The maximal degree of task execution parallelism is also 192. For simplicity, we take the input matrices as the same size so that the shuffle data size can be represented as the multiple size of matrix A. In Fig. 4 we can see that the shuffle



(a) RMM (b) CPMM

Fig. 3. The execution strategies of matrix multiplication on distributed data-parallel systems.

data size grows very fast with the increase of the parallelism degrees for *CPMM* and *RMM* strategies. In the next section, we will demonstrate that when increasing the number of cores for *RMM* and *CPMM*, the increase of the IO cost quickly outweighs the decrease of the computation cost.

3 CRMM Execution Strategy

For two-large-matrix multiplication, we present our key optimization called *CRMM* execution strategy that increases the sub-matrix multiplication concurrency and reduces the IO cost in the entire workflow. Compared with *RMM* and *CPMM* (Section 2), *CRMM* has less computation cost (due to higher concurrency) with a fixed IO cost. As shown in Fig. 4, *CRMM* has less IO cost for a fixed sub-matrix multiplication concurrency.

Observation. Consider two input matrices A and B split into sub-matrices with $M_b \times K_b$ blocks in A and $K_b \times N_b$ blocks in B. From analysis in Section 2.2 and Fig. 4, *RMM* and *CPMM* incur a lot of shuffle IO when increasing the degree of execution parallelism, which may result in poor overall performance. We summarize the theoretical performance analysis of these matrix execution strategies in Table 1.

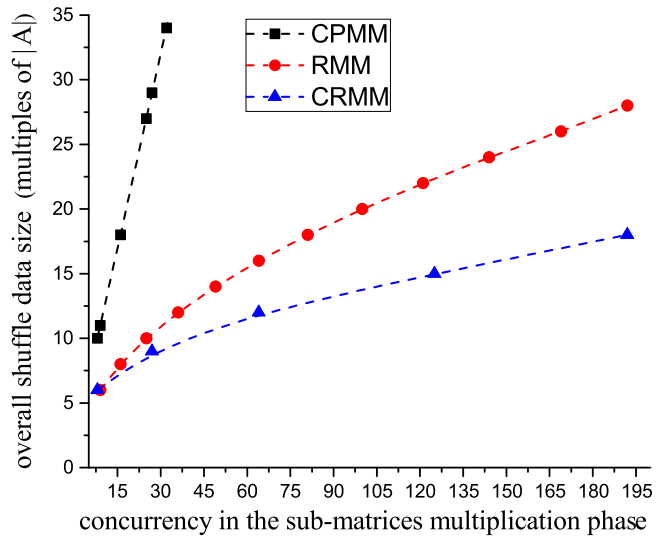


Fig. 4. An intuitive comparison of overall shuffle data size versus concurrency among different matrix multiplication strategies (given 192 cores in total and input matrices are the same size).

The observation is that: for a given *Par* objective (i.e., the number of concurrent cores), the IO cost of *RMM* is determined by M_b and N_b, given the constraint of $Par = M_b \times N_b$. We can safely assume that its IO cost is proportional to $Par^{\frac{1}{2}}$. Similarly, the cost of *CPMM* is determined by K_b, given the constraint of $Par = K_b$, and its IO cost is proportional to *Par*. This explains why compared with *RMM*, *CPMM* is inferior in concurrency: its IO cost quickly increases along with *Par*.

Solution. To overcome the limitations of *RMM* and *CPMM*, we propose an optimized matrix multiplication strategy called *CRMM*. The execution strategy of *CRMM* is illustrated in Fig. 5. It extends from *RMM* but achieves higher concurrency for sub-block matrix multiplication. The concurrency of the two operand matrices in all three dimensions are exploited. In the *join* stage of *CRMM*, each block of A and B is replicated N_b and M_b times respectively. However, different from *RMM*, *CRMM* does not co-group all related $A_{i,k}$ and $B_{k,j}$ to generate the result block $C_{i,j}$ in the first stage. This way, the computation of each $P^k_{i,j} = A_{i,k}B_{k,j}$ can be executed simultaneously, which means $M_b \times K_b \times N_b$ tasks can run concurrently to execute sub-block multiplications. The size of shuffled data during this phase is the same as *RMM*, that is $N_b|A| + M_b|B|$. Then, similar to *CPMM*, we need another shuffle phase to aggregate these intermediate results $P^k_{i,j}$. During this stage, all the $P^k_{i,j}$s are grouped together by the key (i, j). Thus, the size of shuffled data during this phase is $K_b|C|$. Finally in the reduce stage, the *reduceByKey* transformation computes $C_{i,j} = \sum P^k_{i,j}$.

Analysis. We also present the theoretical analysis for *CRMM* in Table 1. As illustrated in Table 1, the execution concurrency of *CRMM* can reach $M_b \times K_b \times N_b$. As a result, its IO cost is proportional to $Par^{\frac{1}{3}}$. Compared with *RMM* and *CPMM*,*CRMM* significantly reduces the IO cost by exploiting concurrency of two operand matrices in all three dimensions. For example, when $Par = 2^{12}$ (i.e., around 4,000 cores), the shuffle size for *CPMM* and *RMM* are $O(2^{12})$ and $O(2^6)$ respectively; while it is only $O(2^4)$ for *CRMM*.

Additionally, we find that *CRMM* is equivalent to *CPMM* or *RMM* in two special cases. On one hand, if the

TABLE 1
Analysis of Different Matrix Multiplication Strategies (Par Means the Number of Logical Cores Used)

Strategy	shuffle data size in 1st phase	parallelism in sub-block matrix multiplication	shuffle data size in 2nd phase	parallelism in aggregating intermediate blocks
RMM	$N_b\|A\| + M_b\|B\|$	$M_b \times N_b = Par$	None	None
$CPMM$	$\|A\| + \|B\|$	$K_b = Par$	$K_b\|C\|$	$M_b \times N_b = Par$
$CRMM$	$N_b\|A\| + M_b\|B\|$	$M_b \times K_b \times N_b = Par$	$K_b\|C\|$	$M_b \times N_b = Par$

dimension k is extremely larger than m and n, i.e., a very flattened matrix multiplying a tall and thin matrix, the blocked format of $M_b \times K_b \times N_b$ can be derived to $1 \times K_b \times 1$. In this case, $CRMM$ is equivalent to $CPMM$ theoretically. Actually, this case happens twice in the pseudo-code of DNN (line 27 and 28 in Fig. 1). It is obvious that as the final result matrix is really smaller than input matrices, using the RMM strategy to gather all related sub-blocks in one single task would result in poor performance. On the other hand, if the dimension k is extremely smaller than m and n, the blocked format of $M_b \times K_b \times N_b$ can be derived to $M_b \times 1 \times N_b$. Under this situation, $CRMM$ is equivalent to RMM in theory and thus another shuffle phase can be avoided.

Verification. To prove our deduction of these three strategies, we test the overall performance on a 12-node Spark cluster (with the same number of cores as in Fig. 4) with the same size of input matrices as (30000×30000). For the RMM strategy, we let M_b equal N_b, and increase them from 4×4 (i.e., using 16 cores) to 13×13 (i.e., using 169 cores). For the $CPMM$ strategy, we simply increase the value of K_b. For the $CRMM$ strategy, we let M_b equal N_b equals K_b, and increase them from $3 \times 3 \times 3$ (i.e., using 27 cores) to $6 \times 5 \times 6$ (i.e., using 180 cores).

The experimental results are shown in Fig. 6. Consistent with our analysis, to achieve the best performance, RMM should only use 81 cores instead of all the cores. $CPMM$ is even worse: the threshold value is around 20-25 cores. As a comparison, the execution time of $CRMM$ keeps decreasing together with the increase of the number of cores; due to the physical resource limitations, we cannot explore the situation of more cores, but we expect that there also exists a threshold value for $CRMM$. Comparing the best performance among the three strategies, we can see that the execution time of RMM is only half of that for $CPMM$, while the execution time of $CRMM$ is even 15 percent lower than that for RMM.

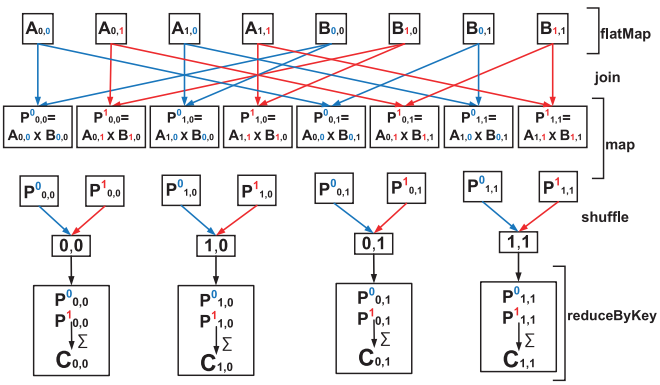


Fig. 5. The CRMM execution strategy.

4 System-Level Optimizations

In this section, we propose several system-level optimizations to further improve the matrix execution performance. First, we introduce the *Batch Calling Native Library (BCNL)* to improve the efficiency of native library calling by increasing the concurrency of data exchange. Second, we elaborate the *Slicing Matrix Transformation (SMT)* strategy to reduce the overhead of block matrix transformation. Third, we present the *Light-Shuffle sub-Matrix co-Grouping (LSMG)* optimization to reduce IO heavy shuffle operations by exploiting the semantics of matrix computation (Section 4.3). Last, we explain the existing heuristic algorithm that we adopt to choose the appropriate partitioning strategy for various matrix scales from real-world scenarios.

4.1 Batch Calling Native Library (BCNL)

Intuition. Matrix operation is computation-intensive. Thus, instead of performing linear algebra computations on JVM, many high-level matrix libraries, including Marlin and Spark MLlib, offload the single-node CPU intensive operations to the high performance linear algebra computing libraries, such as ATLAS [19], LAPACK [20] or Intel MKL [21] via Java Native Interface (JNI). For example, Spark MLlib provides API for multiplying large distributed row-matrix by small matrix. It is executed with the broadcast strategy. When a row-matrix multiplies a local small matrix, it invokes the native library to multiply each row with the broadcasted matrix. Similarly, Marlin and Spark MLlib adopt Breeze,[4] a numerical processing library in Scala, as the underlying library.

However, this may decrease the performance since invoking native library will introduce extra overhead such as Java object initialization and parameter passing. Performance becomes even worse if there is only little computation, for example, vector addition or matrix-vector multiplication. We conduct a group of comparison experiments using one thread on a single computing node (the hardware configuration described in Section 6.1) to verify this. The first case, a 1000×1000 sized matrix multiplies another 1000×1000 sized matrix. The second case, a 1×1000 sized vector multiplies a 1000×1000 sized matrix for 1000 times. Both cases call the native library BLAS using a CPU core for computation, and the first case can speed up about $6.3\times$ than the second case.

Solution. To increase the concurrency of data exchange, Marlin presents a solution to efficiently use the native library. Rather than call the native library row by row, which is actually matrix-vector multiplication in BLAS-2 level, Marlin first initializes a matrix with row-based data

4. Breeze, https://github.com/scalanlp/breeze

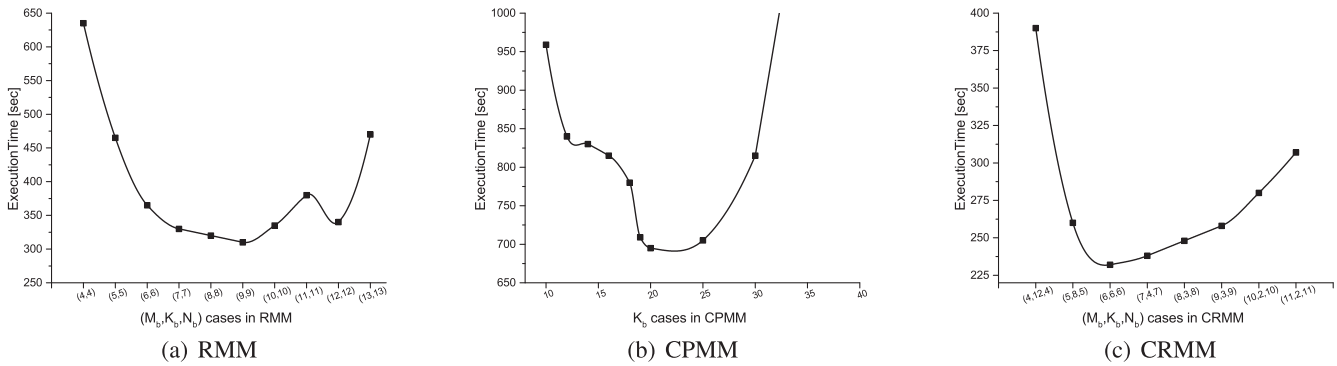


Fig. 6. Tuning the performance of different matrix multiplication strategies on a 12-node Spark cluster.

partition and then multiplies it with the broadcasted matrix by calling the native BLAS-3 level native library. At the end, it retrieves the result of each row by decomposing the result matrix in row-wise. This changes a batch of matrix-vector multiplication into a single matrix-matrix multiplication. As a result, it improves the computation performance from the BLAS-2 to BLAS-3 level.

Besides, Marlin also takes the advantage of column-major property when invoking the native library BLAS. In Breeze, matrix is stored in column-major, which is the same layout as that in the native ATLAS. Thus, during initializing the temporary matrix of each partition, vectors are directly assigned to the matrix by the column-assignment, which leads to less cache miss than the row-assignment method. Therefore, we broadcast the transposition of the local matrix and place it in the left side to multiply temporary matrix of each partition, instead of using the normal processing method.

4.2 Slicing Matrix Transformation (SMT)

Similar to Spark MLlib, there are two types of matrix representations in Marlin. When processing row-matrices multiplication, efficient transformation between row-matrix and block-matrix is usually required. Spark MLlib transforms a row-matrix to a block-matrix by emitting a large number of (i, j, v) coordinates during the shuffle phase. This may bring in a large number of intermediate objects with huge memory footprint pressure and unnecessary data redundancy. Moreover, after such transformation, Spark MLlib treats each sub-matrix as sparse matrix, which makes it hard to execute subsequent matrix computation in the native library.

Marlin provides an efficient mechanism to shuffle sub-vectors or sub-blocks to complete transformation between these two types of representations. The key observation is that some neighboring elements in the old representation are also neighbors in the new representation. Instead of encoding and sending every matric element one-by-one, Marlin searches for sub-vectors or sub-blocks that are also neighbors in the new representation; such a sub-vector or sub-block is encoded and sent as a whole. This method is also applied to row-block and block-block transformations.

4.3 Light-Shuffle Sub-Matrix Co-Grouping (LSMG)

Intuition. In any of the $CPMM$, RMM and $CRMM$ strategies, there always exists a *join* stage to co-group the corresponding sub-matrices together for further computation. Existing distributed matrix computation systems, such as SystemML [7], HAMA [17] and Spark MLlib [5], co-group sub-matrices by adopting the general shuffle mechanism as illustrated in Fig. 7a. The general shuffle mechanism incurs writing and reading a lot of small partition files. This results in heavy random disk I/O operations. Besides, some shuffle policies need to perform external sorting. These factors will slow down the shuffle process. As analyzed in [22], the performance of distributed data-parallel platforms, such as Spark, suffers from heavy random I/O cost.

Solution. Therefore, we propose an optimization called *Light-Shuffle sub-Matrix co-Grouping (LSMG)* to reduce the heavy disk shuffle operations when co-grouping the sub-matrices. This optimization takes advantage of the sub-matrix correspondence information which is a pre-known semantics in matrix computation. As shown in Fig. 7b, with these semantics, each sub-matrix of A can directly fetch the corresponding sub-matrix of B. This way, a large number of

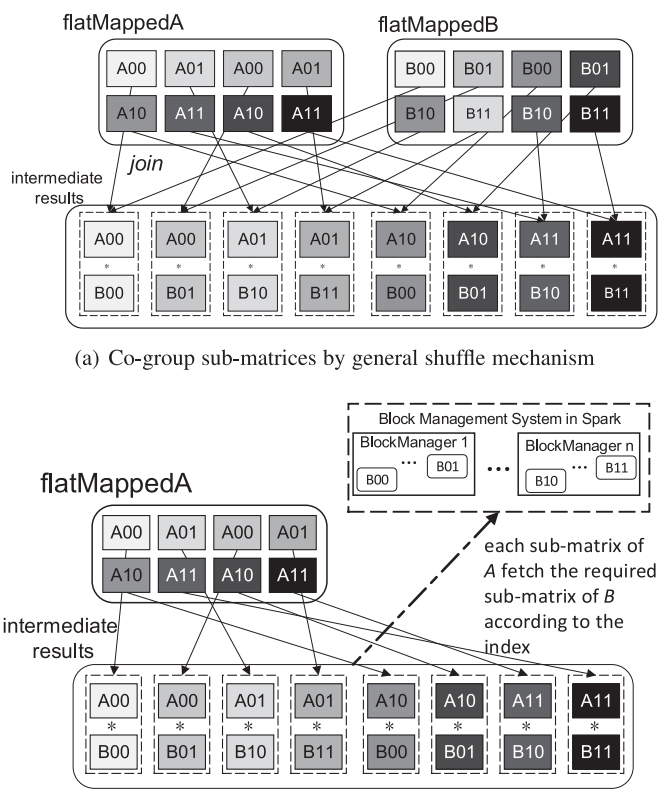


Fig. 7. Light-Shuffle sub-Matrix co-Grouping optimization (*LSMG*).

the sub-matrices in B do not need to be written to and read from disks.

Here, we take the Spark platform as an example to elaborate the implementation of $LSMG$ optimization. We reduce shuffle data writing by optimizing the storage mechanism of the block manager in Spark. First, we replicate and store the sub-matrices on each computing node into its local block manager memory. Second, we register the data information back to the driver client. This way, one of the block-matrix can avoid writing data to disk in the map side. Thus, this optimization can reduce heavy disk shuffle operations by half. To achieve load balance and high parallelism, another group of sub-matrices are still distributed across the cluster so that each task can be assigned with at least one sub-matrix for multiplication.

During the *join* stage, the related sub-matrices are grouped together according to their indexes by directly reading the sub-matrices from the block managers without disk reading incurred by the general shuffle mechanism. We modified the Spark source code to support this mechanism.

Algorithm 1. *ApproachSelection(A,B,thres,env)*

Input: A is a $m \times k$ matrix, B is a $k \times n$ matrix, $thres$ is the threshold of size, env is the physical cluster environment configurations
Output: $C = A \times B$, the output size is $m \times n$

```
1  begin
2    if A.size < thres and B.size < thres and m × n < thres then
3      Local_A = collect(A)
4      Local_B = collect(B)
5      C = Local_Multiplication(Local_A, Local_B)
6    else if A.size < thres then
7      BC_A = broadcast(A)
8      C = BroadcastMM(BC_A, B)
9    else if B.size < thres then
10     BC_B = broadcast(B)
11     C = BroadcastMM(A, BC_B)
12   else
13     // get proper arguments to split matrices
14     conf = getBestSplitParameter(A.size, B.size, env)
15     // choose CRMM strategy to execute matrix multiplication
16     C = Matrix_Multiplication(A, B, conf)
17   return C
```

4.4 Adaptive Strategy Selection

As many studies have proved that different matrix multiplication blocking strategies may result in very different performance even for the same input matrices [10]. Therefore, similar to SystemML [7], Marlin also implemented a heuristic approach for selecting the appropriate execution strategy as well as tuning the block-splitting number. Given two matrices $A \times B$, with the sizes $m \times k$ and $k \times n$, respectively, the detailed algorithm is described in Algorithm 1. An empirical threshold $thres$ is used to indicate the criteria whether or not a matrix is large. If all matrices are smaller than $thres$, we collect them to one single node to execute matrix multiplication. Otherwise, if only one matrix is smaller than $thres$, the broadcasting strategy is used. If both input matrices are larger than $thres$, both of them will be stored in a distributed way and executed with the $CRMM$ strategy.

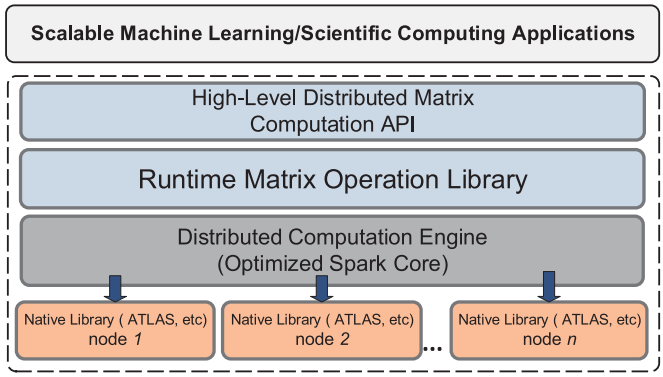


Fig. 8. System architecture.

5 System Design and Interface

We have implemented Marlin as a library along with a set of related matrix operations on top of the Spark platform. Fig. 8 illustrates the overview of system architecture, including three layers: distributed matrix representation and APIs, the Marlin runtime and the optimized Spark core. In this section, we present the matrix computation representation and APIs in Marlin.

A distributed matrix can be represented as either row-matrix or block-matrix representation (the left part of Fig. 2) [5]. These two representations are used in different scenarios. Row-matrix is usually adopted to represent the raw-based input files, e.g., labeled text training data, and each row vector may represent a sample instance of training data. Block-matrix is usually used to represent the intermediate and result matrices of distributed matrix computation. Given a blocking strategy (see the next part), the representation of a matrix can be transformed to a new representation required for computation efficiency (the middle part of Fig. 2). In Spark MLlib and SystemML, the matrix representations are backed by the Spark RDD abstraction, the element type of which is a key-value pair. Marlin also adopts the same way.

Marlin defines a set of high-level APIs that cover basic matrix computing operations, including matrix creation, indexing, slicing operations, and some other advanced operations such as the LU decomposition, the Choleskly decomposition and matrix inverse. In addition to these matrix-based operations, Marlin also provides some operations related to Spark to better utilize the underlying computing platform, as well as APIs to load input data and to save output data. An overview of these APIs can be found in Table 2. Compared with Spark MLlib, Marlin currently provides more matrix APIs that enable user to easily implement scalable complicated machine learning and data mining algorithms or applications.

6 Evaluation

We evaluated Marlin through a series of experiments on a physical cluster with 21 nodes using diversified and representative large-scale matrix computation benchmarks

TABLE 2
Matrix Operation APIs of Marlin

Category	APIs

Matrix Creation	`onesDenVecMatrix(sc, rows, columns)`
	`randomDenVecMatrix(sc, rows, columns)`
	`loadMatrixFile(sc, path)`

Element-wise Operations	`C = A + b`
	`C = A .* B`
	`C = A * b`
	`C = A / b`

Matrix-Matrix Operations	`C = A * B` `C = A + B`
Matrix Layout	`B = A.getSubMatrix(startRow, endRow, startCol, endCol)` `B = A.t`

Other High Level Operations	`B = A.luDecompose()`
	`B = A.choleskyDecompose()`
	`B = A.inverse()`
	`B = A.svd()`

Matrix Storage	`A.saveToFileSystem(path)` `A.saveSequenceFile(path)`

Spark related Operations	`A.partitionBy(partitioner)`
	`A.toDataFrame()`
	`A.cache()`

(see Section 6.1). For comparison, we also evaluate the performance of SystemML, Spark MLlib and SciDB under the same circumstances. Besides that, we compare the performance and memory usage of Marlin with SUMMA [23] in ScaLAPACK [24] which is widely used in high performance computing area. The experimental results are highlighted as follows:

1) Compared with RMM or $CPMM$, the $CRMM$ strategy always achieves the best performance over various matrix multiplication benchmarks (Section 6.2.1).
2) All the system-level optimizations, including *BCNL*, *SMT*, *LSMG* and adaptive strategy selection proposed in Marlin are effective for performance improvement (Section 6.2.2).
3) For large-sized matrix multiplication, Marlin outperforms Spark MLlib, SystemML and SciDB with about $1.29\times$, $3.53\times$ and $2.21\times$ speedup, respectively. For multiplication of large matrix by small matrix, Marlin achieves $2.68\times$, $1.72\times$ and $11.2\times$ speedup (Section 6.2.3) against them respectively.
4) For multiplication of large matrix by small matrix, Marlin achieves $1.15\times$ speedup against MPI. For two large matrix multiplication, MPI outperforms Marlin with $1.10\times$ speedup. For two large matrices with large common dimension, Marlin outperforms MPI with about $5.22\times$ speedup (Section 6.2.4).
5) Marlin has both near-linear data scalability and node scalability (Section 6.3).
6) Evaluation on a realistic DNN workload indicates that Marlin outperforms Spark MLlib, SystemML and SciDB about $12.8\times$, $5.1\times$ and $27.2\times$ speedup, respectively (Section 6.4).

6.1 Experiment Setup

Hardware. All the experiments are conducted on a physical cluster with 21 nodes. Among them, one is reserved to work as the master and all the other 20 nodes work as the workers in Spark. Each node has two Xeon Quad 2.4 GHz processors with 16 logical cores in total, 64 GB memory and two 1 TB 7200 RPM SATA hard disks. The nodes are connected with the 1 Gb/s Ethernet.

Software. All the nodes run on Ext3 file system and the Redhat 6 operating system with JDK 1.7 installed. The version of the underlying Apache Spark used in Marlin is 2.0.2. MLlib carried by Spark 2.0.2 is also installed for performance comparison. The version of SystemML is 0.9, and the version of SciDB is 14.8. We configure the 45 GB memory size of each executor on the worker. The native linear algebra library installed on each local node is BLAS 3.2.1 in ATLAS version. The SUMMA in ScaLAPACK is implemented by OpenMPI 1.8.8.

Benchmarks. Each group of experiments are conducted over various sizes of matrices for different cases. Matrix multiplication needs two input matrices, thus in notation we represent a test case as $m \times k \times n$, which means the sizes of the matrix A and B are $m \times k$ and $k \times n$ respectively and k is called as common dimension. For example, matrix A sized $100 \times 10,000$ multiplying matrix B sized $10,000 \times 2,000,000$ is represented as $100 \times 10K \times 2M$.

6.2 Performance Analysis

6.2.1 Matrix Multiplication Execution Strategies

In this section, we evaluate the execution performance of the execution strategies for large-scale matrix computation in Marlin. They are the $CPMM$, RMM, and $CRMM$ strategies discussed in this paper. The execution graphs are shown in Fig. 3a, 3b, 5 respectively. The experiments are conducted on the same cluster environment with diversified and representative benchmark cases. The matrices in each experiment case are well-blocked according to Section 3, and the behavior of each strategy is also in agreement with our previous experiments on 12-node cluster. Case 1 to case 3 stand for cases with large common dimension k, while case 4 to case 6 stand for general cases with three large dimensions. The last three cases stand for special cases with two large dimensions m and n. The experimental results are shown in Fig. 9.

Cases with Common Large Dimension $\boldsymbol{k}$. In case 1 to case 3, the execution graphs of $CRMM$ and $CPMM$ are actually the same. It can be seen that they both perform well in these special cases, while RMM performs extremely poor. This is because that to avoid another shuffle, RMM does not exploit the execution concurrency of the k dimension, the task execution parallelism of RMM is only $M_b \times N_b$. Moreover, each task in RMM involves K_b times sub-matrix multiplication which are executed locally. In case 1-3, M_b and N_b are small, while K_b is large. Thus, the execution concurrency of RMM is very low, while sub-matrix multiplication workload inside each task is very heavy.

Cases with Three Large Dimensions. In the general cases from 4 to 6, it can be seen that the $CRMM$ strategy always achieves better performance than the other two strategies in these general cases. It achieves about $1.29\times$ speedup against RMM and $3.45\times$ speedup against $CPMM$. This attributes to

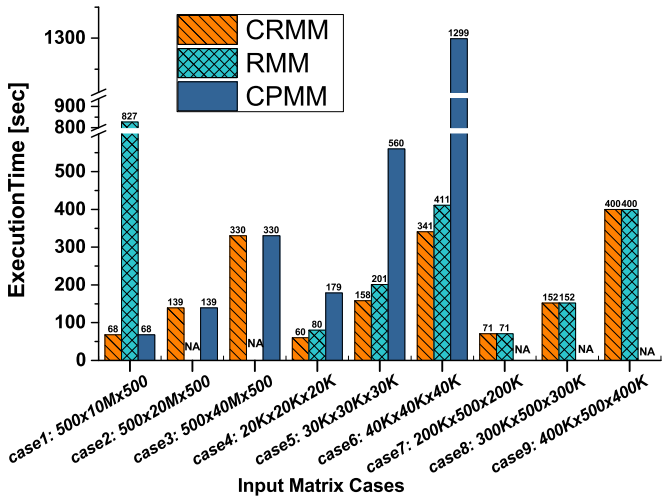


Fig. 9. Performance comparison of various execution strategies (input representation is block-matrix).

the three-dimension concurrency $Con = M_b \times K_b \times N_b$ for $CRMM$. It balances the load of the shuffle, computation and aggregation well.

Cases with Two Large Dimensions m and n. In the cases from 7 to 9 , both $CRMM$ and RMM both perform well while $CPMM$ performs extremely poor. This is because $CPMM$ generates too large intermediate matrices during the second shuffle phase, which cannot be handled properly in the cluster.

6.2.2 Evaluation of Optimization Measures

In this section, we evaluate the effectiveness of the proposed system-level optimization methods in Marlin, including Batch Calling Native Library ($BCNL$) optimization, Slicing Matrix Transformation (SMT) and Shuffle-Light sub-Matrix co-Grouping ($LSMG$).

BCNL optimization. This optimization merges many vectors into a local matrix. Then, this matrix multiplies the broadcasted matrix by calling BLAS-3 native library. Therefore, this optimization is designed for the $BroadcastMM$ case which is a small matrix multiplying a large matrix. In $BroadcastMM$ the large matrix does not need to be blocked, thus the representation of the input matrices is row-matrix which is usually loaded from the row-based text files. The execution time in Table 3 shows that the $BCNL$ optimization can speed up the $BroadcastMM$ around 3 times. Also, the optimization works more effective for cases of larger matrices. This is because that compared with the naive implementation, $BCNL$ can reduce the overheads of calling native library. In conclusion, the $BCNL$ optimization method takes effect.

TABLE 3
Evaluation of Batch Calling Native Library (BCNL) Optimization (Metric is Execution Time in Second, Input Representation is Row-Matrix)

Matrix Size	$BroadcastMM$	$BroadcastMM$ with $BCNL$	Speedup
$500K \times 1K \times 1K$	10	8	1.25
$500K \times 10K \times 1K$	84	24	3.50
$1M \times 1K \times 1K$	14	9	1.56
$1M \times 10K \times 1K$	157	41	3.83
$5M \times 1K \times 1K$	48	21	2.29
$5M \times 10K \times 1K$	727	197	3.69

TABLE 4
Evaluation of Slicing Matrix Transformation (SMT) and Shuffle-Light sub-Matrix co-Grouping (LSMG) Optimizations (Metric is Execution Time in Second, NA Means Failed to Finish in 2,500 Seconds; Input Representation is Row-Matrix)

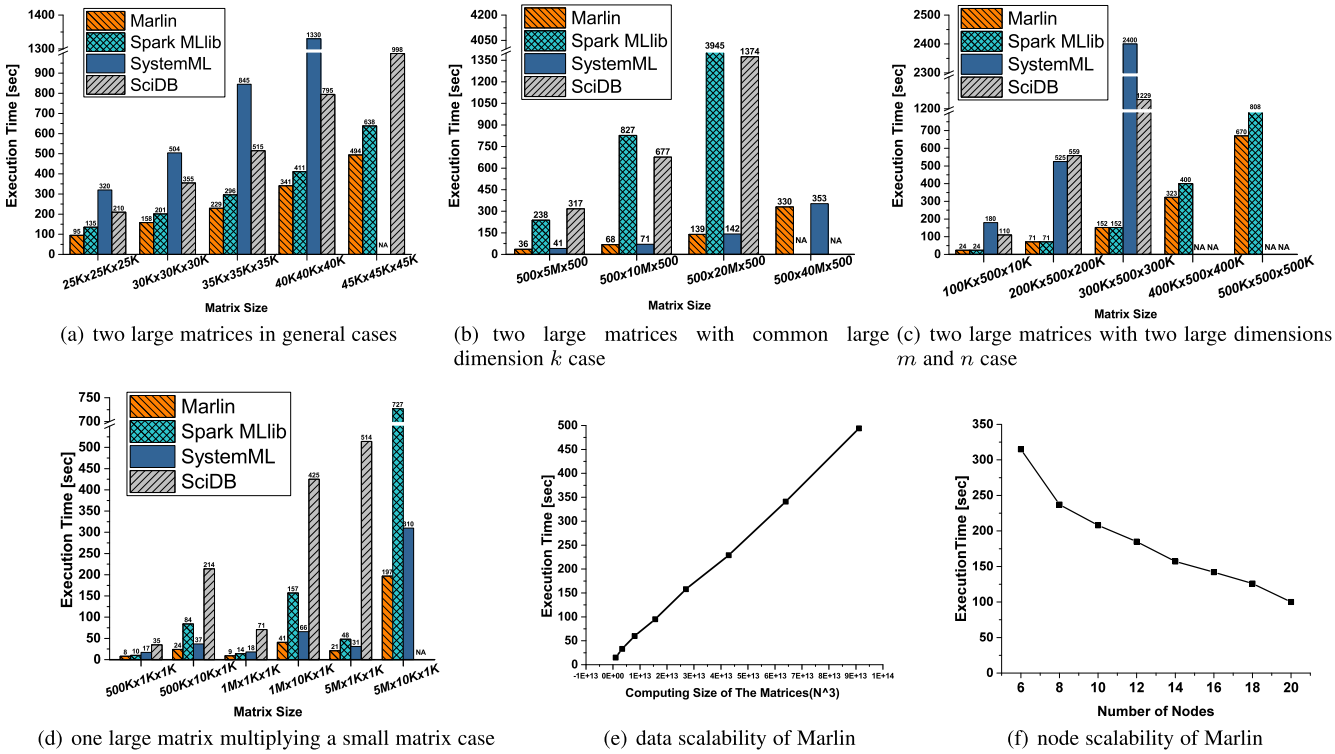
Matrix Size	$CRMM$	$CRMM$ with SMT	$CRMM$ with SMT&LSMG	Spark MLlib
$20K \times 20K \times 20K$	140	88	80	402
$25K \times 25K \times 25K$	339	139	131	926
$30K \times 30K \times 30K$	611	215	206	1702
$35K \times 35K \times 35K$	980	339	310	NA
$40K \times 40K \times 40K$	1881	634	453	NA

SMT and LSMG Optimizations. The SMT optimization aims to efficiently transform row-matrix into block-matrix. And, based on this, $LSMG$ aims at improving sub-matrices co-grouping efficiency. In addition, we also compare the performance of Marlin with Spark MLlib which also supports row-matrix format representation. Experimental results presented in Table 4 prove the effectiveness of the SMT and $LSMG$ optimizations. Besides, due to lacking efficient transformation from row-matrix to block-matrix, Spark MLlib has much poorer performance compared to Marlin. Marlin outperforms Spark MLlib with about $6.78\times$ speedup in these cases.

Adaptive Strategy Selection Optimization. The goal of the adaptive strategy selection proposed in Section 4.4 is to choose the appropriate execution strategy among $LocalMM$, $BroadcastMM$ and $CRMM$ according to the sizes of input matrices. Table 5 demonstrates the performance of three execution strategies along with the execution strategy selected by Algorithm 1 under various cases. The empirical $thres$ is set to 12,000,000 in our environment. For fair comparison, the corresponding optimizations of each execution strategy are used in this context. As shown in Table 5, when dimension sizes of both input matrices are smaller than $thres$, the $LocalMM$ achieves the best performance. When the dimension size of one input matrix is smaller than $thres$ and the dimension size of the other input matrix is larger than $thres$, the $BroadcastMM$ runs fastest. When the dimension sizes of both two matrices are larger than $thres$, the $CRMM$ strategy runs with least time cost. The experimental results show that the proposed adaptive selection strategy chooses the best execution strategy in all input cases above.

TABLE 5
Verification of Adaptive Strategy Selection Optimization (Metric is Execution Time in Second)

Matrix Size	$Local$	$Broadcast$ MM	$CRMM$	Adaptive Selected Strategy
$1K \times 1K \times 1K$	1.4	6.0	2.3	$LocalMM$
$2K \times 2K \times 2K$	2.3	6.7	2.8	$LocalMM$
$500K \times 1K \times 1K$	185	8	40	$BroadcastMM$
$500K \times 10K \times 1K$	2079	22	208	$BroadcastMM$
$10K \times 10K \times 10K$	297	53	15	$CRMM$
$20K \times 20K \times 20K$	2227	207	60	$CRMM$



(a) two large matrices in general cases

(b) two large matrices with common large dimension k case

(c) two large matrices with two large dimensions m and n case

(d) one large matrix multiplying a small matrix case

(e) data scalability of Marlin

(f) node scalability of Marlin

Fig. 10. Performance comparison among different systems and scalability performance of Marlin (input representation is block-matrix).

6.2.3 Performance Comparison with Other Matrix Computation Platforms

In this section, we compare matrix multiplication performance of Marlin, Spark MLlib, SystemML and SciDB. Both Marlin and Spark MLlib support using the native library, while SystemML does not use the native library but has implemented an efficient built-in linear algebra library itself. SciDB encapsulates ScaLAPACK as its underlying distributed linear algebra library.

Large-Scale Matrix Multiplying Small Matrix. All Spark-based systems adopt the $BroadcastMM$ execution strategy that broadcasts the small matrix to each computing node in the cluster. Moreover, Marlin adopts the optimization proposed in Section 4.1 to promote the BLAS-2 level computation to the BLAS-3 level. The experimental results in Fig. 10d show that naively calling the native library to execute the BLAS-2 level computation frequently, which is the same way in Spark MLlib, will decrease the performance due to incurring JNI data copy overhead. Experimental results demonstrate that Marlin outperforms Spark MLlib with about $2.68\times$ speedup and SystemML with about $1.72\times$ speedup. SciDB is too slow and not comparable in these cases. In addition to the overhead and data partition, SciDB also adopts SUMMA [23] which does not perform matrix multiplication well in general clusters [18]. As a result, Marlin outperforms SciDB about $11.2\times$ speedup.

Large-Scale Matrix Multiplying Large-Scale Matrix. The results are shown in Fig. 10a. SystemML performs worst among the three Spark-based systems in the case. Marlin outperforms Spark MLlib with about $1.29\times$ speedup and SystemML with about $3.53\times$ speedup. Also, Marlin outperforms SciDB with about $2.21\times$ speedup. Spark MLlib itself adopts the built-in RMM execution strategy for matrix multiplication. SystemML chooses a special strategy to execute matrix multiplication according to matrix dimensions and cluster resources. Moreover, SystemML has its own linear algebra library implementation instead of calling the native library. SciDB is a complete data management system aiming at supporting scientific computation. It takes the responsibility of data partition and query processing. Before performing matrix operations, SciDB needs to re-distribute the data to each computing node to satisfy the requirement of ScaLAPACK. Moreover, SciDB maintains a failure handling mechanism during the computation, which introduces extra overhead [25].

We also evaluate the performance of two special cases discussed in Section 3. Fig. 10b shows the experimental results of the special cases with one large dimension k. On one hand, similar to the case 1 to 3 in Fig. 9, it can be seen that the performance of the $CPMM$ strategy adopted in SystemML is close to that of Marlin, while the RMM strategy adopted in Spark MLlib performs worst. Similar to the case of one large matrix multiplying a small matrix, SciDB cannot handle one-large-dimension matrix multiplication well and thus behaves much worse than Marlin. On the other hand, Fig. 10c shows the experimental results of the special cases with two large dimension m and n. As discussed in Fig. 9, the RMM and $CRMM$ strategies have the same execution graph for these cases, thus the performance of Marlin is similar to that of Spark MLlib. The $BroadcastMM$ strategy also fits the last two situations in Fig. 10c. Marlin actually adopts the $BroadcastMM$ strategy which performs a little better than the RMM strategy in Spark MLlib. In SystemML, users cannot control the underlying partition number and it chooses a small number of partitions by default to represent each matrix in this case. Using too few partitions results in only a small number of tasks executed in parallel.

TABLE 6
Performance Comparison of Marlin and MPI
(Execution Time: Second, Memory Usage: GB)

Matrix Size		Marlin		MPI Implementation	
		Time	Memory	Time	Memory
Scenario 1	$20K \times 20K \times 20K$	60	9.5	54	0.9
	$30K \times 30K \times 30K$	158	18.1	146	1.3
Scenario 2	$500 \times 10M \times 500$	68	15.8	362	4.7
	$500 \times 20M \times 500$	139	19.3	711	8.9
Scenario 3	$200K \times 500 \times 200K$	71	17.2	67	15.5
	$300K \times 500 \times 300K$	152	33.1	142	34.4
Scenario 4	$500K \times 10K \times 1K$	22	3.8	24	2.6
	$1M \times 10K \times 1K$	37	4.5	45	4.8

Due to the low parallelism, SystemML does not achieve good performance in these cases.

6.2.4 Performance Comparison with MPI Implementation

In this section, we evaluate the performance of matrix multiplication execution time and memory usage between Marlin and the MPI implementation. For the MPI implementation, we directly adopt the SUMMA algorithm [23] provided in ScaLAPACK [24] which is a widely-used MPI-based distributed matrix computation library. SUMMA is a grid-based algorithm which assumes/arranges processors residing on a two- or three-dimensional grid [26]. Each processor only communicates with its neighbors. In high performance computing environment, SUMMA implemented by MPI is one of the most widely-used parallel matrix multiplication algorithm. Unlike the MPI implementation, Marlin is built on distributed data-parallel computing platforms.

The performance comparison of Marlin and the MPI implementation in various matrix multiplication scenarios are presented in Table 6. In execution time, the MPI implementation performs better than Marlin in scenario 1 and 3 with $1.10\times$, $1.06\times$ speedups respectively. In scenario 4, Marlin outperforms MPI by $1.15\times$ speedup. This is because Marlin adopts the $BroadcastMM$ execution strategy for this scenario. Thus, it can avoid transferring the large input matrix across the network. The matrices in scenario 2 are very skew-shaped, Marlin achieves $5.22\times$ speedup against MPI in this scenario. This is because that SUMMA involves a lot of communication cost across processes when dealing with one-large-dimension matrix multiplication [18].

The $Memory$ in Table 6 represents the maximum amount of memory usage in a single processing unit during the execution. The executing process numbers of Marlin and MPI are set to the same in the experiments. Marlin adopts the $CRMM$ strategy in scenario 1 and the $CPMM$ strategy in scenario 2 respectively. In these two scenarios, Marlin uses much more memory than the MPI implementation. This is because that both $CRMM$ and $CPMM$ have an extra shuffle phase to aggregate the intermediate results. Also, different from the MPI implementation implemented in C++ language, Marlin is implemented in Scala. Thus, the overhead of maintaining the data structures in JVM in Marlin also costs extra memory usage.

TABLE 7
Execution Time Per Iteration of Neural Network

Neural network version	Time cost(seconds)
Marlin	3.5
SystemML	18.0
Spark MLlib	44.9
SciDB	95.3

In scenario 3, Marlin adopts the RMM strategy which only contains one shuffle phase as shown in Table 1. In scenario 4, Marlin adopts the $BroadcastMM$ strategy, which does not need to transfer large-scale matrix across the network. Thus, in scenario 3 and 4, Marlin costs similar memory usage as the MPI implementation does.

Overall, Marlin achieves comparable performance to the widely-used MPI-based matrix multiplication algorithm SUMMA. Moreover, Marlin is built on top of Spark, which has better fault tolerance and ease-to-use features than MPI.

6.3 Scalability

Scalability is of great importance to distributed software frameworks. In this section, we evaluate the scalability of Marlin by carrying out two groups of experiments: (1) scaling up the size of the input matrix while fixing the number of nodes, and (2) scaling up the number of nodes while fixing the size of the input matrix. Experimental results of data scalability are shown in Fig. 10e.

Data Scalability. We make an observation that the execution time of Marlin grows near linearly when increasing data size.

Node Scalability. We also conduct a group of experiments to evaluate the node scalability performance of Marlin. The dimensions of the input matrices are fixed at $25{,}000\times 25{,}000 \times 25{,}000$ in these experiments. It can be seen from Fig. 10f that the execution time of Marlin decreases near linearly when increasing the number of cluster nodes. This indicates that Marlin achieves good scalability.

6.4 DNN Application Performance

Finally, we evaluate the performance of training the three-layer neural network model described in Fig. 1. The neural network model contains 300 units in the hidden layer. The MNIST [16] data is adopted as the training dataset which contains 8.1 million samples with 784 features for each sample. We use 1 percent of the total data as the mini-batch of samples to train the model during each iteration. We also try to implement this application with the operation APIs in Spark MLlib. Unfortunately, Spark MLlib still lacks several matrix operation APIs to implement the entire algorithm. Therefore, for these APIs, we add them into MLlib in terms of the design of Marlin. Moreover, the RDD co-partition strategy is applied in both Marlin and the Spark MLlib implementation. In the co-partition strategy, we specify the same partitioner to input feature data and the label data. The partitioner and cache data are preserved during subsequent actions to reduce shuffles. We also implement this three layer neural network application on SystemML based on its DML language and SciDB with its Python interfaces.

As shown in Table 7, for each training iteration, Marlin outperforms Spark MLlib about $12.8\times$ speedup, SystemML

about $5.1\times$ speedup and SciDB about $27.2\times$ speedup on average. This is due to their different matrix multiplication mechanisms. MLlib adopts RMM which gathers all the related sub-matrices on the same node and uses only one task to execute multiplication in this neural network program. SystemML automatically chooses the $BroadcastMM$ strategy according to its strategy selection method, while Marlin automatically selects the $CRMM$ strategy. Another reason for the performance difference between Marlin and Spark MLlib is that MLlib treats the input matrix as sparse matrix. Due to its inefficient transformation from row-matrix to block-matrix, Spark MLlib is not able to leverage the native library for acceleration in the first neural network training layer. Unlike those three Spark-based systems, SciDB is not an in-memory distributed system and thus the DNN application can not benefit much from caching the training input data in memory. As a result, its training speed is much slower than that of the others.

7 Related Work

Distributed Programming Models. The widely-used distributed data-parallel platforms, such as Hadoop [2], Spark [4], Dryad [13], Flink [12] and Piccolo [27], adopt a stage-based or DAG-based execution workflow which is good for fault tolerance and scalability. The programming models they provided to users are MapReduce [1], RDD [4], key-value pair [27], or graphs [28], [29], [30]. These systems lack native and efficient support for linear algebra programming model which is widely-used by data scientists. There are ongoing efforts to provide the Python or R support on top of Hadoop (RHadoop [31]) or Spark (SparkR [32]). However, these language bindings still lack efficient matrix support for distributed data-parallel systems.

Presto [9], extending the R language, presents sparse matrix partitioning to provide load balance. Spartan [33] and DistArray [34] extend the Python language to support distributed arrays. Spartan proposed a distributed array framework programming model. It can automatically determine how to best partition n-dimensional arrays and to co-locate data with computation to maximize locality. These optimizations can also be applied to Marlin.

Matrix Computations. In the HPC environment, distributed matrix computing systems/libraries, such as SUMMA, are usually developed and executed on customized super computers [23]. They are not widely adopted in the big data environment because of the obstacles in fault tolerance, usability, and scalability. In the Big Data era, HAMA [17], Ricardo [35], Mahout [6], MadLINQ [36], SystemML [7], [10], [14], and DMac [25], have proposed general interfaces for large-scale matrix computations. HAMA provides distributed matrix computations on Hadoop and stores matrices on HBase. However, the performance is relatively poor due to the shuffle overhead and disk operations [17]. The largest matrix dimension used in performance evaluation in [17] is only 5000×5000.

Ricardo converts matrix operations to the MapReduce jobs but still inherits the inefficiency of the MapReduce framework. MadLINQ from Microsoft translates LINQ programs into a DAG that is executed on the Dryad platform. It has exploited the extra parallelism using fine-grained pipeline and thus is efficient on demand failure recovery mechanism. Compared to Marlin built on top of Spark, MadLINQ does not exploit efficient memory sharing in the cluster. Moreover, MadLINQ is hard to integrate with the widely-used Hadoop/Spark big data ecosystem stack. SystemML mainly focuses on supporting high-level declarative language. It aims at flexible specification of machine learning algorithms and automatic generation of hybrid runtime execution strategies. To achieve high-level abstraction, SystemML does not support users to tune system-level details such as partitioning and parallelism, while Marlin still supports system-level tuning but focuses on supporting high performance matrix computation. DMac exploits the matrix dependencies in matrix programs for efficient matrix computation in the distributed environment and the optimization can be applicable to Marlin.

The preliminary version of Marlin just naively adopted the Replication-based Matrix Multiplication (RMM) execution strategy [37]. Karsavuran proposes a locality-aware matrix-vector multiplication algorithm but focuses on sparse matrix only [38]. MatrixMap focuses on designing a unified interface to support both algebra operations and irregular matrix operations on data-parallel systems [39]. Its matrix data structure is called Bulk key matrix (BKM) which stores vector-oriented data indexed by key. This data structure has more overhead than the sub-block matrix format in Marlin. Also, MatrixMap does not use the execution concurrency optimization proposed in Marlin.

Reza et al. [40] accelerates the matrix computation that fits the simple idea: separating matrix operations from vector operations and shipping the matrix operations to run on the cluster, while keeping vector operations local to the driver. Matrix computation that can adopt this optimization includes singular value decomposition, Spark port of the popular TFOCS optimization package. However, the approach proposed in [40] also does not study how to parallelize two large-scale matrix multiplication on Spark, because it involves complicated parallel execution strategy and a lot of data exchange during the execution. Marlin proposed in this paper fills this space.

Heterogeneous Matrix Computations Libraries. There are many efficient GPU-based matrix libraries such as cuBLAS/cuSPARSE [41] and BIDMach [42]. Different from them, Marlin is built on top of general purpose distributed data-parallel frameworks, which brings three advantages. First, many existing infrastructures are not equipped with GPUs. In these scenarios, Marlin can be used to accelerate the matrix multiplication applications on homogeneous clusters. Second, many distributed data-parallel platforms, such as Spark, are general platforms for big data processing. Therefore, matrix operations built on the distributed data-parallel platforms can easily be integrated with other components, such as ETL, streaming processing and graph processing, to form an efficient and unified data analytical processing pipeline. Third, the size of the distributed memory in distributed data-parallel platforms can easily scale up to hundreds of TB, while the global memory of GPU is usually less than dozens of GB. Thus, systems like Marlin on distributed data-parallel platforms can also scale up to handle much larger amount of matrix data than GPU-based libraries.

Different from the above libraries built on single GPU, HetroSpark [43] is a GPU-accelerated heterogeneous architecture integrated with Spark. It combines the massive compute power of GPUs and scalability of CPUs and system memory resources for applications that are both data and compute intensive. However, HeteroSpark itself does not study how to efficiently parallelize large-scale matrix multiplication which is carefully studied in our work Marlin.

Matrix Blocking Strategies. There is a significant progress made by the High Performance Computing (HPC) research community towards matrix blocking. SUMMA [23], [26] is widely used in the HPC community. This blocking strategy achieves good performance on the grid or torus-based supercomputers [44] but may not perform well in more general cluster topologies [45]. BFS/DFS-based blocking algorithms [46] view the processor layout as a hierarchy rather than a grid and they are based on sequential recursive algorithms. Among them, CARMA [18] is efficient since it minimizes both memory footprints and communication when blocking matrices in three dimensions. Besides, the equal-split blocking in [17] can be regarded as a special case of CARMA. In data-parallel platforms, the above mentioned matrix blocking strategies also need to work with the underlying data-parallel execution strategy.

8 Conclusion and Future Work

In this paper, we try to improve the performance of large-scale matrix multiplication on distributed data-parallel platforms. To improve execution concurrency, we developed a novel matrix computation execution strategy called *CRMM* to reduce the IO cost. Then, a number of system-level optimizations, including *Batch Calling Native Library (BCNL)*, *Slicing Matrix Transformation (SMT)* and *Light-Shuffle sub-Matrix co-Grouping (LSMG)*, are also proposed. Marlin provides a group of easy-to-use matrix operation APIs on Spark.

Experimental results performed on a cluster with 21 nodes reveals that, compared with existing systems, Marlin can achieve $1.29 - 11.2\times$ speedup for large-sized matrix multiplication and up to $27.2\times$ speedup for a realistic DNN workload. Also Marlin achieves good data and node scalability.

In the future, we plan to explore efficient sparse matrix computation strategies on data-parallel platforms.

Acknowledgments

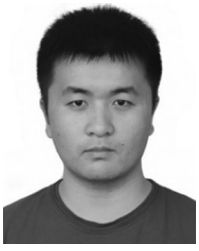
This work is partially supported by the National Natural Science Foundation of China under Grant Numbers 61572250 and 61602194, the Jiangsu Province Industry Support Program under Grant Number BE2014131, and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Yihua Huang is the corresponding author of this paper.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[2] "Apache Hadoop," 2011. [Online]. Available: http://hadoop.apache.org/
[3] "Apache Spark," 2016. [Online]. Available: http://spark.apache.org/
[4] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 1–12.
[5] "Apache Spark MLlib," 2016. [Online]. Available: http://spark.apache.org/mllib/
[6] "Apache Mahout," 2016. [Online]. Available: https://mahout.apache.org/
[7] "SystemML Documents," 2016. [Online]. Available: http://systemml.apache.org/
[8] E. R. Sparks, et al., "Mli: An api for distributed machine learning," in *Proc. IEEE 13th Int. Conf. Data Mining*, 2013, pp. 1187–1192.
[9] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 197–210.
[10] A. Ghoting, et al., "Systemml: Declarative machine learning on MapReduce," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 231–242.
[11] Y. Jia, "Learning semantic image representations at a large scale," Ph.D. dissertation, EECS Dept., Univ. California, Berkeley, CA, USA, May 2014.
[12] "Apache Flink," 2016. [Online]. Available: http://flink.apache.org/
[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. ACM SIGOPS Operating Syst. Rev.*, 2007, pp. 59–72.
[14] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss, "Resource elasticity for large-scale machine learning," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2015, pp. 137–152.
[15] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "Scidb: A database management system for applications with complex analytics," *Comput. Sci. & Eng.*, vol. 15, no. 3, pp. 54–62, 2013.
[16] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST Ddatabaase of handwritten digits," 2015. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[17] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the MapReduce framework," in *Proc. IEEE Second Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 721–726.
[18] J. Demmel, et al., "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 261–272.
[19] "Automatically Tuned Linear Algebra Software (ATLAS)," 2016. [Online]. Available: http://math-atlas.sourceforge.net/
[20] "LAPACK : Linear Algebra PACKage," 2016. [Online]. Available: http://www.netlib.org/lapack/
[21] "Intel Math Kernel Library (Intel MKL)," 2016. [Online]. Available: https://software.intel.com/en-us/intel-mkl
[22] A. Davidson and A. Or, "Optimizing shuffle performance in spark," (2013). [Online]. Available: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project16_report.pdf
[23] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency-Practice Experience*, vol. 9, no. 4, pp. 255–274, 1997.
[24] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *Proc. 4th IEEE Symp. Frontiers Massively Parallel Comput.*, 1992, pp. 120–127.
[25] L. Yu, Y. Shao, and B. Cui, "Exploiting matrix dependency for efficient distributed matrix computation," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2015, pp. 93–105.
[26] W. F. McColl and A. Tiskin, "Memory-efficient matrix multiplication in the BSP model," *Algorithmica*, vol. 24, no. 3/4, pp. 287–297, 1999.
[27] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 1–14.
[28] Y. Low, et al., "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.
[29] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[31] R. Analytics, "Rhadoop: A collection of five R packages that allow users to manage and analyze data with Hadoop," 2012. [Online]. Available: https://github.com/RevolutionAnalytics/RHadoop
[32] AMPLab, "SparkR (R on Spark)," 2016. [Online]. Available: https://spark.apache.org/docs/latest/sparkr.html
[33] C.-C. Huang, et al., "Spartan: A distributed array framework with smart tiling," in *Proc. 2015 USENIX Annu. Tech. Conf.*, 2015, pp. 1–15.
[34] "Distarray documents," 2016. [Online]. Available: http://docs.enthought.com/distarray/
[35] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson , "Ricardo: integrating r and hadoop," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 987–998.
[36] Z. Qian et al., "Madlinq: Large-scale distributed matrix computation for the cloud," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 197–210.
[37] R. Gu, et al., "Efficient large scale distributed matrix computation with spark," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 2327–2336.
[38] M. O. Karsavuran, K. Akbudak, and C. Aykanat, "Locality-aware parallel sparse matrix-vector and matrix-transpose-vector multiplication on many-core processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1713–1726, Jun. 2016.
[39] Y. Huangfu, J. Cao, H. Lu, and G. Liang, "Matrixmap: Programming abstraction and implementation of matrix computation for big data applications," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst.*, 2015, pp. 19–28.
[40] R. Bosagh Zadeh, et al., "Matrix computations and optimization in apache spark," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 31–38.
[41] "cuBLAS," 2016. [Online]. Available: https://developer.nvidia.com/cublas
[42] J. Canny and H. Zhao, "Bidmach: Large-scale learning with zero memory allocation," in *Proc. BigLearning, NIPS Workshop*, 2013, pp. 1–8.
[43] P. Li, Y. Luo, N. Zhang, and Y. Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2015, pp. 347–348.
[44] E. Solomonik and J. Demmel, "Matrix multiplication on multidimensional torus networks," in *Proc. High Perform. Comput. Comput. Sci.*, 2013, pp. 201–215.
[45] J. Demmel et al., "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 261–272.
[46] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proc. 24th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2012, pp. 193–204.

Rong Gu received the PhD degree in computer science from Nanjing University, in Dec. 2016. He is an assistant researcher with State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include parallel computing, distributed systems and distributed machine learning. He was a student member of the IEEE.



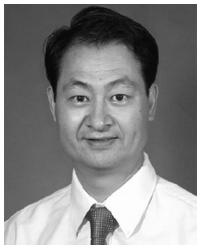
Yun Tang received the BS degree in Nanjing University, China, in 2013. He is currently working towards the MS degree in Nanjing University. His research interests include distributed dataflow system and cloud computing. He is a student member of the IEEE.

Chen Tian received the BS, MS, and PhD degrees from Department of Electronics and Information Engineering from Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is an associate professor with State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming and urban computing.

Hucheng Zhou received the PhD degrees from Department of Computer Science and Technology of Tsinghua University, 2011. He is a researcher in Microsoft Research Asia. His research interests span multiple areas including distributed systems, data-parallel computing, large-scale machine learning, mobile computing, program analysis and compiler optimization, etc.

Guanru Li is a software engineer with Microsoft Search Technology Center. He joined Microsoft in 2015, after graduated from Shanghai Jiaotong University with a BS degree. His current work includes providing Spark as a service, developing machine learning and streaming applications on Spark.

Xudong Zheng received the MS degree from Beihang University. He is a software develop engineer from Microsoft Search Technology Center. He worked on various projects for large scale machine learning and data mining, which includs optimization and adoption of Spark for some applications in Microsoft. He joined Microsoft at 2013.



Yihua Huang received the bachelor's, master's, and PhD degrees in computer science from Nanjing University. He is currently a professor in computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. His main research interests include parallel and distributed computing, big data parallel processing, distributed machine learning algorithm and system, and Web information mining.