Efficient large scale distributed matrix computation with spark

Conference Paper · October 2015

DOI: 10.1109/BigData.2015.7364023

7 authors, including:

Yihua Huang
Nanjing University
50 PUBLICATIONS **456** CITATIONS

SEE PROFILE

Efficient Large Scale Distributed Matrix Computation with Spark

Rong Gu*, Yun Tang*, Zhaokang Wang*, Shuai Wang*, Xusen Yin†, Chunfeng Yuan*and Yihua Huang*
*National Key Laboratory for Novel Software Technology
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing University, Nanjing, China 210093
Email: {gurong,tangyun,wangzhaokang}@smail.nju.edu.cn,{swang,cfyuan,yhuang}@nju.edu.cn
†Intel Corporation, Beijing, China, 100190
Email: xusen.yin@intel.com

***Abstract*—Matrix computation is the core of many massive data-intensive analytical applications such mining social networks, recommendation systems and nature language processing. Due to the importance of matrix computation, it has been widely studied for many years. In the Big Data ear, as the scale of the matrix grows, traditional single-node matrix computation systems can hardly cope with such large data and computation. Existing distributed matrix computation solutions are still not efficient enough, or have poor fault tolerance and usability. In this paper, we propose Marlin, an efficient distributed matrix computation library which is built on top of Spark. Marlin contains several distributed matrix operation algorithms and provides high-level matrix computation primitives for users. In Marlin, we proposed three distributed matrix multiplication algorithms for different situations. Based on this, we designed an adaptive model to choose the best approach for different problems. Moreover, to improve the computation performance, instead of naively using Spark, we put forward some optimizations including taking advantage of the native linear algebra library, reducing shuffle communication and increasing parallelism. Experimental results show that Marlin is over an order of magnitude faster than R (a widely-used statistical computing system) and the existing distributed matrix operation algorithms based on MapReduce. Moreover, Marlin achieves comparable performance to the specialized MPI-based matrix multiplication algorithm SUMMA but uses a general dataflow engine and gains common dataflow features such as scalability and fault tolerance.**

***Keywords*-matrix computation; parallel algorithm; in-memory computing; Spark**

I. Introduction

Due to the key role in a wide range of data-intensive scientific applications, such as large-scale numerical analysis and computational physics, matrix computation is considered as an important part of constructing computational platforms for such problems [1]. In Big Data ear, many real-world applications, such as social media analytics, web-search, computational advertising and recommender systems, have created an increasing demand for scalable implementations of matrix computation on massive datasets [2][3][4]. As the rapid growth of the above applications, there has been a growing need for scalable and efficient matrix computation, which can hardly be handled by the single-node matrix computation libraries due to hardware resource limitation.

There is a significant progress made by the research community towards efficient distributed matrix computation. It started with early works on the realm of High Performance Computing (HPC), including the well-known ScaLAPACK [5] solution. In recent years, as MapReduce [6] is becoming a generic parallel programming model, there also have been attempts to implement matrix operations on MapReduce framework (e.g. HAMA [7]). However, this solution is not efficient because MapReduce has a lot of overheads and does not take advantage of the distributed memory [8] well. Microsoft also develops a system MadLINQ [8] in MapReduce-like framework. Unlike Hadoop MapReduce, it has high expressiveness for experimental algorithms.

In this paper, we propose Marlin [1], an efficient distributed matrix computation library built on top of Spark [9] which is an distributed in-memory cluster computing framework. Marlin contains several distributed matrix operations and especially focuses on matrix multiplication which is a fundamental kernel of high performance scientific computing. We classify the matrix multiplication problems into three categories. Correspondingly, three distributed matrix multiplication algorithms, named *Marlin-Blocking*, *Marlin-CARMA* and *Marlin-Broadcast*, are proposed respectively. Based on this, an adaptive model is proposed to select the best matrix multiplication approach for problems of different characteristics. Further, we put forward some optimization methods to improve the performance rather than simply adopting Spark. The optimizations include taking advantage of single-node native libraries for local numerical computation, reducing shuffle communication and increasing degree of parallelism. Experimental results show that Marlin achieves significant speedups over the distributed matrix computation algorithms based on MapReduce [6] and the widely-used single-node statistical software, R [10]. Also, Marlin is comparable to or faster than SUMMA, the widely-used MPI-based matrix multiplication library. Benefitting from the general-purpose dataflow of Spark, Marlin achieves good scalability, fault tolerance and ease of usage features.

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

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

II. PRELIMINARY

A. Parallel Matrix Multiplication Algorithms

Grid-based approach - The grid-based algorithms [11], [12] regard processors as residing on a two- or three-dimensional grid. Each processor only communicates with its neighbors. In traditional distributed computing model, such as MPI, SUMMA [13] is the most widely-used parallel matrix multiplication algorithm. The multiplication computation in SUMMA is iterative with several rounds. In each iteration, each processor broadcasts the blocks of matrix A it owns to the whole processor row, and broadcasts the blocks of matrix B to the whole processor column in parallel, after received these blocks, each processor then executes multiplication once. The size of the submatrix is set by users, which means this algorithm has opportunities for tuning. Grid-based algorithms achieve high performance when they are implemented to the grid or torus-based topologies in many modern supercomputers [14]. However they may not perform well in more general topologies [15].

BFS/DFS approach - Besides the grid-based algorithms, the BFS/DFS approaches [16], [17] are also proposed to parallelize the matrix multiplication algorithm. BFS/DFS-based algorithms view the processor layout as a hierarchy rather than a grid and they are based on sequential recursive algorithms. Among BFS/DFS-based algorithms, CARMA [15] is the one minimizing communication for all matrix dimensions and memory footprints, which applies the BFS/DFS approach to the dimension-splitting recursive algorithm. During each recursive step, the largest dimension of the three is split by half, yielding two sub-problems. These sub-problems are solved in either a BFS step or a DFS step, depending on the available memory. Demmel J et al. have proved that SUMMA is only communication-optimal for certain matrix dimensions, while CARMA can minimize communication for all matrix dimensions cases [15].

Data-Parallel approach - In recent years, researchers have proposed HAMA [7], a distributed matrix multiplication systems built on top of the MapReduce data parallel computing framework. There basically exist two parallel matrix multiplication approaches: iterative approach and block approach. In the iterative approach, each map task receives a row index of matrix B as a key, and the column vector of the row as a value. Then, it multiplies all columns of $i\text{-}th$ row of matrix A with the received column vector. Finally, a reduce task collects the $i\text{-}th$ product into the result matrix. The block approach contains two MapReduce jobs with a parallelism parameter called block number b. In the first MapReduce job, the input matrices are split into $b \times b$ small submatrices in a 1-D representation, called $collectionTable$. Each row of the $collectionTable$ has two submatrices of $A(i,k)$ and $B(k,j)$. In the second MapReduce job, the results are collected from the $collectionTable$. During the job, each map task works only on the $collectionTable$ to reduce the data movement over the network.

B. Spark Distributed Computing Framework

Spark [9] is an efficient distributed cluster computing framework for fast big data processing. The fundamental feature is its in-memory parallel execution model. Storing datasets in the aggregate memory of a cluster is good for efficient processing iterative algorithms. The second key feature is that, deferring from the fixed two-stage data flow model in MapReduce, Spark provides very flexible DAG-based (directed acyclic graph based) data flows.

For the programming model, Spark offers the Resilient Distributed Datasets (RDDs), which is partitioned and immutable collection of data items that allow for low-overhead fault-tolerance without requiring checkpointing and rollbacks. RDD equips with a variety of built-in operations to transform one RDD to another one. Spark can cache the RDDs in memory across the worker nodes, making data reuse faster. RDDs achieve fault-tolerance based on the lineage information which is tracked for re-constructing any lost partitions of the RDD when a worker node fails.

Besides RDD, Spark additionally provides another important immutable abstraction, *Broadcast*. Broadcast variables are initialized at the driver node and then shared to the worker nodes. Spark ensures that each broadcast variable is copied to each worker only once instead of packaging it with every closure. Usually, broadcast variables are made available at all the worker nodes at the start of any parallel RDD operations (like *map* or *foreach*) by using a suitable data distribution mechanism. Several topology-aware network-efficient algorithms for efficiently distributing the Spark's broadcast variables are proposed in [18].

III. RELATED WORK

There is a significant progress made by the research community towards parallel matrix multiplication, which results in many mature distributed matrix multiplication libraries readily available. Among them, SUMMA has been widely used in the large scale matrix multiplication applications. However, this algorithm is only optimal in certain memory ranges. Therefore, the '2.5D' algorithms [11], [12] as well as BFS/DFS-based algorithm, such as CARMA, are proposed to solve the rest scope of the problem.

Previously, many research efforts are devoted to designing and developing efficient parallel matrix multiplication algorithms on traditional distributed frameworks, e.g. MPI. These solutions always require that the matrices and the intermediate results can be resided at the memory of the cluster. The disadvantage of these solutions is obvious that they cannot work when the data cannot be kept in memory totally or the users want to reserve memory for other applications, which is quite common in a shared cluster environment. Also, these methods need to design complicated fault tolerance mechanisms case by case. In other word, for the pursuance

of high performance, these solutions adopt lower level primitives, which severely affect programmability, scalability and robustness as a result. To solve the low programmability of traditional distributed computing environments, pbdR [19] tightly couples R [10] with the ScaLAPACK [5] and MPI libraries, which enables developing high-level distributed data parallelism in R and also utilizing HPC platforms, but still suffers the fault tolerance problems.

In recent years, systems like MapReduce gain great success due to their good usability, scalability and fault-tolerance features. In the case of parallel matrix computation, HAMA [7], built on Hadoop with representing and storing matrices on HBase, provides distributed matrix computations. However, the execution performance of HAMA is not efficient due to the overhead and disk operations of the MapReduce jobs. In paper [7], HAMA just compares its performance to the MPI implementation of iterative approach and the largest matrix dimension used is 5000 by 5000 in the evaluation, which is not large compared to the related work on MPI. On the other side, Microsoft develops MadLINQ [8] which translates LINQ programs into a DAG set of parallel operators that can be executed on the Microsoft Dryad platform. However, this approach does not exploit efficient memory sharing in the cloud.

IV. Overview of System Design

In this section, we first describe the system stack of Marlin and the related systems it interacts with. Then, we give a brief introduction to the typical features of Marlin.

A. System Stack

As illustrated in Figure 1, Marlin is built on top of Spark and provides high-level matrix computation interfaces to big data applications. Marlin adopts Spark to distribute the data and schedule the computing tasks to executors in the cluster. For each task, the executor uses Breeze, a single-node low-level linear algebra library, to conduct the CPU-intensive matrix computation. Breeze provides Scala APIs to Marlin through Spark, and it calls C/Fortran-implemented native library, such as BLAS, to execute the linear algebra computation through the Java Native Interface (JNI).

B. Features

1) Native Linear Algebra Library Acceleration: Matrix computation is a typical computation-intensive task which has been researched well. There exist a lot of single-node high performance linear algebra libraries, such as BLAS, Lapack and MKL. Basically, Marlin takes a divide-and-conquer strategy to deal with the large scale matrix computation. Each computation of the divided submatrix is executed on a single node. Instead of performing linear algebra computations on JVM which has low performance for this, Marlin offloads the CPU-intensive operation from JVM to the native linear algebra library (e.g. BLAS, Lapack) by JNILoader, which speeds up the performance significantly.

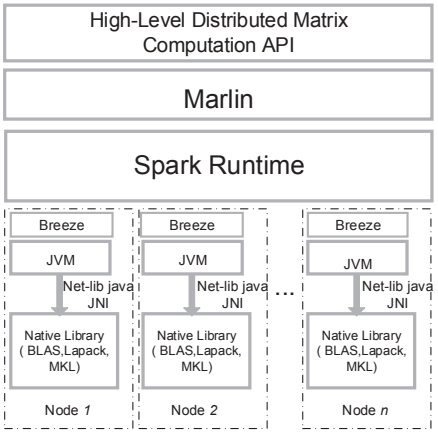

Figure 1. The system stack of Marlin and its related systems

2) Fine-grained Fault Tolerance and Ease to Use: Marlin can be regarded as an application built on top of Spark, thus it also achieves the fine-grained fault tolerance which is extended from Spark. Additionally, different from the HPC libraries, Marlin offers developers with high level matrix computation interfaces in Scala/Java which can accelerate the development of big data applications on top of it. Moreover, users can use it interactively from the Scala shells.

3) Efficient Distributed Matrix Operations: The Marlin library contains quite a few distributed matrix computing operations. Besides the large scale matrix multiplication, it also has some other matrix operations, such as matrix-matrix addition, subtraction, element wise multiplication, addition, get submatrix and so on. These simple matrix operations are easy to be implemented on Spark and can achieve high efficiency, as involving little network communication. Designing and implementing efficient distributed matrix multiplication is complex and the rest part of this paper discusses this problem in detail.

V. Matrix Multiplication On Spark

A. Representing Large Scale Matrices on Spark RDD

Before designing the distributed matrix multiplication algorithms, we firstly introduce the distributed matrix representation on the Spark RDD abstract.

As illustrated in Figure 2, distributed matrices in Marlin can be expressed in three types, *Block Matrix*, *Dense/Sparse Vector Matrix* and *Coordinate Matrix* respectively. Each representation can be transformed to another easily.

A Block Matrix is represented as a key-value pair with the key denoting the block id and the value carrying the sub-matrix in the block. Each key-value here is an element of a RDD. A blockID is basically a wrapper over (block row, block column). And this type has superiority when used in the block-based matrix multiplication.

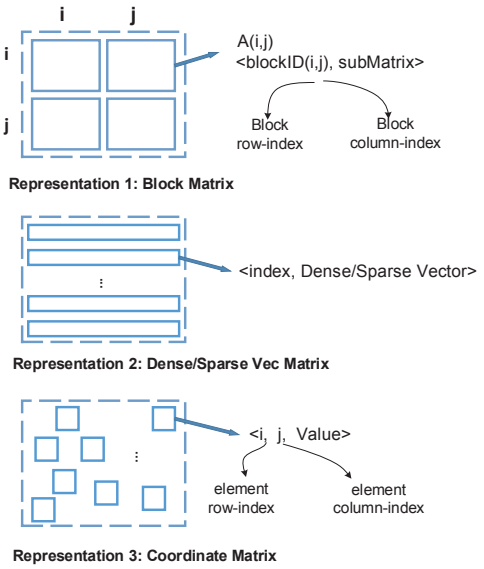

Figure 2. The representation of Matrix on Spark RDD

A Dense/Sparse Vector Matrix is a row-oriented distributed matrix with row indices, expressed as a key-value pair: (rowIndex, Dense/Sparse Vector). Also, each key-value here is an element of a RDD. Referred to MLlib[2] , Marlin also supports two types of local vector: DenseVector and SparseVector. A dense vector is essentially a double array representing its entry values, while a sparse vector is stored by two arrays: indexes and values. For example, a vector (1.0,0.0,4.0) can be represented in dense format as [1.0,0.0,4.0] or in sparse format as (3,[0,2],[1.0,4.0]), where 3 is the length of the vector. This matrix representation is a basic data type, especially in data mining, where a row-vector in matrix has an actual meaning as a sample instance.

A Coordinate Matrix is a distributed matrix backed by an RDD of its entries. Each entry is a tuple of $(i : \text{Long}, j : \text{Long}, value : \text{Double})$, where i is the row index, j is the column index, and $value$ is the entry value. This matrix representation has superiority when the matrix is very sparse.

After introducing the matrix representations, we describe the three distributed matrix multiplication algorithms adopted in Marlin in the following subsections.

B. Approach 1: Block-splitting matrix multiplication

The first approach is called block-splitting matrix multiplication, which splits two input matrices into blocked matrices and executes the submatrix multiplications in parallel. As described in Figure 3, the workflow of this algorithm can be divided into 6 steps. In *splitting* step, each input matrix has been preceded to $b \times b$ blocks, and in *flatMap* step each block

[2]MLlib is a scalable machine learning library built on top of Spark.http://spark.apache.org/mllib/

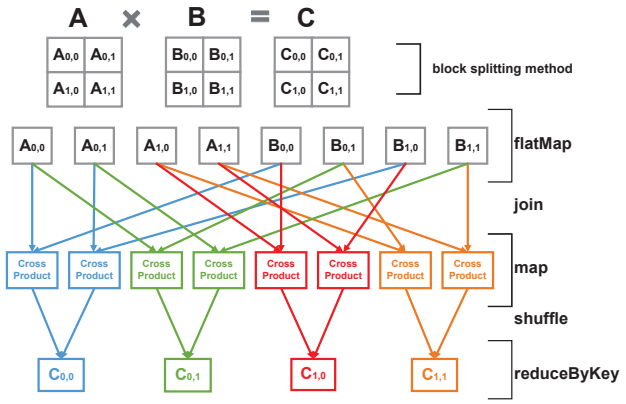

Figure 3. The workflow of block-splitting approach matrix multiplication on Spark programming model

then emits b times, which need write to disk b times in Spark programming model. In *join* step, b^3 blocks of matrix B, the output of previous *flatMap* step, are transferred through the network in order to get related blocks together for the next map step. In *map* step, two related submatrices execute once cross product locally. After the above three steps, there still exists another *shuffle* step to gather related cross product and then sum these partial results together in *reduceByKey* step. This approach is suitable for multiplying two square matrices. And another advantage is that, it offers a parameter called block number, represented as b in the above workflow explanation, to users for tuning the degree of parallelism.

C. Approach 2: CARMA matrix multiplication

When two input matrices are not square, the above dimension-splitting method is no longer very suitable. To solve this problem, we refer to the equal representation of dimension-splitting in BFS steps of CARMA and design a dimension-splitting method similar to CARMA. The splitting method in CARMA is shown in Algorithm 1. The basic workflow of this splitting approach based on Spark is shown in Figure 4. The overall procedure is quite similar to approach 1, but due to the CARMA splitting method, some steps below can be skipped. Before *flatMap* step, matrix A has been generated to $r * s$ blocks while matrix B generated to $s * t$ blocks. On one hand, if $r = t = 1$, which means each block does not need to emit several times and matrix multiplication can be done locally, then the *flatMap* and *join* step can be skipped. This can improve performance greatly, and the condition can be satisfied when the dimension k is extremely large. On the other hand, if $s = 1$, the cross product of *map* step is the final results in matrix C, which means the *shuffle* and *reduceByKey* step can be skipped, resulting in performance improvement. CARMA [15] has been proved to be communication-optimal. We also adopt CARMA algorithm for block splitting in this approach.

Different from the traditional distributed framework, besides the network communication, Spark programs need to write data to disk in shuffle phase when encountering the shuffle dependency during RDD transformation. This phase would greatly influence the overall performance. In subsection E, We will give a detail time complexities analysis.

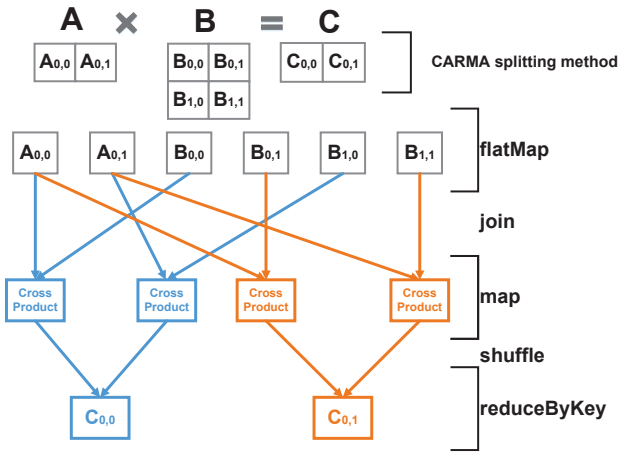

Figure 4. The workflow of CARMA approach matrix multiplication on Spark programming mode, here $r = 1, s = 2, t = 2$

D. Approach 3: Broadcast matrix multiplication

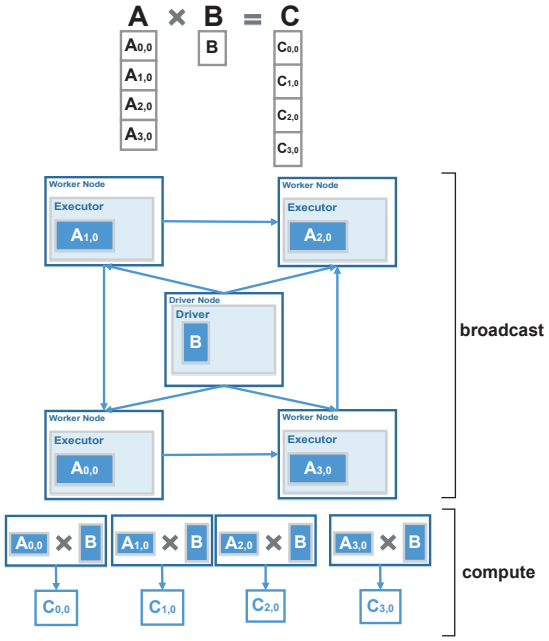

Figure 5. The workflow of broadcast approach matrix multiplication on Spark programming model

If we want to improve the overall performance, we usually need to avoid the shuffle phase as far as possible in Spark. Based on this inspiration, we propose the broadcast approach which reduces the shuffle cost by adopting broadcasting variables. If matrix B is quite small, it will be broadcast to each executor to avoid shuffling the large scale matrix A across network, which can gain great performance improvement. This approach is illustrated in Figure 5. We just divide the whole procedure into two steps, first broadcasting the variable, and then executing the computation in each node's local memory in parallel. The level of parallelism is related to the number of blocks split in matrix A, here we denote it as parameter r. This approach is particular suitable for the cases when one input matrix is not large.

E. Approaches Analysis and Selection

1) Analysis of the Time Complexity: Here we build a brief time cost model to guide the approach selection for different matrix multiplication cases.

Algorithm 1: $CARMA(A, B, C, m, k, n, P)$ the splitting method

Input: A is an $m \times k$ matrix, B is a $k \times n$ matrix, P is related to the real num of cores across the cluster P_{real}.
Output: $C = AB$ is $m \times n$

```
begin
    if P = 1 then
        return SequentialMultiply( A, B, C, m, k, n )
    if n is the largest dimension then
        Parallel do
            CARMA(A, B_left, C_left, m, k, n/2, P/2)
            CARMA(A, B_right, C_right, m, k, n/2,P/2)
    if m is the largest dimension then
        Parallel do
            CARMA(A_top, B, C_top, m/2, k, n, P/2)
            CARMA(A_bot, B, C_bot, m/2, k, n, P/2)
    if k is the largest dimension then
        Parallel do
            CARMA(A_left, B_top, C, m, k/2, n, P/2)
            CARMA(A_right, B_bot, C, m, k/2, n, P/2)
end
```

First, we analyze the time cost models of the CARMA approach and the block-splitting approach since their main procedures are essentially similar. The parameters m, k, n, P used below are defined in Algorithm 1. Given the defined parameter P, dimensions m, k, and n are split into r, s, and t parts respectively, where $P = r * s * t$. As Figure 3 and 4 illustrated, we divide the cost of whole procedure into 5 steps and omit the pre-splitting step.

1) Step 1 is a *flatMap* operator preparing data to disk for the *join* operator. Each block in matrix A will be emit t times, while each block in matrix B will be emit r times, thus the cost can be derived as $IO_{fs}(t|A| + r|B|)$, $|A|$ means the size of matrix A.
2) *Join* operator only shuffles one matrix (A or B) through the network. Assuming only shuffling the matrix B, the cost spending on the network communication can be derived as $Network(r|B|)$. At the same time, the program reads blocks in matrix A locally,

where the cost is much less than the networking cost, and we can omit it in this step.

3) After two related submatrices are gathered together by fetching from the network and reading locally, then the matrix multiplication would be conducted locally. It is clearly that the computing cost of this step is $\mathcal{O}(mnk)$. Actual time consumed in this step is relevant to the basic linear algebra library used, and we can call native library to greatly reduce time cost in this step. According to the split method, if s is not equal to 1, then the approach needs another two steps below to sum up the related submatrices.
4) After submatrix multiplication is done, each result block needs to be written to disks for the next shuffle phase. Similar to step 1, the cost can be derived as $IO_{fs}(s|C|)$. In Spark programming model, steps 2 to 4 are pipelined in one stage.
5) Finally, *reduceByKey* operator fetches the related submatrices through network and performs $s-1$ times addition. Due to some local data, the cost is close to $Network(s|C|)$.

Now, we unify the costs into three stages represented in Spark programming model and transform different cost types (IO cost and network cost) to time cost. Stage 1 consists of the *flatMap* step. We name it *stage-flatMap*, which the serial cost is

$$Cost(stage\text{-}flatMap) = \alpha \cdot IO_{fs}(t|A| + r|B|)$$

α is related to the disk IO speed. Stage 2 contains *join* and *map* step as illustrated in Figure 4. We name it as *stage-multiply*, with the serial cost of

$$\begin{aligned} Cost(stage\text{-}multiply) = q \cdot (\beta \cdot Network(r|B|) \\ + Compute(mnk) + \alpha \cdot IO_{fs}(s|C|)) \end{aligned}$$

And, β is related to the throughput of the network. We define a variable parameter $q = f(r, s, t)$ to represent the overhead of runtime. Generally , if the two submatrices are quite large it may need more garbage collection(GC) time in JVM, while more smaller submatrices need more network and IO time. The last stage has *shuffle* and *reduceByKey* step. We named it as *stage-sum*, as time used in addition is far less than fetching data through the network, the serial cost of this stage is :

$$Cost(stage\text{-}sum) = \beta \cdot Network(s|C|)$$

As the program executes in parallel, to get the overall time, we need to divide the serial time cost with the parallelism. The parallel overall time cost can be generalized as:

$$\begin{aligned} Cost(stage\text{-}flatMap)/\min\{P_{real}, (rs + st)\} \\ + Cost(stage\text{-}multiply)/\min\{P_{real}, rst\} \\ + Cost(stage\text{-}sum)/\min\{P_{real}, rt\} \end{aligned}$$

where P_{real} represents the total number of the physical CPU cores in the cluster.

Specially, when performing two square matrices multiplication, in the blocking-splitting approach, three stage cost can be derived as below, here $r = s = t = b$, $m = n = k$.

$$\begin{aligned} Cost(stage\text{-}flatMap) &= \alpha \cdot IO_{fs}(2bn^2) \\ Cost(stage\text{-}multiply) &= q \cdot (\beta \cdot Network(bn^2) \\ &\quad + Compute(n^3) + \alpha \cdot IO_{fs}(bn^2)) \\ Cost(stage\text{-}sum) &= \beta \cdot Network(bn^2) \end{aligned}$$

The time cost model of broadcast approach is relatively simple. As all data are stored in memory, there are no any disk writing operations. The cost model can be derived as $Broadcast(N \cdot |B|) + q \cdot Compute(mnk)$, here N denotes the nodes number across the cluster, and the parameter q has a functional relationship to r (the number of blocks split in matrix A). If r is too large, which means matrix A is split into too many blocks along the row, the overhead of tasks launching and cleaning cannot be omitted. If r is small, this leads to losing high parallelism and cannot take advantage of reusing the resources in JVM.

2) Adaptive Approaches Selection: Based on the time cost model analyzed above, we put forward a heuristic algorithm for selecting the appropriate matrix multiplication approach when given two distributed matrices. As shown in Algorithm 2, for the broadcast-approach, λP_{real} is the number of blocks split across the row of matrix A, in our environment, which recommended it as $8 \times P_{real}$. In blocking-approach, parameter b is the block number across the row and column, in consideration of the Spark official suggestion and the experimental result, which is recommended as $\lfloor \sqrt[3]{3 \times P_{real}} \rfloor$. Similarly, in CARMA approach, we set the parameter $2 \times P_{real}$, not the real number of cores across the cluster P_{real} to the approach.

Algorithm 2: $ApproachSelection(A, B, m, k, n, P_{real})$

Input: A is an $m \times k$ matrix, B is a $k \times n$ matrix, P_{real} is the real number of cores across the cluster
Output: $C = AB$ is $m \times n$

```
begin
    if A or B is under broadcast threshold then
        C = BroadscastMultiply(A, B, λP_real)
    else if m, k, n are close equal then
        C = BlockingMultiply(A, B, b)
    else
        C = CarmaMultiply(A, B, 2P_real)
    return C
end
```

VI. Evaluation

In this section, we evaluate the execution efficiency and scalability of Marlin respectively. First, we conduct a series of experiments to individually evaluate the effect of each proposed optimization and approach. Second, for

comparison, we also evaluate the performance of Marlin, SUMMA [13] (implemented in MPI), HAMA (implemented in MapReduce) [7], and single-node R over various cases. Lastly, we evaluate the scalability performance of Marlin.

A. Experimental Setup

All the experiments are conducted on a physical cluster with 17 nodes. Among them, one is reserved to be master, and the left 16 nodes are used for computing. Each node has two Xeon Quad 2.4 GHz processors altogether 16 logical cores, 24 GB memory and two 2 TB 7200 RPM SATA hard disks. The nodes are connected with 1 Gb/s Ethernet. All the nodes run on RHEL6 operating system and Ext3 file system. The version of the underlying Spark for Marlin is 1.1. And we adopt ATLAS version BLAS 3.2.1 as the native linear algebra library in Marlin. The Hadoop v2.3.0 with jdk 1.6 is installed on this cluster for the compared system. The compared SUMMA is implemented in ScaLaPack 2.0.2 via pbdR. And, a single-node R in version 3.1.1 is also installed for performance comparison.

For description simplicity, we define some terms that will be used in the analysis. Marlin contains three matrix multiplication approaches. Thus, the block-splitting approach is denoted as *Marlin-Blocking*, the CARMA approach is denoted as *Marlin-CARMA*, and the broadcast approach is denoted as *Marlin-Broadcast*. The best approach selected by our adaptive selection algorithm is just denoted as $Marlin$. We perform the experiments over different sizes of matrices which are called different cases. Matrix multiplication needs two input matrices, then we represent a test case as $m \times k \times n$, where size of matrix A is $m \times k$ and the size of matrix B is $k \times n$. In experiments, some cases failed to return results in a reasonable time (1 hour here), their execution time are denoted as NA in the figures and tables.

B. Performance Analysis of Matrix Multiplication

1) Effects of Adopting Native Linear Algebra Library: Firstly, we evaluate the effect of taking advantage of the native linear algebra library. The experiments adopts all the 16 computing nodes in the cluster, and the results are shown in Figure 6. The two input matrices are both square, with dimensions $m = k = n$. We find that both the execution time of enabling BLAS and the disabling BLAS increase as the input matrices' size scales up. Performance difference is not significant for small input matrices. However, as the scale of the input matrices goes up, the enabling BLAS way is significantly faster than the disabling BLAS way. For example, after the dimension case of $15000 \times 15000 \times 15000$, adopting native library is around 2 times faster than simply using Java. The reason is that when the matrices getting large, the computation becomes the bottleneck. And, BLAS, Fortran-implemented is native linear algebra library, is much faster than the libraries based on JVM for CPU-intensive tasks, such as matrix computation.

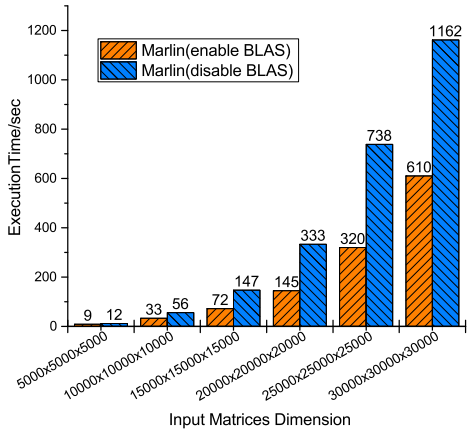

Figure 6. Performance comparison between Marlin enabling BLAS and disabling BLAS

2) Effects of Adaptive Approach Selection: In this subsection, we evaluate the performance of the three approaches on various cases. The experimental results on the representative cases are shown in Figure 7. We can see that the cases fall into three groups with different characteristics. Cases 1 to 3 demonstrate that one of the input matrices is not so large, cases 4 to 6 stand for situations where both input matrices are large and square, and cases 7 to 9 represent situations where both two matrices are large but only with one large dimension. From the performance results, we can find that *Marlin-Broadcast* is around 10x faster than other two approaches for cases 1 to 3. This is because *Marlin-Broadcast* does not involve any shuffle process when the input matrix is small enough to be broadcast to the worker nodes. For cases 4 to 6, the *Marlin-Blocking* can gain better performance than the others, because its tuning block number feature is suitable for large square matrices. From cases 7 to 9, we can find that when one dimension of the two matrices is extremely large, *Marlin-CARMA* approach is significantly faster than the others, because it avoids the shuffle process of input matrix splitting.

3) Effects of Tuning the Matrix Split Granularity: In this subsection, we experimentally study the time cost model analyzed in section V-E1. First, we study the time cost model of blocking splitting approach, and the equal-splitting can be regarded as a special case of this approach. The experiments are conducted by tuning the split block number on the same input matrices which is $20000 \times 20000 \times 20000$. The execution time is shown in Figure 8. The total number of the logical cores in the cluster is 256, thus when block number is less than 7, the computation in the cluster has a low level of parallelism. In these cases, the time cost on *stage-flatMap* and *stage-sum* is quite large. As the block number increasing, *stage-multiply* takes the benefit of reusing the resources. In our experimental environment, we find that

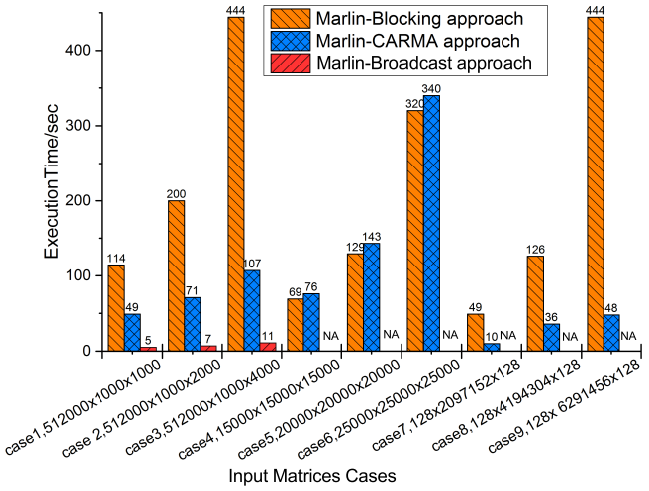

Figure 7. Performance comparison of three multiplication approaches in Marlin

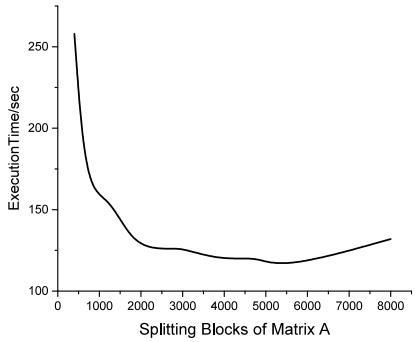

Figure 9. Execution time of two matrices with different split granularity, using broadcast-approach

the best value for the block number is 9, which means the number of total tasks in Spark runtime is $9^3 = 729$, each core has about 2.84 tasks. This fits the 2-3 tasks per CPU core recommended in the Apache Spark tuning guide[3].

Furthermore, we study the time cost model of broadcast approach. The size of the input matrices is $10^6 \times 6000 \times 6000$. The experimental result is illustrated in Figure 9 . We can see that trend is similar to Figure 8 and verifies the analysis in this time model, which means that there exists a split number r along the row which can balance the benefit of resource reusing and the overhead of tasks launching.

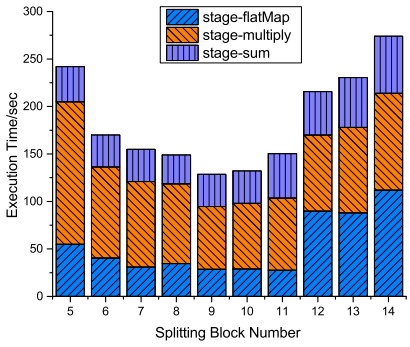

Figure 8. Execution time of two square matrices with different split granularity

4) Performance Comparison With Other Systems: In this section, we compare the performance of Marlin with the similar and representative systems, including SUMMA, HAMA and single-node R. SUMMA is a widely-used MPI-based matrix computation algorithm and it is used in ScaLaPack library. HAMA is a famous matrix computation library built on top of Hadoop. We also choose R, single-node statistical software which also adopts BLAS for acceleration. We used four groups of experiments to cover all the situations.

Group 1 represents the cases where one of the matrices is not large and can be broadcast out. Group 2 stands for the cases where two input matrices are not so large in size, but dimension m and n are large which lead to extremely large product result. According to the approach selection algorithm, Marlin adopts broadcast approach for both group 1 and group 2. The experimental results are shown in Table I. It can be seen that performance of Marlin is much faster than the MapReduce and R. This is because, compared with Marlin, HAMA has a lot of disk writing and network synchronization operations and R is only single-node, hard to deal with large matrix efficiently. Moreover, we find that marlin achieves even better performance than SUMMA. This is because SUMMA is multi-process mode and fails to utilize the SMP architecture well. In its implementation, different processes on the same computing node still transfers the data through passing message over the local network, while in one Spark executor, different threads share the data directly through memory in the same process space.

The performance comparison of the two large matrix multiplication is illustrated in Figure 10 and Figure 11. The Figure 10 shows the skew-shaped matrix cases, while the Figure 11 shows the square matrix cases. We find that the performance of Marlin is comparable to SUMMA. For skew-shaped matrix cases, Marlin is even faster. This is because SUMMA is not communication-optimal on the one large dimension case [15], while the CARMA adopted in Marlin is. Moreover, Marlin is built on top of Spark, which has better fault tolerance than SUMMA.

[3]Apache Spark tuning guide http://spark.apache.org/docs/1.0.0/tuning.html#level-of-parallelism

Table I
PERFORMANCE COMPARISON OF THE FOUR SYSTEMS, IN THE CASES THAT ONE OF THE MATRICES IS NOT SO LARGE, MARLIN ADOPTS BROADCAST APPROACH (THE UNIT OF EXECUTION TIME IS SECOND).

Matrix dimension	Marlin	SUMMA	HAMA	R
512000x1000x1000	5	10.6	1250	148
1024000x1000x1000	10	20.3	3000	297
1536000x1000x1000	12	29	NA	906
2048000x1000x1000	13	39	NA	3302
2560000x1000x1000	16	79	NA	NA
65536x128x65536	7	7	NA	1163
81920x128x81920	8	9.7	NA	NA
98304x128x98304	10	15	NA	NA
114688x128x114688	13	17	NA	NA
131072x128x131072	16	21.4	NA	NA

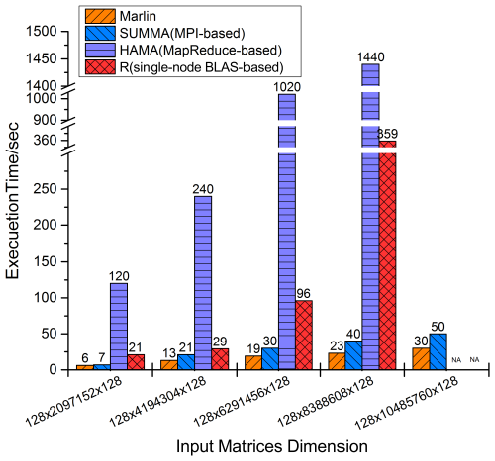

Figure 10. Performance comparison of the four systems, in the cases that two large-scale matrices with one large dimension

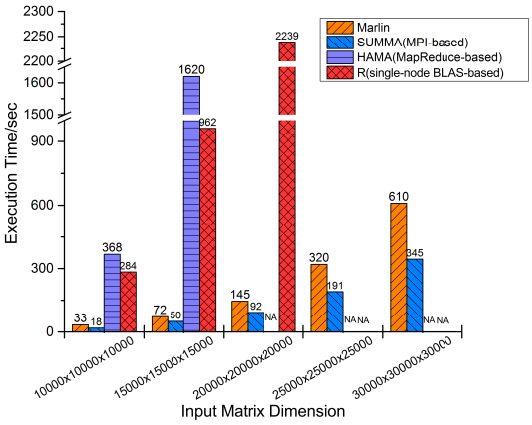

Figure 11. Performance comparison of the four systems, in the cases that two large-scale square matrices

C. Scalability Performance Analysis

We evaluated the scalability of Marlin by carrying out two experiments (1) scaling the size of input matrix while fixing the number of machines, and (2) scaling the number of machines while fixing size of input data.

1) Data Scalability: Experimental results are shown in Figure 12, and we observe that execution time of Marlin grows close to linearly with the increase of data size. It indicates that Marlin achieves near linear data scalability.

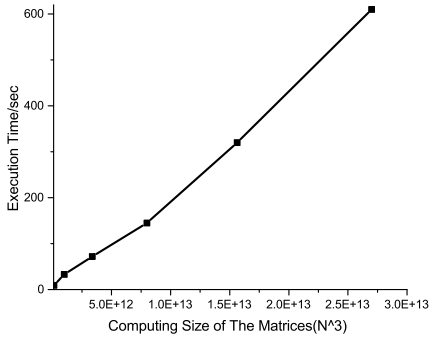

Figure 12. Data scalability

2) Node Scalability: We also conduct a group of experiments to evaluate the machine scalability performance of Marlin. The dimensions of the input matrices are fixed at $15000 \times 15000 \times 15000$ in these experiments. It can be seen from Figure 13, the execution time of Marlin decreases about 3 times as the number of cluster nodes increases. This means Marlin has near linear machine scalability.

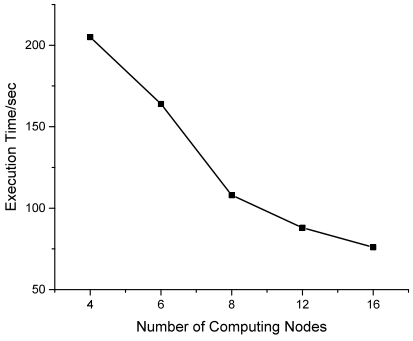

Figure 13. Node scalability

VII. CONCLUSION AND FUTURE WORK

Matrix computation is a fundamental kernel of scientific computing and machine learning applications such as social network mining. In this paper, we propose Marlin, an

efficient distributed matrix computation library built on top of Spark. Three distributed matrix multiplication algorithms, suitable for different scenarios, are designed in Marlin. Also, an adaptive model is proposed to select the best matrix multiplication approach for problems with different characteristics. Experimental results show that Marlin achieves over an order of magnitude speedup than the existing distributed matrix operation algorithms based on MapReduce and single-node R. Also, Marlin is comparable to or even faster than the widely-used MPI matrix multiplication algorithm, SUMMA. Moreover, Marlins is built on top of a general dataflow engine and gains good dataflow features such as scalability and fault tolerance.

In the future work, we plan to design and implement more distributed matrix computation algorithms, such as matrix factorization, into Marlin. We also want to explore more optimizations for dense matrix multiplication and study the sparse matrix computation problem on Spark.

VIII. Acknowledgments.

This work is funded in part by China NSF Grants (No.61572250), Jiangsu Province Industry Support Program (BE2014131) and China NSF Grants (No.61223003).

References

[1] G. Stewart, "The decompositional approach to matrix computation," *Computing in Science & Engineering*, vol. 2, no. 1, pp. 50–59, 2000.

[2] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: an API for distributed machine learning," in *13th IEEE International Conference on Data Mining (ICDM), Dallas, TX, USA, December 7-10, 2013*, 2013, pp. 1187–1192.

[3] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: distributed machine learning and graph processing with sparse matrices," in *2013 European Conference on Computer Systems (EuroSys), Prague, Czech Republic, April 14-17, 2013*, 2013, pp. 197–210.

[4] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *27th International Conference on Data Engineering(ICDE), Hannover, Germany, April 11-16, 2011*, 2011, pp. 231–242.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers," in *Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 1992, pp. 120–127.

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

[7] 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 *CloudCom*. IEEE, 2010, pp. 721–726.

[8] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "Madlinq: large-scale distributed matrix computation for the cloud," in *EuroSys*. ACM, 2012, pp. 197–210.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*. USENIX Association, 2012.

[10] "R project," 2011. [Online]. Available: http://www.r-project.org/

[11] W. F. McColl and A. Tiskin, "Memory-efficient matrix multiplication in the bsp model," *Algorithmica*, vol. 24, no. 3-4, pp. 287–297, 1999.

[12] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *2011 European Conference on Parallel and Distributed Computing(Euro-Par)*. Springer, 2011, pp. 90–109.

[13] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency-Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[14] E. Solomonik and J. Demmel, "Matrix multiplication on multidimensional torus networks," in *VECPAR 2012*. Springer, 2013, pp. 201–215.

[15] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *27th IEEE International Parallel & Distributed Processing Symposium(IPDPS)*. IEEE, 2013, pp. 261–272.

[16] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *ACM Symposium on Parallelism in Algorithms and Architecture(SPAA)*. ACM, 2012, pp. 193–204.

[17] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen: Implementation and performance," in *2012 The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE Computer Society Press, 2012, p. 101.

[18] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *2011 ACM International Conference on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, vol. 41, no. 4. ACM, 2011, pp. 98–109.

[19] D. Schmidt, G. Ostrouchov, W.-C. Chen, and P. B. Patel, "Tight coupling of r and distributed linear algebra for high-level programming with big data." in *SC Companion*, 2012, pp. 811–815.