Exploiting Matrix Dependency for Efficient Distributed Matrix Computation

Lele Yu Yingxia Shao Bin Cui
Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
{leleyu, simon0227, bin.cui}@pku.edu.cn

ABSTRACT

Distributed matrix computation is a popular approach for many large-scale data analysis and machine learning tasks. However existing distributed matrix computation systems generally incur heavy communication cost during the runtime, which degrades the overall performance. In this paper, we propose a novel matrix computation system, named DMac, which exploits the matrix dependencies in matrix programs for efficient matrix computation in the distributed environment. We decompose each matrix program into a sequence of operations, and reveal the matrix dependencies between operations in the program. We next design a dependency-oriented cost model to select an optimal execution strategy for each operation, and generate a communication efficient execution plan for the matrix computation program. To facilitate the matrix computation in distributed systems, we further divide the execution plan into multiple un-interleaved stages which can run in a distributed cluster with efficient local execution strategy on each worker. The DMac system has been implemented on a popular general-purpose data processing framework, Spark. The experimental results demonstrate that our techniques can significantly improve the performance of a wide range of matrix programs.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

Keywords

matrix computing; dependency analysis; distributed system

1. INTRODUCTION

Matrix-based computation is one of the most general approaches for a deep analysis or manipulation of the data at hand. An array of machine learning algorithms, such as collaborative filtering, Cholesky factorization [12], singular value decomposition (SVD) and LU factorization, are built on top of matrix computation. It is also true for many data mining algorithms, like Betweeness Centrality [13] and PageRank. Taking account of the importance and generality of matrix computation, it is natural to develop efficient execution engines for large-scale matrix computation. Different distributed matrix computation systems have been proposed recently, such as SystemML [14, 18], MadLINQ [20], and Mahout [4].

For data processing tasks in the distributed environment, communication cost is generally a key factor of the overall performance and may become a bottleneck of the system execution. In MadLINQ [20], the matrix computation algorithms are expressed as sequential programs operating on tiles (i.e., sub-matrices) and executed on DryadLINQ [22]. SystemML [14, 18], which is the latest distributed matrix computation system, expresses machine learning algorithms (i.e., matrix computation) in an R-like high-level language and executes them in the MapReduce framework [11]. It produces an execution plan via the piggybacking algorithm which considers the characteristics of operations and matrices separately. In these systems, the matrix computation generally incurs expensive repartition phases to redistribute the matrix, which brings in heavy communication cost and makes the matrix computation still expensive in distributed systems.

By examining the matrix computation processing in the distributed environment, we observe the operations in a matrix program have specific requirements for the distribution of input matrices in the system. The matrix operations in the program may access some matrices successively, i.e., the input matrix of an operation can be the output of its preceding operation. We call it "matrix dependency" in the matrix program. When the requirements and the partition schemes of the matrices are inconsistent, an expensive repartition phase is needed. The following code segment describes a representative example to clarify the problem. Code 1 is an illustration of the popular ML algorithm Gaussian Non-Negative Matrix Factorization [16] (GNMF). GNMF is an algorithm for finding two factor matrices, W and H, such that $V \approx WH$.

```
val max_iteration = 10

// load the matrices
val V = load(path = ...)
val (W, H) = (RandomMatrix, RandomMatrix)

// iteratively update W and H
for ( i <- 0 until max_iteration) {
   H = H * (W.t %*% V) / (W.t %*% W %*% H)
   W = W * (V %*% H.t) / (W %*% H %*% H.t)
}
```

Code 1: GNMF algorithm in Scala

In GNMF, we can see there exist matrix dependencies during the matrix computation. First, when computing the multiplication [1] of

[1]Multiplication is represented by %*%, while cell-wise multiplication is represented as *.

$W.t$ and W for H in the i^{th} iteration, the matrices $W.t$[2] and W are dependent on the results from the previous iteration. SystemML simply maintains the matrix W with hash partition scheme which is not suitable for the multiplication operation, therefore a repartition phase is required before the real computation. This is also true for computing $H\%*\%H.t$. Second, even if the partition scheme of W from the previous iteration meets the requirement, SystemML needs to repartition it for $W.t$ as well. Actually, being aware of the dependency between W and $W.t$, the two matrices can be mutually transformed without communication. Therefore, by analyzing the matrix dependency in the program and eliminating the inconsistency, we can reduce the number of communication operations and improve the overall performance. However, the existing systems ignore such matrix dependency information in the program, and hence lose the optimization opportunity for communication.

In this paper, we propose a new matrix computation system, named DMac, which novelly exploits the matrix dependency during the matrix computation to reduce the communication cost in the distributed environment. We integrate partition schemes into the matrix dependency analysis and show the effect of various dependencies on the communication. By utilizing the dependency information, we build a cost model to select execution strategies for matrix operators, and design a plan generation algorithm to construct a communication efficient execution plan for the whole matrix computation job. Once the plan is generated, DMac divides the plan into some un-interleaved stages by merging all operations without communication into a single stage, which are executed across a distributed cluster. Furthermore, each worker uses a block-based approach for efficient local matrix computation to improve the local performance. We have implemented the prototype of DMac on Spark [23] and conduct comprehensive experiments on various matrix based applications.

In summary, the main contributions of our work are listed as follows.

1. We develop a new matrix computation system (DMac) for efficient matrix computation in the distributed environment.
2. We propose to exploit the matrix dependency in a matrix program to reduce the communication cost. To the best of our knowledge, we are the first to propose the concept of matrix dependency and show its benefit in distributed matrix computation.
3. We design a cost-model based algorithm to generate communication efficient execution plans for matrix computation tasks.
4. We implement a prototype DMac system, and conduct extensive empirical studies on various matrix programs. The experimental results demonstrate the superiority of our approach compared with the existing methods.

The remainder of the paper is organized as follows. Section 2 presents the overview of DMac system. The definition of matrix dependency and its analysis are described in Section 3. We present the algorithm to generate the execution plan in Section 4. Section 5 depicts the deployment of DMac in a distributed cluster and its implementation on Spark. The experimental results are presented in Section 6. Section 7 introduces the related work, and finally we conclude the work in Section 8.

2. OVERVIEW OF DMAC

DMac is a distributed matrix computation system. The high-level architecture of DMac is shown in Figure 1. For each submitted matrix program, the plan generator in DMac is responsible to

[2] W^T and $W.t$ are used interchangeably.

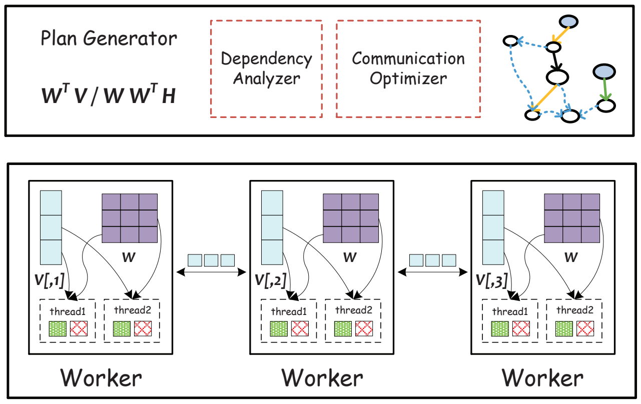


Figure 1: Architecture of DMac

create a communication efficient execution plan for the matrix computation task. The plan is next scheduled with a sequence of stages, making that the operations in each stage can run without any communication. Each stage is distributively executed in a cluster with a set of workers. To execute matrix operations efficiently at each worker locally, a block-based approach which can fully exploit the parallelism of modern machine is applied.

As we introduced previously, the communication cost is a major factor of the performance for distributed matrix computation. Therefore, the key issue of the system is how to construct a communication efficient execution plan in our system design. Once a user submits a matrix program to DMac, the dependency analyzer in plan generator first decomposes the program into a sequence of operations along with analyzing the matrix dependencies, which contains the information about how the operations refer to the output matrices of their preceding operations. Because various partition schemes can be applied for one matrix, each operation has many alternative execution strategies which specify the partition scheme requirements of the corresponding operation. The optimizer in DMac helps select the optimal execution strategy for each operation based on the dependency-oriented cost model. The overall execution plan can be generated after each operation's execution strategy is determined. In the next two sections, we will present the novel concept of matrix dependency, and show how to generate the communication efficient execution plan by exploiting the dependency.

3. MATRIX DEPENDENCY

In this section, we give a formalized definition of matrix dependency, and analyse its effect on the communication of matrix computation. The traditional definition of data dependency is a situation in which a program statement refers to the data of a preceding statement. By investigating the matrix program, we observe that the operators in the program may access some matrices successively, i.e., there does exist kinds of "dependency" between different matrix operators. In the distributed environment, the situation could be more complicated since each matrix operator is associated with a partition scheme of input/output matrices and there exist multiple execution strategies for each operator. Therefore, the system has to effectively utilize the dependencies between operators to choose an efficient execution strategy for each operator.

3.1 Preliminaries

Before introducing the dependency definition in matrix program, we describe the notations used to illustrate the dependency and how the dependency affects the communication of operators. The notations and their explanations are summarized in Table 1.

Notations	Description
op_i, op_j	operator in the program
$Precede(op_i, op_j)$	op_i is performed earlier than op_j in the program.
p_i, p_j	partition scheme of matrix
$EqualB(p_i, p_j)$	p_i and p_j are both Broadcast scheme.
$EqualRC(p_i, p_j)$	p_i and p_j are the same, either Row scheme or Column scheme.
$Oppose(p_i, p_j)$	p_i is Row scheme while p_j is Column scheme and vice verse.
$Contain(p_i, p_j)$	p_i is Broadcast scheme while p_j is either Row scheme or Column scheme.
$In(A, p_i, op_i)$	op_i requires matrix A with p_i as input.
$Out(A, p_i, op_i)$	op_i generates matrix A with p_i.

Table 1: Related notations for matrix dependency

- **Operator:** As we introduced previously, the matrix computation task can be decomposed into a sequence of operations in the execution, each of which corresponds to an operator. There are five binary operators suppported in DMac, including multiplication, addition, substraction, cell-wise multiplication and cell-wise division. Furthermore, unary operator between a constant and a matrix is also supported.
- **Partition scheme:** In a matrix program, each matrix is associated with a partition scheme. DMac adopts three partition schemes, which are Row scheme, Column scheme and Broadcast scheme. Row/Column scheme, denoted as r/c, partitions the elements belonging to the same row/column into the same partition. Broadcast scheme, denoted as b, generates a replica of every element at each partition. Note that Broadcast is not a partitioning method, but it can indicate the data placement information in the cluster, which is similar to row/column partition scheme. For the ease of representation, we regard Broadcast as a kind of partition scheme. These three partition schemes are sufficient for all the matrix operations, since the partition schemes for the output matrix of the operations must be one of them. The partition schemes adopted by DMac are one-dimensional partitioning methods, which can reduce the number of shuffle operations for the operators supported in DMac. The two-dimensional partitioning methods, such as chunk-based [8] and block-cyclic [5], have their own merits to process some other matrix operators but with more computation stages, which will be investigated in future work. Furthermore, we define four constraints between two partition schemes which are $EqualB(p_i, p_j)$, $EqualRC(p_i, p_j)$, $Oppose(p_i, p_j)$ and $Contain(p_i, p_j)$.
- **Event:** An event is an input (output) process of reading (writing) a single matrix by an operator. $In(A, p_i, op_i)$ is an *input event* and $Out(A, p_i, op_i)$ is an output event. The event notation will help us properly capture the dependency between dependent operators.

3.2 Dependency Definition

We proceed to give the formalized definition of matrix dependency, which implies whether the operator refers to the output matrix of a preceding operator.

DEFINITION 1 (**MATRIX DEPENDENCY**). *An input event $In(B, p_j, op_j)$ is dependent on an output event $Out(A, p_i, op_i)$ if $B = A$ or $B = A^T$, and $Precede(op_i, op_j)$ holds. This dependency between $In(B, p_j, op_j)$ and $Out(A, p_i, op_i)$ is called Matrix Dependency.*

Matrix dependency reveals the relationship between the input event and output event of two matrix operators. By considering the partition schemes of matrices A and B and the relationship between them, there exist 18 combinations between $Out(A, p_i, op_i)$ and $In(B, p_j, op_j)$. Each combination requires a matrix process to make the partition scheme of A satisfy the demand of B. We analyse these combinations and find that actually eight different matrix processes are sufficient. Thereby, we classify the matrix dependencies into eight types corresponding to the matrix processes, and use the matrix process as the name of dependency type. Table 2 lists the specific conditions for each type of matrix dependency and the dependency type column contains the corresponding names of the matrix process.

We further check whether the matrix dependency will introduce communication cost, since it is a critical issue for us to select an optimal execution strategy for each operator. It is clear some matrix processes need to repartition the matrix, and hence incur the communication in the cluster, while the others do not. Therefore, we divide eight types of dependencies into two categories, and give the detailed analysis as follows.

Dependency with Communication Cost

If the dependency incurs communication cost for matrix processing, it belongs to the Communication Dependency category. There exist four types of matrix dependencies in this category.

1. **Partition Dependency:** It depicts the scenario where the partition scheme required by op_j is opposed with the partition scheme generated by op_i on the same matrix. Thus, a repartition step is needed to eliminate the inconsistency of partition schemes between op_i and op_j. Clearly, the repartition of the matrix will introduce the communication in the distributed environment.
2. **Transpose-Partition Dependency:** It appears when $B = A^T$ and both p_i and p_j are Row scheme or Column scheme. To fulfill the requirement of this dependency, a transpose operation is first applied and a repartition step is also needed to change the partition scheme. Therefore, communication cost is induced.
3. **Broadcast Dependency**: It means that op_i generates matrix A with a Row or Column partition scheme while op_j requires a Broadcast scheme of the input matrix. In this situation, repartition is needed to broadcast A across the cluster, and hence the communication is needed.
4. **Transpose-Broadcast Dependency:** When $B = A^T$ and $Contain(p_j, p_i)$, a matrix transpose process and a broadcast step are required to satisfy this dependency, thus communication happens.

Dependency without Communication Cost

If the dependency will not incur any communication cost, we classify it in the Non-Communication Dependency category.

1. **Reference Dependency:** It exists when the matrix generated by op_i and its partition scheme exactly satisfy the input requirement of op_j. No communication cost will be introduced since op_j can directly use the data generated by op_i without a repartition step.
2. **Transpose Dependency**: If $B = A^T$ and $Oppose(p_i, p_j)$ or both p_i, p_j are Broadcast scheme, a local transpose operation can satisfy the requirement of this dependency, i.e., no communication is needed.
3. **Extract Dependency:** It depicts that op_i generates a Broadcast scheme of matrix while op_j requires a Row scheme or Column scheme. Thus, op_j can obtain the required scheme through an extract operation on the data generated by op_i.

Events	Conditions	Dependency Type	Communication
$Out(A, p_i, op_i)$ / $In(B, p_j, op_j)$	$A = B$ && $Oppose(p_i, p_j)$	Partition	Yes
	$A = B^T$ && $EqualRC(p_i, p_j)$	Transpose-Partition	Yes
	$A = B$ && $Contain(p_j, p_i)$	Broadcast	Yes
	$A = B^T$ && $Contain(p_j, p_i)$	Transpose-Broadcast	Yes
	$A = B$ && $(EqualRC(p_i, p_j) \mid\mid EqualB(p_i, p_j))$	Reference	No
	$A = B^T$ && $(Oppose(p_i, p_j) \mid\mid EqualB(p_i, p_j))$	Transpose	No
	$A = B$ && $Contain(p_i, p_j)$	Extract	No
	$A = B^T$ && $Contain(p_i, p_j)$	Extract-Transpose	No

Table 2: Description of the eight types of matrix dependencies

This extract operation can be conducted locally on the workers and no communication operation is invoked.

4. **Extract-Transpose Dependency:** It exists when $B = A^T$ and $Contain(p_i, p_j)$. Thus, op_j can obtain the required scheme through a local extract operation and a following transpose operation. Both operations will not introduce communication cost.

The communication analysis of different matrix dependencies enables us to select an efficient execution strategy for the operations in the matrix program, which facilitates the generation of a communication efficient execution plan of the matrix computation task.

4. PLAN GENERATION

Given a matrix program, DMac decomposes it into a sequence of matrix operators. For each operator, a dependency oriented cost model is utilized to select the execution strategy with the minimum communication. The detailed algorithm of generating the execution plan is described in Section 4.2.

4.1 Dependency-Oriented Cost Model

By considering the matrix dependencies between operators' events, a cost model is utilized to find a strategy with the minimum communication cost for each matrix operator. $\forall\, op_i, S_i = \{s_{i0}, s_{i1}, ..., s_{in}\}$ represents the candidate execution strategies for op_i. For $0 \leq k \leq n$, the input event set and the output event of s_{ik} are denoted as IS_{ik} and out_{ik} respectively. Note that, for each execution strategy of an operator, there are only one output event and at most two input events. Then the communication cost introduced by s_{ik} can be obtained by the following equation:

$$Cost(s_{ik}) = \sum_{in \in IS_{ik}} Cost(in) \quad + \quad Cost(out_{ik})$$

The total communication cost of s_{ik} is the summation of costs introduced by all the input events and output event inside s_{ik}. In DMac, strategy choosing for matrix operators is conducted in the program order. Given a matrix operator op_i, let $OutputSet$ indicate the set of output events from those strategies which are selected for the operators ahead of op_i. Considering an input event $In(A, p_i, op_i)$ from an execution strategy s_{ik}, if there exists an output event $Out(B, p_j, op_j) \in OutputSet$ such that $In(A, p_i, op_i)$ is dependent on $Out(B, p_j, op_j)$, the matrix dependency *dep* between these two events only needs to be analyzed in three situations.

1: *dep* belongs to Non-Communication Dependency category.
2: *dep* is either *Partition* or *Transpose-Partition* dependency.
3: *dep* is either *Broadcast* or *Transpose-Broadcast* dependency.

In Situation 1, no communication cost will be introduced by $In(A, p_i, op_i)$; while communication is required in Situation 2 and Situation 3. Therefore, the communication cost of $In(A, p_i, op_i)$ can be derived as follows:

$$Cost(In(A, p_i, op_i)) = \begin{cases} 0 & \text{Situation 1} \\ |A| & \text{Situation 2} \\ N \times |A| & \text{Situation 3} \end{cases}$$

Here, N is the number of nodes in the cluster, and $|A|$ is the size of matrix A which can be derived through a worst-case estimation method. The detail description of this method is presented in Section 5.1. Note that, the matrix size may vary if we apply matrix compression techniques for sparse matrix in the real implementation; however, it will not affect the analysis here.

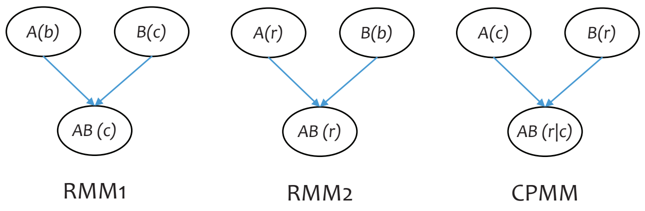


Figure 2: Execution strategies for multiplication. A(p) is the matrix A with partition scheme p.

The cost of each input event can be directly derived by the dependency type between it and its dependent output event. However, the cost of an output event can be different with respect to the selected operator execution strategy, specifically for matrix multiplication operation. For example, there are three execution strategies for matrix multiplication [14], called Replication based Matrix Multiplication (*RMM1*, *RMM2*) and cross-product matrix multiplication (*CPMM*). The details of three strategies are specified in Figure 2. In these execution strategies, *CPMM* introduces communication cost during its execution while there is no communication cost for the other two strategies. We use the cost of an output event to represent the communication cost introduced during the execution of a certain strategy.

The communication cost for an output event $Out(A, p_i, op_i)$ of s_{ik} can be obtained by the following equation:

$$Cost(Out(A, p_i, op_i)) = \begin{cases} N \times |A| & \text{if } s_{ik} \text{ is } \textit{CPMM} \\ 0 & \text{otherwise} \end{cases}$$

When the execution strategy is *CPMM*, the communication cost is $N \times |A|$. The cost of the output event of other strategies is zero.

According to the cost analysis above, an execution strategy with the minimum communication cost for matrix operator op_i can be chosen from the following equation:

$$s_i = \arg\min_{s \in S_i} Cost(s) \qquad (1)$$

4.2 Algorithm of Execution Plan Generation

The plan generation for a matrix program can be considered as a problem of choosing a strategy for each matrix operator such that the total communication of all the strategies is minimized. In DMac, we first decompose the matrix program into a sequence of matrix operators. For each operator, an execution strategy is chosen to perform the computation. To express the matrix dependencies in the program, DMac provides five extended operators. Furthermore, we propose two heuristic rules in the algorithm design to optimize the communication cost of execution plan.

4.2.1 *Extended Operators*

To express the dependencies in the matrix program and construct the execution plan, DMac extends the original matrix operators by adding five new operators, including `partition`, `broadcast`, `transpose`, `reference` and `extract`.

The `partition` operator performs a repartition step to change any partition scheme to Row scheme or Column scheme. The `broadcast` is responsible to change a partition scheme to Broadcast scheme by replicating the local partition to other computing nodes. These two operators can be used to represent the dependencies with communication

Dependencies without communication cost are represented by `reference`, `transpose` and `extract`. The `reference` operator is a null operation only for representing the *Reference* dependency and is applied when p_i in $In(A, p_i, op_i)$ is same as p_j in $Out(A, p_j, op_j)$. The `transpose` operator will generate a transpose version of a matrix with complementary partition scheme only by local operations. The `extract` operator is corresponding to the inconsistency between Broadcast scheme and other two schemes. It generates Row scheme or Column scheme through performing a filter step at local.

4.2.2 *Heuristic Methods*

Through analysing the operator sequence and its matrix dependencies, we observe the matrix broadcast and matrix dependency with repartition requirement are the major reasons incurring the communication. Our algorithm applies two heuristic methods to reduce the communication in the matrix computation, named **Pull-Up Broadcast** and **Re-assignment**. The description of these two methods are specified as follows.

HEURISTIC 1. ***(Pull-Up Broadcast)*** *If two input events $In(A, p_i, op_i)$, $In(B, p_j, op_j)$ satisfy the following conditions, $B = A$ or $B = A^T$, $Contain(p_j, p_i)$, $Precede(op_i, op_j)$ and $Cost(In(A, p_i, op_i)) > 0$, $Cost(In(B, p_j, op_j)) > 0$, then add an output event $Out(A, b, op_i)$.*

When the conditions specified above are satisfied, we can observe that these two input events both introduce communication cost. The cost from $In(A, p_i, op_i)$ is produced by the repartition step for generating a Row scheme or Column scheme of A. The other cost from $In(B, p_j, op_j)$ is brought in by a broadcast step. The communication cost generated by $In(A, p_i, op_i)$ can be eliminated once op_j is performed before op_i. Hence, we broadcast A in advance.

HEURISTIC 2. ***(Re-assignment)*** *In case that $In(A, p_i, op_i)$ is dependent on $Out(B, p_j, op_j)$ and $Cost(In(A, p_i, op_i)) > 0$, if p_j has multiple values, we re-assign p_j to the value with minimum $Cost(In(A, p_i, op_i))$.*

Some strategies can generate an output event, $Out(B, p_j, op_j)$, with multiple partition schemes, such as the *CPMM* strategy. Furthermore, when multiplying two matrices with the same size, like BB^T, RMM1 and RMM2 can generated result matrix with different partition scheme while introducing the same amount of communication cost. If there is a following input event, $In(A, p_i, op_i)$, being dependent on $Out(B, p_j, op_j)$, the value of p_j should be carefully chosen to construct Non-Communication Dependency between these two events. DMac re-assigns the value of p_j if $In(A, p_i, op_i)$ introduces communication cost.

4.2.3 *Algorithm Description*

Algorithm 1 depicts the logic of the plan generation algorithm. We first initialize an `OutputSet` and an `InputSet` which will be

Algorithm 1 generating execution plan

```
Input: $matrix\ program\ P$
 1: OutputSet $\leftarrow \Phi$, InputSet $\leftarrow \Phi$
 2: Plan $\leftarrow \Phi$
 3: Matrix Operator List $\leftarrow P$
 4: foreach $op_i$ in Matrix Operator List do
 5:     $S_i \leftarrow$ candidate execution strategy set for $op_i$
 6:
 7:     /* select a strategy for $op_i$*/
 8:     $s_i \leftarrow \arg\min_{s \in S_i} Cost(s)$
 9:
10:     foreach $In(A, p_i, op_i)$ in $s_i.getInput$ do
11:         if $Cost(In(A, p_i, op_i)) > 0$ then
12:             if Re-Assignment satisfies then
13:                 perform Re-Assignment
14:             else
15:                 if Pull-Up Broadcast satisfies then
16:                     perform Pull-Up Broadcast
17:                 end if
18:             else
19:                 OutputSet.$add(Out(A, p_i, op_i))$
20:             end if
21:         end if
22:         InputSet.$add(Input(A, p_i, op_i))$
23:         add dependency introduced by $In(A, p_i, op_i)$ into Plan
24:     end foreach
25:     add $op_i$ into Plan
26: end foreach
27: return Plan
```

used to store the generated input events and output events. An empty plan is also created.

In Line 3, we decompose the given matrix program into a sequence of matrix operators. At this decomposing phase, if there are multiple operators can be executed simultaneously, we put the operators with multiplication ahead of the other operators because matrices will be probably broadcasted by multiplication. According to Heuristic 1, a pull-up broadcast operation can benefit communication. For each matrix operator, we select the execution strategy with the minimum communication cost according to the current $OutputSet$ and Equation 1 (Lines 4-8).

If an input event introduces communication cost, we utilize Heuristic 1 and Heuristic 2 to reduce the communication cost at Lines 12-17. If the communication cost of an input event cannot be avoided, a repartition step will be performed during the execution to fulfill the requirement of this input event. To decrease the communication cost produced in the following operators, we add an output event into the `OutputSet`, as it is described at Line 19. Each input event generated by s_i is added into the `InputSet` at Line 22, which can help the successive operators to apply the Pull-Up Broadcast rule. We then add the dependency generated by the selected strategy into the plan at Line 23. In our plan, dependency is represented with the extended operators.

In the end of outer for loop, op_i is added into the plan at Line 25. Thereby, the execution plan for the whole matrix program is generated, which consists of the original matrix operations and some helpful dependency operators.

4.2.4 *An Example of Generated Execution Plan*

Figure 3 shows the execution plan generated by DMac for the first iteration phase of GNMF (Code 1). The plan is represented by a direct acyclic graph (DAG) where nodes are matrices and edges stand for operators. Each matrix, represented as an ellipse in the figure, is associated with its distributed data layout information. For example, $H_1(c)$ is the input matrix H applying a column scheme and $W_1(b)$ means that W_1 is broadcast to all computing nodes.

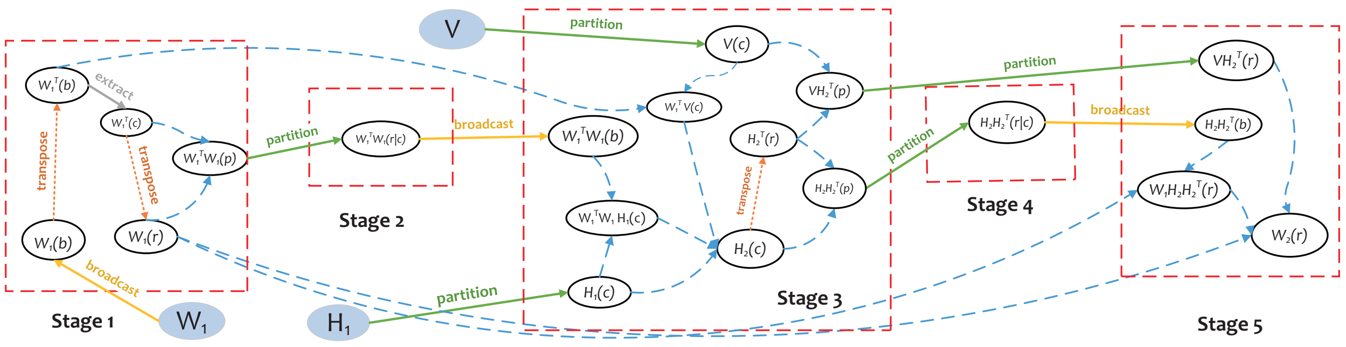


Figure 3: Execution plan for GNMF. The dashed blue arrow means local operation.

Thus the plan gives us an intuitive perspective about how each (intermediate) matrix is partitioned among the cluster. The matrix dependencies are represented by the extended operators introduced in Section 4.2.1. These operators help simplify the representations of various dependencies in a matrix program and distinguish them clearly. We give a brief description for the process of the first 5 operators in the plan as follows.

The first operator is $W.t\ \% * \%\ V$. Since W^TV is larger than W^T, the strategy with the minimum communication cost is to get a broadcast scheme of W^T and perform multiplication with a Column scheme of V. Thus, $W^T(b)$ is generated from a broadcast operator and a transpose operator. A Column scheme of W^TV, denoted as $W^TV(c)$ in Figure 3, is generated.

The following operator is $W.t\ \% * \%\ W$, the strategy with the minimum communication cost is *CPMM*. The two input events of *CPMM* are $In(W, r)$ and $In(W^T, c)$, which can be obtained from $W^T(b)$ according to *Extract-Transpose* dependency and *Extract* dependency. The output of *CPMM* can generate multiple partition schemes for W^TW.

The following operator is $W^TW\ \% * \%\ H$. Since W^TW is a tiny matrix, the strategy chosen is to broadcast W^TW and perform multiplication with a Column scheme of H. Thus, $H(c)$ and $W^TW(b)$ are generated. The output of this operator is W^TWH with the Column scheme, which is $W^TWH(c)$.

The next operation is $H\ *\ W^TV$. $H(c)$ and $W^TV(c)$ have been generated by the preceding operators. Thus, the strategy chosen here is to perform the cell-wise multiplication on $H(c)$ and $W^TV(c)$, which will not introduce any communication cost. The output of this operator is denoted as X and its partition scheme is Column scheme.

The following operator is $X\ /\ W^TWH$. Since both matrices generated from preceding operators are associated with Column scheme, the strategy with minimal cost here is a cell-wise division on $X(c)$ and $W^TWH(c)$. The communication cost of this operator is zero.

After the process of the above operators, the first H computation in Code 1 (Line 9) is finished and the following operators will be executed in the similar way.

5. PLAN EXECUTION AND SYSTEM IMPLEMENTATION

In this section, we will describe the matrix size estimation method and how the generated plan is executed in the distributed environment. Since the matrix operations, like multiplication, are also cpu-intensive tasks, providing an efficient execution process in the cluster is not a trivial task. We first introduce the scheduling of execution plan, followed by the execution flow inside each computing node. The implementation of DMac on the Spark platform will be also presented.

5.1 Worst-Case Matrix Estimation

The characteristics of each matrix is the key feature and it is utilized by the dependency-oriented cost model in DMac. To derive the size of each matrix, a worst-case estimate method is employed. There are two important features needed to infer the size of a matrix. One is the dimension of matrix and the other is the sparsity. The dimension of each matrix can be exactly inferred for many linear algebra operators (e.g., matrix multiplication and addition). Inferring the sparsity, however, is more difficult due to the skew of data. In DMac, the sparsity or number of non-zero items of the input matrices can be pre-computed offline or specified by the user. Then the size of the intermediate matrix is estimated through the worst-case method.

Unary operators can perseve the sparsity between input and output matrices while binary operators cannot. Considering a binary matrix operator $C = op(A, B)$, s_A, s_B are the sparsity of matrix A, B respectively. Then the sparsity of matrix C, denoted as s_C can be estimated through the following equation:

$$s_C = \begin{cases} 1 & \text{if op is multiplication} \\ Max(s_A + s_B, 1) & \text{otherwise} \end{cases}$$

5.2 Plan Scheduling

Given an execution plan generated by Algorithm 1, DMac first schedules it into several un-interleaved stages where each stage can be executed among the cluster without network communication. DMac employs a traverse-based algorithm to find the boundaries between stages. The result matrix of the matrix computation acts as the starting point of the traversal. The algorithm traverses the plan following the matrix dependency. When the traverse encounters a dependency with communication cost (e.g., `partition` operator or `broadcast` operator) and there is no other matrix dependencies without communication cost that lead to unvisited nodes, the boundary of current stage is found. As shown in Figure 3, the execution plan of the first iteration of GNMF algorithm is divided into five stages, where the boundaries between stages are either `partition` operator or `broadcast` operator or both.

Each stage is then ran across the cluster. Since network communication only happens between stages, the computing tasks inside a stage can be perfectly dispatched to the nodes in the cluster and executed independently. Every computing node fetches the input data, which are portions of input matrices, and executes the matrix operators in this stage. After that, the nodes write the output matrix to the local disk for later computation.

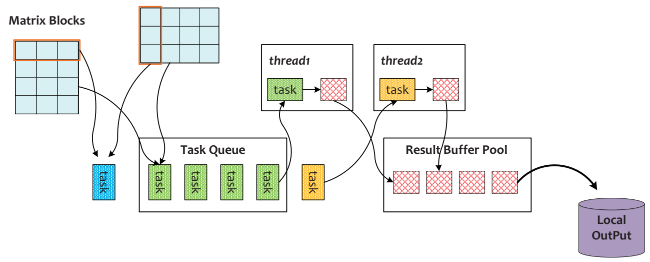


Figure 4: Execution flow inside a worker

5.3 Local Execution on Computing Node

In DMac, a block-based execution strategy is adopted to exploit the parallelism of multi cores at each computing node. Matrices are split into sub-matrices called block. All the blocks can be computed in parallel. Note that there are two levels of partitioning in DMac. First, a given matrix is partitioned into blocks and block becomes the base computing unit. When performing computation among cluster, DMac distributes each block based on the partition scheme to the corresponding computing node and performs the local execution independently.

The execution flow inside each computing node is illustrated in Figure 4. First, according to the logic of local operator, the meta data of operations which can be executed independently are packaged into a task, and the task is put into a task queue. Each task will generate a block of the results. Thereafter, multiple threads are created and each one takes a task from the task queue to execute independently.

The input blocks of a single computing node reside in the shared memory, thus they can be read by all the threads. Inside each thread, an In-Place execution approach is designed for matrix operations to improve the utilization of intra-thread memory. A result buffer pool is employed for reusing the inter-thread memory. It maintains a fixed number of blocks in memory. At the beginning of each task inside a thread, it acquires a clean block from the result buffer pool. After the task is finished, the block will be returned to the pool.

In-Place Execution

Usually the matrix operation is also memory-intensive. Take the matrix multiplication as example, which is the most complex but common operation in a matrix program. For a matrix A with $M_A \times N_A$ blocks and a matrix B with $M_B \times N_B$ blocks, the computation of $A \,\%*\%\, B$ will incur $M_A \times N_A \times N_B$ times of block multiplications. Naively parallelizing these block-based computations requires a buffer to store all the intermediate results and aggregate them at last. Such buffer implementation causes expensive memory consumption.

In DMac, we apply an In-Place based implementation. The In-Place approach packages the computations (e.g., block multiplication and aggregation) which contribute to the same result block into a task. Inside each task, the generated results can be directly updated into the result block in-place without extra buffer cost. In other words, no extra memory is needed to allocate the intermediate matrix, and hence reduces the memory cost.

The Choice of Block Size

The size of each block is an important factor and can affect the overall performance significantly. Generally, the determination of block size needs to be hand-tuned according to different input and cluster configuration. DMac automatically determines the size of each block by making a trade-off between the memory consumption and parallelism.

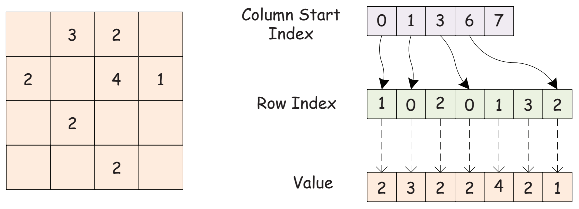


Figure 5: Compress Sparse Column format

In order to reduce the memory consumption, DMac always tends to partition sparse matrix into large blocks. In DMac, sparse block is represented as the compressed sparse column (CSC) format [10], while a one-dimensional array is used for dense block. Due to the sparsity of the original matrix, most blocks are sparse as well. Figure 5 illustrates the CSC format which needs three arrays to represent a sparse matrix. The value array stores the non-zero items while a row index array indicates the row index for each item. The j^{th} element in the Column Start Index array equals the starting point in the row index array for column j. For each sparse block b with $m \times n$ size and s sparsity, the consumed memory can be derived as follows.

$$Mem(b) = \begin{cases} 4n + 8mns & \text{:sparse} \\ 4mn & \text{:dense} \end{cases}$$

When a large matrix is partitioned into multiple blocks, each block needs a Column Start Index array. Considering an $M \times N$ matrix A with sparsity S, the size of row index array and value array are both $M \times N \times S$ regardless of partitioning. However, the size of Column Start Index array is N before partitioning while $\frac{M}{m} \times n$ after partitioning. The total memory consumption for a partitioned matrix can be derived from the following equation

$$Mem(A) = \begin{cases} 4N\frac{M}{m} + 8MNS & \text{:sparse} \\ 4MN & \text{:dense} \end{cases} \tag{2}$$

Therefore, the storage space for a sparse matrix can be decreased when each block is in large size.

However, to fully exploit the parallelism, DMac shall ensure that each thread executes at least one task. According to the characteristic of In-Place based approach, the number of tasks equals to the number of result blocks. Investigating the cell-wise matrix operators, RMM-based matrix multiplication and CPMM-based matrix multiplication, we can figure out that if matrix A with block size m is executed among K workers, the number of total tasks on a worker is at least $\frac{MN}{Km^2}$ which is caused by the RMM-based matrix multiplication. Therefore, the upper bound of row size of each block can be determined by

$$m \leq \sqrt{\frac{MN}{LK}} \tag{3}$$

where L is the local parallelism in a worker. For simplicity, we use square block in DMac, so the column size is set to m as well. Based on this equation, DMac can automatically choose the sufficiently large block (e.g., a value of m is near the upper bound) while fully exploiting the parallel ability of the cluster.

5.4 System Implementation

We proceed to discuss the implementation of DMac. Currently it is implemented as a library of Spark and provides classes, including DenseMatrix and SparseMatrix, which can be directly used to conduct matrix operations for users. All the operators supported in DMac can be direactly called through class methods to perform computation.For the ease of programming, we provide a set of R-Like symbols to represent each matrix operator through the language feature of Scala. Users can call matrix operations in DMac in a similary way comparing to R. Specific program examples are presented in Section 6.4. A sparsity value can be assigned by user for the input matrix or pre-computed before the matrix programs. The size of intermediate matrices are estimated and the statistics is maintained at the driver program of Spark. The execution plan is also generated at the driver program.

DMac utilizes the expression ability of RDD, which is the distributed memory abstraction in Spark, to represent each distributed matrix with items indicating blocks. The basic and extended operators are implemented based on the transformation operations of RDD, like map, groupByKey and reduceByKey. DMac takes advantage of the cache mechanism of Spark to express matrix dependencies, such as reference dependence. We customize the partition interface of Spark with different partition schemes to fulfil the requirement of DMac, i.e., three partition schemes used in DMac are added. Furthermore, the *Extract* and *Transpose* dependencies are expressed as a local transformation operation of RDD.

To implement the In-Place based execution approach, the reduceByKey interface of Spark is modified. In the original implementation of Spark, reduceByKey utilizes an in-memory HashMap to perform a map-side combine operation, which is exactly the buffer based approach. This combine operation induces heavy pressure on memory usage. Since the results generated by the In-Place based approach are already combined in DMac, it does not need any further combine operation. Thus, the map-side combine operation is turned off in DMac.

6. AN EXPERIMENTAL STUDY

In this section, we present an experimental study to evaluate the proposed distributed matrix computation system, DMac. We experimentally demonstrate the efficiency of execution plan, performance on various matrix applications, and the scalability of DMac. The performance is mainly measured by the average execution time and the communication cost for various matrix computation programs.

6.1 Experimental Setup

The experiments are conducted on a cluster with four physical nodes by default, where each node has a 2.6GHz CPU, 48GB memory and a 10TB hard disk. The maximal memory allocation in JVM is set to 32GB. To demonstrate the scalability of DMac with respect to the number of workers, we also test the proposed approach on a 20-node cluster with the same hardware equipment.

Datasets. Our experiments are conducted on both real-world and synthetic datasets. We use five real-world datasets in the evaluation, including Netflix [6], soc-pokec, cit-Patents, LiveJournal and Wikipedia[3], whose detailed meta-data information will be presented in the corresponding experiments. The synthetic datasets are generated by a random data generator which can produce a sparse matrix V with d rows and w columns in s sparsity. Two dense matrices with more than 8 billion non-zero items and 96 million non-zero items respectively are also used in our experiment.

Baseline comparison. Different distributed matrix computation systems have been proposed recently, and SystemML [14, 18] is the latest one among them. Since SystemML is a matrix computing system originally implemented on Hadoop, we migrated SystemML to the Spark platform for the fair comparison, which is denoted as SystemML-S. We implemented the core techniques of SystemML on Spark and utilized the `cache` mechanism provided by Spark to put the intermediate result into memory; thus, SystemML-S can avoid redundant disk access. Furthermore, we deployed the same local execution strategy for both SystemML-S and DMac. In summary, the only difference between SystemML-S and DMac is that SystemML-S generates the execution plan without utilizing matrix dependency while DMac does. Furthermore, two famous distributed matrix computing systems, ScaLAPACK [5] and SciDB [8], are also used in the evaluation.

6.2 Efficiency of Execution Plan

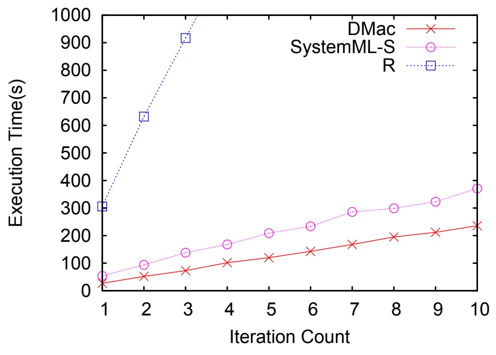
The matrix execution plan is the most important issue for distribute matrix computation systems. In the first experiment, we demonstrate the performance gain contributed by the communication efficient execution plan in DMac. The GNMF algorithm (Code 1) is used as the benchmark. The input matrix V is the Netflix dataset and the factor size for matrix W and H is set to 200, which is a reasonable value for the Netflix dataset.

Figure 6 shows the results including execution time and communication cost of running GNMF on DMac and SystemML-S on the 4-node cluster. In Figure 6(a), we can see that DMac is about 1.6 times faster than SystemML-S in terms of execution time. Meanwhile, both DMac and SystemML-S perform better than the R solution which is one of the most efficient in-memory matrix computation solutions in a single machine. Please note that, the original SystemML is slower than the R [14]. By implementing the matrix computing in Spark and equipping the local execution strategy of DMac, the performance of SystemML-S can be improved in one order of magnitude compared with the original SystemML.

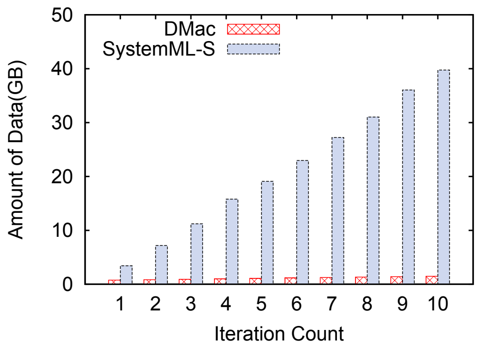
The performance gain of DMac compared with SystemML-S is mainly obtained from our novel communication efficient execution plan. Figure 6(b) shows the detailed communication cost comparison of SystemML-S and DMac. The SystemML-S transfers around 40GB data in the cluster while it is only about 1.5GB for DMac. The cause of significant gap in communication cost between SystemML and DMac is that DMac fully exploits the matrix dependency in the program. In the plan generated by SystemML-S, the matrix dependencies (e.g., **Reference** dependency and **Transpose** dependency) are not taken into consideration. Hence, for each matrix operator, the input matrices have to undergo a repartition phase which increases communication cost. For example, the computation phase of $H\ *\ (W^TV)\ /\ (W^TWH)$ first partitions three input matrices with a hash partitioning method. Thus, blocks with the same row and column index will be sent to the same reducer to perform the cell-wise multiplication and cell-wise division. In contrast, DMac can conduct this computation phase without any communication cost. After the preceding operators in DMac, these three input matrices are already in Column scheme, which can satisfy the requirement of these two cell-wise operators.

For both DMac and SystemML-S, the overall execution time consists of two major parts, i.e., computation and communication. We also investigated the effect of utilizing matrix dependency on these two factors. By analysing the result of execution time in this experiment, we find that the percentage of time costed for communication in SystemML-S is around 44% while it is only 6% in DMac. Since the computation costs are almost the same on DMac and SystemML-S, we can conclude that utilization of matrix dependency in DMac can significantly reduce the communication cost, and hence DMac yields better overall performance.

[3]http://konect.uni-koblenz.de



(a) Accumulated execution time



(b) Accumulated Communication Cost

Figure 6: The performance of running GNMF on Netflix dataset

6.3 Effect of Local Execution Strategy

To process the matrix computation in distributed systems, we also proposed an efficient local execution strategy for each worker in the cluster. Since the matrix computation generally has high memory demand, the block-based strategy is designed to optimize the memory usage and better exploit the parallelism of modern computers at each computing node. We next show the effect of local execution strategy in DMac, i.e., the comparison between two different implementations of the block-based approach, and the influence of memory block size. We use matrix multiplication for this evaluation, since multiplication is a primary and complex operation. Four real-world graphs are used as the input matrices and the meta data of graphs is listed in Table 3.

Graph	Node#	Edge#
soc-pokec	1,632,803	30,622,564
cit-Patents	3,774,768	16,518,978
LiveJournal	4,847,571	68,993,773
Wikipedia	25,942,254	601,038,301

Table 3: Statistics of datasets

In-Place vs. Buffer

Two different implementations of the local block-based approaches are compared. One is the In-Place based implementation mentioned in Section 5.3; the other is a traditional Buffer implementation which parallelizes multiplication of two matrices randomly, buffers the intermediate block in memory and finally performs the aggregation.

Figure 7 illustrates the memory consumption on four graphs. It is clear that the In-Place based implementation is much more efficient than the Buffer implementation. For LiveJournal dataset, the memory cost of Buffer implementation is about 5G more than that of In-Place based implementation. For Wikipedia dataset, the In-Place based implementation can finish the computation with using only 16G memory on each node; while the Buffer method cannot run successfully due to its too high memory requirement. Note that, each node has maximal 48G memory in our cluster. The In-Place based implementation can effectively organize the parallelization of block multiplication, and hence the memory consumption is better controlled. Moreover, for soc-pokec and cit-Patents datasets, which are much sparser compared with the other two graphs, the differences between In-Place based implementation and Buffer implementation are relatively slighter, since the sizes of the intermediate results are smaller.

Influence of block size

As discussed in Section 5.3, the block size affects the overall performance. Figure 8(a) illustrates the execution time comparison for

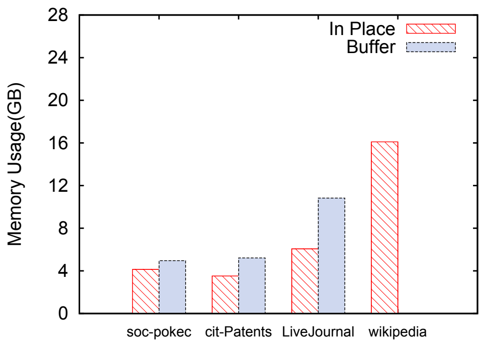


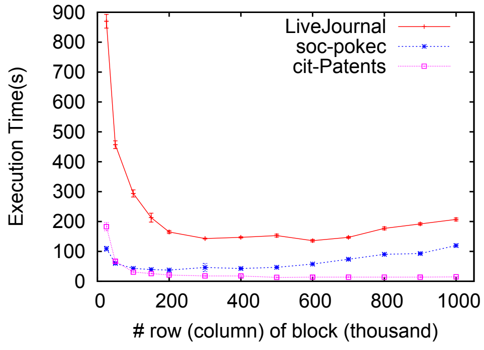
Figure 7: Memory usage

different block sizes on various datasets. First, when the block size is small, the huge memory consumption leads to the poor performance. In LiveJounal datasets, when the block size is smaller than 200k, the execution time becomes larger than 155s. In soc-pokec dataset, the performance becomes worse when the block size is smaller than 50k. The total memory usage on each computing node for different block sizes is shown in Figure 8(b). For LiveJournal, when the block size is close to 10k, it consumes nearly 19G memory while the ideal situation only costs around 6G. The overhead is caused by the duplicated storage of the Column Start Index array when the size of block is small. Generally, with the increasing of block size, the memory cost decreases.

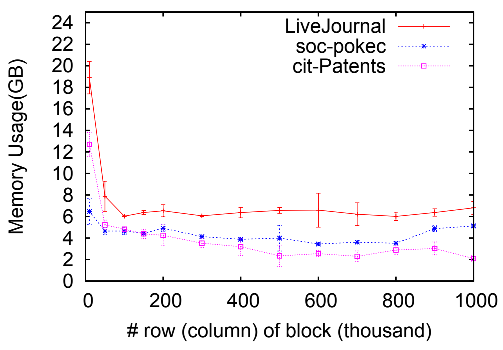
Second, the execution time is increased when the block size is larger than a threshold. This is because when the block size increases, the number of tasks for local parallel execution is decreased. The threshold can be estimated through Equation 3. Here in our 4-node cluster, K is four and L is eight, the threshold is about 856k, 289k and 667k respectively for LiveJournal, soc-pokec and cit-patent. Figure 8(a) shows that when the block size exceeds 800k (LiveJournal) or 300k (sco-pokec), the performances become worse. The slight difference between the actual results in the experiment and estimation of Equation 3 comes from the skewness of datasets. However, the results in Figure 8(a) verify that Equation 3 can guide DMac to choose a reasonable block size. The error bars in these two figures indicate that the trend is not caused by noises.

6.4 Performance on Various Matrix Applications

To demonstrate the robustness of DMac on various matrix computation tasks, we next evaluate the performance on other well-known matrix applications, including PageRank, Linear Regression, Collaborative Filtering and Singular Value Decomposition. We describe how these algorithms can be expressed in matrix pro-

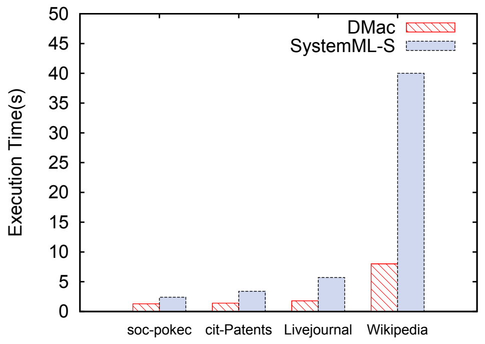


(a) Execution time

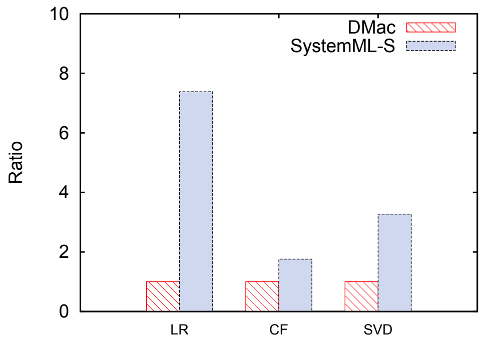


(b) Memory usage

Figure 8: Inference of block size



(a) PageRank



(b) Linear Regression, collaborative filtering and SVD

Figure 9: Performance on various matrix applications

gram and the way they are written in DMac (As shown in Appendix), and present the corresponding results.

PageRank is ran on various datasets in Table 3. As shown in Figure 9(a), DMac consistently performs better than SystemML-S across the datasets. For the Wikipedia dataset, the average execution time per iteration in DMac is approximately 8s while it needs about 40s in SystemML-S. In PageRank, the $link$ matrix will be used in each iteration. DMac can cache the Column scheme of $link$ in memory to avoid redundant communication cost according to the **Reference** dependency. Therefore, only a Broadcast scheme of $rank$ matrix will be sent across the cluster at each iteration, which is usually a small matrix. In SystemML-S, although the $rank$ matrix can also be cached into memory, the partition scheme does not meet the requirement of the `multiply` operator. The above result analysis can also explain why DMac performs better on other three datasets.

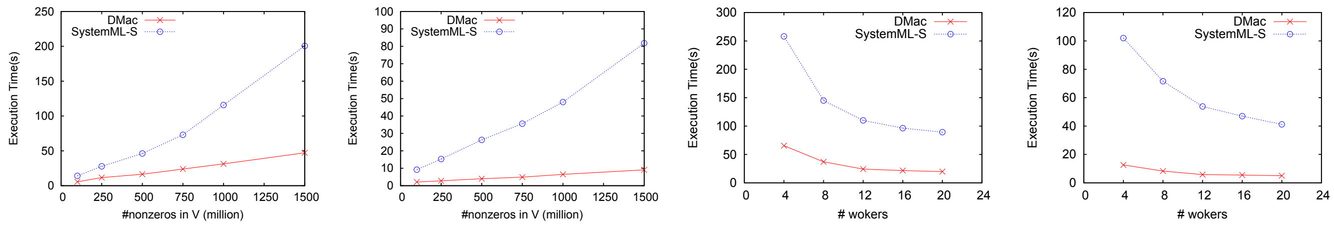
Linear regression is tested with a synthetic $10^8 \times 10^5$ matrix V which contains 1 billion non-zero items, and collaborative filter [7] uses the Netflix dataset which is a standard dataset for the algorithm. The SVD implementation is also evaluated on the Netflix dataset. The performances of these three applications are shown in Figure 9(b). In each application, the execution time has been normalized to the execution time of DMac. For linear regression, DMac outperforms SystemML-S with more than seven times. The execution plan generated by DMac partitions V only once through the whole computation process. However, the cache version of V and V^T in SystemML-S are not satisfied the requirement of partition schemes in the `multiply` operator, and hence they still need to be repartitioned. The core computation of SVD is two `multiply` operators. Similar to Linear Regression, reduntant partition for matrix V would be avoided in DMac. The execution time of SVD in DMac is 291s while it is 954s in SystemML-S. In the collaborative filtering, there are two `multiply` operators. The plans generated from DMac and SystemML-S are both the *RMM* execution strategy for these two `multiply` operators. The total communication cost in DMac is $n \times |R|$ where n is the number of nodes. However, SystemML-S needs to broadcast matrix R twice in each task and partition the intermediate result RR^T, which is a dense matrix with more than 300 million non-zero items. As a result, the execution time of DMac is only 151s while it is 264s for SystemML-S.

6.5 Performance on Scalability

Scalability is an essential feature for distributed matrix computation systems. The following experiments show the high scalability of DMac in terms of the size of datasets and the number of workers.

To obtain datasets in different scale, we generated the synthetic matrices V with fixing the number of columns to 100000 and varying the number of rows. Therefore the number of non-zeros in V varies linearly. This matrix generating process is the same as in [14].

Figure 10(a) and Figure 10(b) show the average execution time per iteration of GNMF algorithm and Linear Regression algorithm with increasing the number of nonzero in the input matrices respectively. From the figures, we can see that with the increase of input matrix size, the gap between SystemML-S and DMac also increases. In the GNMF algorithm, if the size of V increases, the sizes of matrices W, VH^T and WHH^T also increase. In the execution plan generated by SystemML-S, these three matrices need to be repartitioned at each iteration. W will be partitioned four times since there are four references in each iteration while VH^T and WHH^T will be repartitioned once. However, in the execution plan of DMac, W only needs to be partitioned once. In each iteration, only Row scheme of W is required. Column scheme of W^T



(a) GNMF (b) Linear Regression (c) GNMF (d) Linear Regression

Figure 10: Scalability

can be directly generated from Row scheme of W through **Transpose** dependency. Meanwhile, the partition schemes of VH^T and WHH^T generated by the previous operations are also with Row scheme. Thus, no repartition step is needed according to **Reference** dependency. In summary, DMac can obtain a steady performance with the increase of input matrix. For Linear Regression algorithm, SystemML-S needs to repartition the input matrix V twice at each iteration. The communication cost increases sharply when V becomes large. In the execution plan generated by DMac, the input matrix V only needs to be partitioned once through the whole computation process. Thus, with the increasing input size, the communication cost in DMac can be well-controlled, and hence DMac yields better scalability.

GNMF and linear regression are next ran on a sparse matrix with 2 billion non-zero items to demonstrate the scalability of DMac with respect to the number of workers. The results are shown in Figure 10(c) and Figure 10(d). We can clearly see that the performance of DMac is improved gradually with the increasing number of workers from 4 to 20. For example, in GNMF algorithm, the execution time is about 65s per iteration when there are four workers; while it can be reduced to 20s when the number of worker increases to 20, achieving a speedup of 325%.

6.6 Comparison with other systems

We further present the experimental results of comparing DMac with ScaLAPACK and SciDB. ScaLAPACK [5] is a library of linear algebra routines for distributed memory computers and provides a series of interfaces for distributed matrix computation. SciDB [8] is an open-source data management system orienting for large (petabyte) scale array data. A linear algebra library in SciDB accepts matrix as inputs and provides functionality for matrix operators (e.g., inverse, multiplication, transpose and SVD).

Matrix multiplication is a very common and primitive operation, and many graph algorithms can be represented through it. Therefore, we use the matrix multiplication to conduct the following comparison. Two datasets with different sparsity are used. The first one contains two matrices V_1 and H^T. Matrix V_1 is constructed from the Netflix dataset and H is a dense matrix with 480189 rows and 200 columns. The sparsity of V_1 and H is 0.01 and 1 respectively. In the second dataset, the matrix V_1 is replaced by a matrix V_2 which has the same dimension characteristics with V_1 but has the sparsity of 1. ScaLAPACK, SciDB, SystemML-S and DMac are all running on a 8-node cluster and each node launches eight processes (instances) for these systems.

	ScaLAPACK	SciDB	SystemML-S	DMac
MM-Sparse	107s	11m35s	18.5s	17s
MM-Dense	116s	12m15s	133s	121s

Table 4: Comparision of ScaLAPACK, SciDB, SystemML-S and DMac

The "MM-Sparse" row in Table 4 shows the results of multiplying V_1 and H. The performance of DMac is significantly better than those of ScaLAPACK and SciDB, because DMac can efficiently support the operators processing both sparse and dense matrices. Both ScaLAPACK and SciDB are not well tuned for sparse matrices [4], and they handle the sparse matrix as the way on dense one. The result of multiplying V_2 and H is presented in the "MM-Dense" row. Comparing with the result of the sparse matrix, there is nearly no much difference in ScaLAPACK and SciDB, while DMac needs more time to handle a dense matrix. However, the performance of DMac is still comparable with ScaLAPACK which is a highly tuned library for matrix computation. The reason is as follows. First, DMac adopts the one-dimensional partitioning methods (e.g., row or column), which can benefit the matrix multiplication on MapReduce like systems. No communication will be invoked after the data shuffling is completed. Second, The local execution method in DMac adopts a shared memory architecture and each thread can read the input data simultaneously without any interference. Since ScaLAPACK is built on top of MPI, processes communicate with each other through messages. Hence, multiple processes will be created on a single node and data is transferred through messages instead of share memory. SciDB is a complete data management system aiming at scientific computation. It takes the responsibility of data partition and query processing. Before performing matrix operations, SciDB needs to redistribute the data on each computing node to satisfy the requirement of ScaLAPACK. Meanwhile, SciDB maintains a failure handling mechanism during the computation, which introduces extra overhead. Hence, it costs more time to complete the operation comparing to ScaLAPACK. When performing only a multiplication operator, the improvement of DMac against SystemML-S is not significant as earlier evaluations, because they share the same local execution method and the total communication cost is the same.

7. RELATED WORK

MapReduce has emerged as a popular programming model for parallel data processing. It simplifies the development of distributed parallel application for programmers. Hadoop [2] and Spark are two popular open source implementations of MapReduce computation model. However, the MapReduce interface is fundamentally restrictive. It is difficult to write an efficient program for a real-world matrix application. Therefore, different distributed matrix computation systems have been proposed based on the general large scale data processing platforms. Pegasus [15] is a Hadoop-based library that implements a class of graph mining algorithms which can be expressed via repeated matrix-vector multiplications. However, only considering a single primitive is not enough. SystemML [14], HAMA [21], Mahout [4] and MadLINQ [20], have provided a general interface for large-scale matrix computations. MadLINQ utilizes a fine-grained pipelining execution model to aggressively explore the inter-vertex parallelism. But, using pipeline obtains slight performance gain since it cannot decrease the total

[4]http://acts.nersc.gov/scalapack/

amount of communication data. SystemML has provided an R-like matrix interface language, which helps users escape from hand-coding MapReduce programs. SystemML automatically translates user-written program into a series of MapReduce jobs running on Hadoop. A cost model is utilized in SystemML to determine the execution strategy for matrix multiplication. A hybrid parallelism execution strategy is also proposed in SystemML to get comparable performance with in-memory systems [18]. Compared with these large-scale matrix computation systems, DMac novelly exploits the dependencies in the matrix programs to construct a communication efficient execution plan.

Matrix computation has also been a focus area in the HPC community for many years. BLAS [1] and LAPACK [3] have provided a series of routines to efficiently support matrix operators. ScaLAPACK [5] is the distributed variant to parallelize the computations among multiple computing nodes. Although ScaLAPACK is highly expressive, it is complicated to program a complex matrix application. Meanwhile, it is the users' responsibility to partition the matrix into blocks and place them on each working node. SciDB is an open-source data management system for scientific application. SciDB [8] adopts an array data model and has provided a great query language which is friendly to matrix computation. However, it still treats each matrix operator independently and is lack of the ability to reduce communication cost for a matrix application.

Next we review the work on partition schemes for matrix computations. Various partitioning methods have been adopted to perform distributed matrix computation. Chunk-based partitioning is used in SciDB and block-cyclic partitioning is employed in ScaLAPACK. These two partitioning methods can be classified as two-dimensional partitioning method, while the row and column partitioning adopted in DMac and SystemML can be regarded as one-dimensional partitioning method. Two-dimensional partitioning method produces a more balance partition while one-dimensional partitioning can reduce the number of aggregation during the computation. DMac adopts the one-dimensional partitioning methods since it is more suitable for matrix multiplication on MapReduce-like systems.

At last, we briefly discuss the works have already laid foundations for communication optimization in distributed computing environment, which include data reusing via a loop-aware scheduling method for iterative application [9], transformation-based optimizer to avoid repartitioning operations [24], cost based optimization for workflows [19, 17]. These optimization methods are designed to handle the general parallel programs, and cannot be utilized to support the matrix operators with dependency in distributed matrix computation.

8. CONCLUSION

In this paper, we have developed a novel matrix computation system (DMac) for efficient matrix computation in the distributed environment. We identified a new concept of matrix dependency which can capture the reference relationship between operators in the matrix program. Unfortunately, the dependency information is ignored in the existing matrix computation systems, which results in frequent repartition process and heavy communication. The exploitation of matrix dependency can help us choose execution strategy with minimal communication for each operator. Consequently, we can generate a communication efficient execution plan for the whole matrix computation task. To accelerate the processing in distributed systems, we further divided the execution plan into multiple un-interleaved stages which are then run in a cluster with efficient local execution strategy on each worker. The DMac system has been implemented on a popular general-purpose data processing framework, Spark. The extensive empirical studies on various matrix programs demonstrate the superiority of our approach compared with existing methods.

ACKNOWLEDGMENTS

The research is supported by the National Natural Science Foundation of China under Grant No. 61272155 and 61472009, 973 program under No. 2014CB340405, and Seeding Grant of Peking University 2014-MI-09.

9. REFERENCES

[1] *BLAS*. http://www.netlib.org/blas/.
[2] Hadoop. http://hadoop.apache.org/.
[3] *LAPACK*. http://www.netlib.org/lapack/.
[4] *Mahout*. https://mahout.apache.org/.
[5] *ScaLAPACK*. http://www.netlib.org/scalapack/.
[6] J. Bennett and S. Lanning. The netflix prize. In *KDD Cup'2007*, pages 35–35.
[7] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI'1998*, pages 43–52.
[8] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD'2010*, pages 963–968.
[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *VLDB'2010*, 3(1-2):285–296.
[10] T. A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. 2006.
[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm. ACM*, 51(1):107–113, 2008.
[12] M. L. Eaton and I. Olkin. Best equivariant estimators of a cholesky decomposition. *The Annals of Statistics*, 15(4):1639–1650, 12 1987.
[13] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, Mar. 1977.
[14] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE'2011*, pages 231–242.
[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM'2009*, pages 229–238.
[16] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.
[17] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *VLDB'2012*, 5(11):1196–1207.
[18] B. Matthias, T. Shirish, R. Berthold, S. Prithviraj, T. Yuanyuan, R. B. Douglas, and V. Shivakumar. Hybrid parallelization strategies for large-scale machine learning in systemml. In *PVLDB'2014*, pages 553–564.
[19] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: sharing across multiple queries in mapreduce. *VLDB'2010*, 3(1-2):494–505.
[20] 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'12*, pages 197–210, 2012.
[21] 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' 2010*, pages 721–726, 2010.
[22] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'2008*, volume 8, pages 1–14.
[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud'2010*, pages 10–10.
[24] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE'2010*, pages 1060–1071.

APPENDIX

A. MATRIX PROGRAMS OF VARIOUS APPLICATIONS IN DMAC

A.1 Program of PageRank

Code 2 shows the matrix program of PageRank algorithm. The $link$ is a row-normalized $N \times N$ adjacent matrix generated from the corresponding network while $rank$ is a random vector (i.e., $1 \times N$ matrix) representing the initial PageRank values for each node. The main logic of PageRank algorithm can be represented by the multiplication operation of $rank$ and $link$.

```
val max_iteration = 10
val link = load(path = ...)
var rank = new RandomMatrix(1, N)
for ( i <- 0 until max_iteration) {
   rank = (rank %*% link) * 0.85 + D * 0.15
}
```

Code 2: PageRank algorithm

A.2 Program of Collaborative Filtering

The collaborative filtering algorithm is depicted in Code 3. Input matrix R records the ratings between users and items with $R[i, j]$ indicating the rating between user j and item i. An item similarity matrix can be obtained by computing $R \% {*} \% R.t$. The predicted ratings of all items for each user are obtained by multiplying the similarity matrix with R. A normalization step is needed at last to get the final predict value for each user-item pair.

```
val R = load(path = ...)
val result = R %*% R.t %*% R
val predict = result.normalize
```

Code 3: Collaborative Filtering algorithm

A.3 Program of Linear Regression

Code 4 specifies an implementation of a conjugate gradient algorithm for linear regression problem. Each row in the input matrix V represents a training data points in a high-dimensional, sparse feature space. y is a vector with $10^5 \times 1$ size, indicating the target label for each training data. w is the output vector which has the parameters of the regression model.

```
val max_iteration = 10
val V = load(path = ...)
val y = load(path = ...)
var w = new RandomMatrix(100000, 1)
val lambda = 0.000001
val r = (V.t %*% y) * -1
var p = r * -1
var norm_r2 = (r * r).sum

for (i <- 0 until max_iteration) {
   val q = (V.t %*% (V %*% p)) + p * lambda)
   val alpha = norm_r2 / (p.t %*% q).value
   w = w + (p * alpha)
   val old_norm_r2 = norm_r2
   r = r + (q * alpha)
   norm_r2 = (r * r).sum
   val beta = norm_r2 / old_norm_r2
   p = (r * -1) + (p * beta)
}
```

Code 4: LinearRegression algorithm

A.4 Program of Singular Value Decomposition

Code 5 shows the implementation of distributed SVD for large matrices using the Lanczos algorithm[5]. Matrix V is the one which needed to be decomposed and $rank$ is the desired rank of the approximation for singular values. Through the iterative computation, a tridiagonal matrix will be built and the singular values are derived from the local tridiagonal matrix.

```
val V = load(path = ...)
val rank = 100
var vc = new RandomMatrix(n, 1)
var vp = new RandomMatrix(n, 1, 0)
val triDiag = new LocalDenseMatrix(rank, rank)
var beta = 0

for (i <- 0 until rank) {
   var w = V.t %*% (V %*% vc)
   val alpha = (vp.t %*% w).value
   w = w - (vp * beta)
   w = w - (vc * alpha)
   beta = v.norm(2)
   vp = w
   vc = vp
   triDiag[i][i] = alpha
   if (i > 0) {
      triDiag[i-1][i] = beta;
      triDiag[i][i-1] = beta;
   }
}
triDiag.computeSingularValue()
```

Code 5: SVD algorithm

[5]http://en.wikipedia.org/wiki/Lanczos_algorithm