Rima: An RDMA-Accelerated Model-Parallelized Solution to Large-Scale Matrix Factorization

Preprint · August 2018

CITATIONS
0

READS
143

1 author:

Jinkun Geng
Tsinghua University
30 PUBLICATIONS **16** CITATIONS

SEE PROFILE

Rima: An RDMA-Accelerated Model-Parallelized Solution to Large-Scale Matrix Factorization

***Abstract*—Matrix factorization (MF) is a fundamental technique in machine learning and data mining, which gains wide application in many fields. When the matrix becomes large, MF cannot be processed on a single machine. Considering this, many distributed SGD algorithms (e.g. DSGD) have been developed to solve large-scale MF on multiple machines in a model-parallel way. Existing distributed algorithms are primarily implemented under Map/Reduce or PS (parameter server)-based architectures, which incur significant communication overheads. Besides, existing solutions cannot well embrace the benefit of RDMA/RoCE transport and suffer from scalability problems. Targeting at these drawbacks, we propose Rima, which uses ring-based model parallelism to solve large-scale MF with higher communication efficiency. Compared with PS-based SGD algorithms, Rima also consumes less queue pairs (QPs) and can thus better leverage the power of RDMA/RoCE to accelerate the training speed. Our experiment shows that, compared with PS-based DSGD when solving $1M \times 1M$ MF, Rima achieves comparable convergence performance after equal number of iterations, but reduces the training time by 68.7% and 85.4% via TCP and RDMA respectively.**

***Index Terms*—matrix factorization, SGD, RDMA, communication efficiency**

I. Introduction

Matrix factorization (MF) is a typical technique in machine learning and data mining, which has been widely applied in many areas, including personalized recommendation [1] [2] [3] [4] [5], social network analysis [6] [7] [8] [9], web mining [10] [11] [12], bioinformation analysis [13] [14] [15] and so on.

For a matrix R, MF tries to find two other matrices P and Q (much smaller than R) to approximate R (i.e. $PQ \approx R$). Among the efficient algorithms for MF, stochastic gradient decent (SGD) gains the most attention, especially due to its success in KDD Cup 2011 [16] and Netflix Competition [17]. When the input matrix R grows very large, MF cannot be handled by a single machine, thus necessitating distributed algorithms to solve the problem by multiple machines in a model-parallel way. Distributed SGD (DSGD) [18] and its variants (e.g. HogWild! [19], FPSGD [20], NOMAD [21], MLGF-MF [22], etc) are developed to execute the factorization.

Existing distributed algorithms for MF are primarily implemented under Map/Reduce [18], [22], [23] or PS-based architectures [24]–[28]. Between them, PS-based architecture proves to outperform Map/Reduce for iterative model-parallel algorithms [27], [29]. For MF under Map/Reduce architecture, *shuffle* and *reduce* operations are called during each iteration, generating significant communication overheads and disk I/O overheads [27]. However, PS-based architecture also has distinct drawbacks, which we argue as follows.

Firstly, PS-based architecture constrains the performance of RDMA in large scale. Under PS-based architecture, high computation capability can be achieved via multi-core CPUs or GPUs, but communication cost remains as the major overheads [27]–[31]. Recent years have witnessed the rapid development of RDMA/RoCE [32], [33], which significantly reduces the transport time compared with TCP, by implementing the networking protocol on NICs and bypassing the kernel stack. However, PS-based architecture cannot well embrace the benefit of RDMA, because the incast communication causes large consumption of queue pairs (QPs) on the PS. For a PS-based system with w workers, the PS needs to establish w reliable connections with them. Too many QPs can cause more cache misses in the hardware and drop down the performance of RDMA [34], [35]. Moreover, the incast communication under PS-based architecture may bring other RDMA-related problems, such as head-of-the-line blocking [33], [36] and spreading congestion [36]. Such scalability problems seriously hurt PS-based solution to large-scale MF when using RDMA.

Secondly, the distributed algorithms under PS-based architecture also incur more traffic redundancy. As what will be illustrated in Section II, during each iteration, PS will firstly partition R into multiple blocks and distribute one block for each worker to execute SGD on it. After the computation, each worker gains the updates for some entries in P and Q, and then it needs to *push* the whole updates to the PS and waits for the PS to again distribute one block to it. We will show that it is not necessary to *push* all the updates to the PS and thus there can be considerable workload reduction.

Targeting at the drawbacks of PS-based solutions for MF, we design Rima, which adopts ring-based architecture and reduces the traffic redundancy in SGD-based MF. Compared with existing works, Rima enjoys three main advantages to achieve faster MF.

1) Rima executes with ring-based model parallelism, and each server only requires to maintain two connections with its two neighbors respectively. In this way, the QP consumption is largely reduced and the performance of RDMA will not drop down in larger-scale MF.
2) Rima elaborately assigns R blocks to workers and halves the transmission workload, thus reducing the communication time among servers and accelerating the training process for MF.
3) Rima also better overlaps computation/communication with disk I/O. When R is very large and cannot be

wholly read into the memory, Rima can read the proper block of R in advance, thus reducing the overheads caused by disk-based operations.

The remaining part of this paper is organized as follows. Section II illuminates the background and motivation of our work. Section III describes the design details of Rima. Section IV presents the experiment evaluation of Rima with both real data set and synthetic data set. Section V summarizes the related work and Section VI concludes the paper.

II. Background and Motivation

A. SGD to solve MF

The problem of MF can be formulated as follows: Given an $m \times n$ matrix R, MF aims to use two matrices P and Q to approximate R, with P as an $m \times k$ matrix and Q as a $k \times n$ matrix ($k << m$ and $k << n$), i.e.

$$R \approx PQ \tag{1}$$

More specifically, it can be formulated as

$$r_{ij} = \sum_{u=0}^{k-1} p_{iu} q_{uj} \tag{2}$$

SGD tries to find the proper P and Q with an iterative algorithm. During each iteration, SGD will calculate the gradient for each entry in R and use these gradients to update P and Q. With the initial value of P and Q, SGD iteratively optimizes the current solution with gradient descent and calculates new P and Q, which can better approximate R. During each iteration, SGD will execute three main steps described as follows.

1) Firstly, the error between the current solution and the ground truth can be calculated.

$$e_{ij}^2 = (r_{ij} - \sum_{u=0}^{k-1} p_{iu} q_{uj})^2 \tag{3}$$

2) Then, the gradient for each p_{iu} and q_{uj} can be obtained.

$$\frac{\partial e_{ij}^2}{\partial p_{iu}} = -2 e_{ij} q_{uj} \tag{4}$$

$$\frac{\partial e_{ij}^2}{\partial q_{uj}} = -2 e_{ij} p_{iu} \tag{5}$$

3) Finally, the current P and Q will be updated for a better solution.

$$p'_{iu} = p_{iu} + \eta(2 e_{ij} q_{uj} - \lambda p_{iu}) \tag{6}$$

$$q'_{uj} = q_{uj} + \eta(2 e_{ij} p_{iu} - \lambda q_{uj}) \tag{7}$$

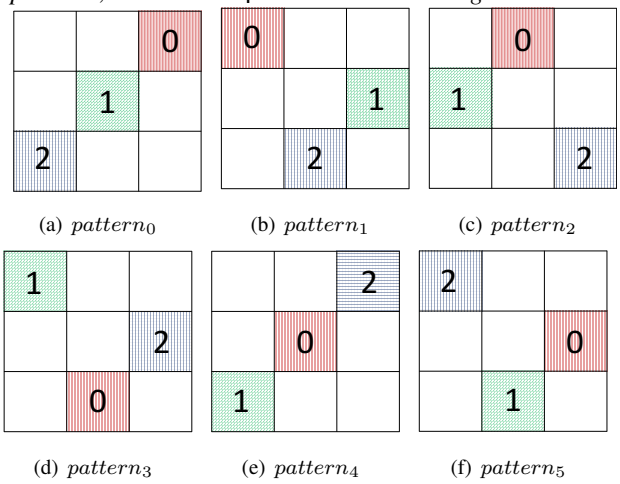
η and λ are hyper-parameters predefined in SGD. η is called *learning rate*; λp_{iu} and λq_{uj} are called *regularization items* to avoid over-fitting.

After the iteration, p'_{iu} and q'_{ui} are calculated based on p_{iu} and q_{ui}, and compose new P and Q for the next iteration. SGD keeps optimizing P and Q in this way until the solution converges or the threshold condition is reached.

B. DSGD to solve Large-scale MF

When R is very large, the training process is very time-consuming and the storage exceeds the memory capacity on a single machine, thus necessitating distributed algorithms to partition the workload. DSGD [18] is one of the representative algorithms to solve large-scale MF in a distributed way. Following the main idea of DSGD, many variants are also proposed [20]–[22] to enhance the original algorithm in different aspects. Since these distributed algorithms work in a similar way, under Map/Reduce or PS-based architectures, we only describe the workflow of DSGD in this section, through which the drawbacks of DSGD and its variants can be better illustrated.

As shown in Section II-A, given a rating entry in R (i.e. r_{ij}), the ith row of P (i.e. p_{i*}) and the jth column of Q (i.e. q_{*j}), SGD can calculate the updates for p_{i*} and q_{*j} without other rows or columns. Considering this, DSGD firstly partitions the whole matrix R into $w \times w$ blocks (w equals to the number of workers). Then, DSGD chooses w *interchangeable* blocks to form a *pattern*. We say two blocks are *interchangeable* if they do not share the same rows or columns. Figure 1 illustrates six *patterns*, each is composed of 3 *interchangeable* blocks.



(a) $pattern_0$ (b) $pattern_1$ (c) $pattern_2$

(d) $pattern_3$ (e) $pattern_4$ (f) $pattern_5$

Fig. 1: 6 *patterns* in 3×3 block partition

The algorithm of DSGD can be described as Algorithm 1. Under PS-based architecture, the PS generates *pattern*s to cover all blocks in R. During each iteration, the PS chooses one *pattern* and assigns different blocks in this *pattern* to workers. Then each worker *pulls* from PS the corresponding rows in P and columns in Q, and calculates updates for P and Q. Afterwards, the updates are *pushed* to the PS and the worker waits for the PS to assign new blocks to continue the next iteration.

An illustrative example is presented in Figure 2. There are 3 workers involved in the PS-based system, thus R is partitioned into 3×3 blocks (denoted as $R_{0,0} \sim R_{2,2}$) and 3 *interchangeable* blocks form a *pattern*. Correspondingly, P is partitioned into 3 blocks by row (denoted as P_0^0, P_1^0, and P_2^0) and Q is partitioned into 3 blocks by column (denoted as Q_0^0, Q_1^0, and Q_2^0) (refer to Figure 2(a)). During the iteration, the PS picks one *pattern* and assigns the blocks to workers. Then, the P blocks and Q blocks are sent to workers based

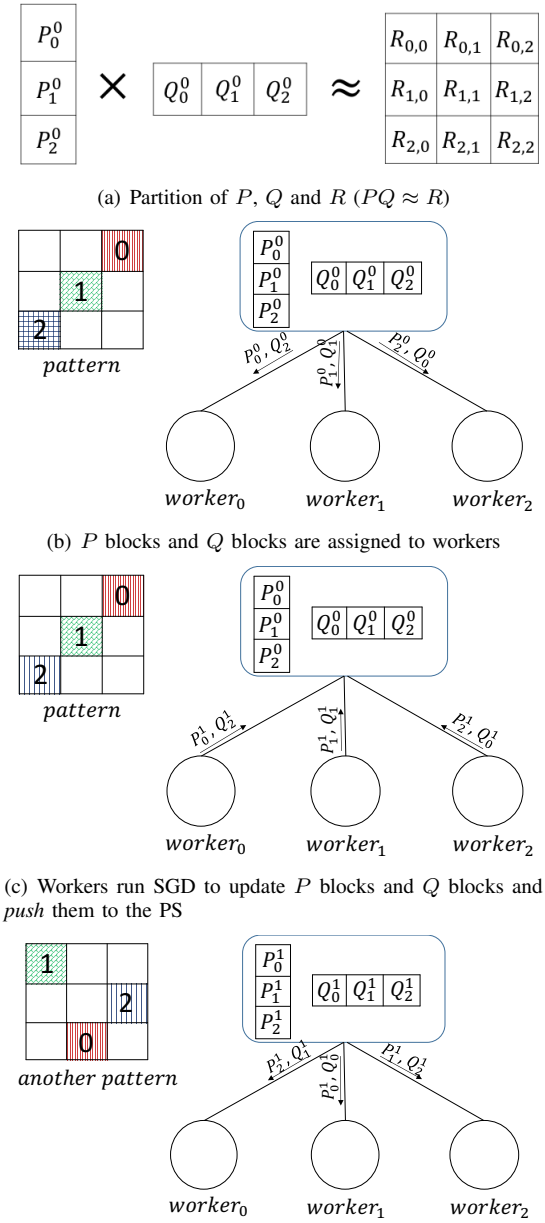
Algorithm 1: DSGD for MF under PS-based Architecture

Input:
w: The number of workers
m: The row number of R
n: The column number of R
k: The column number of P (i.e. the row number of Q)
T: The iteration number of DSGD

```
DSGD()
    PS initializes P and Q
    PS partitions R into w × w blocks of size m/w × n/w each
    Correspondingly, PS also partitions P (by row) and Q (by column) into w blocks
    PS generates s patterns with each one containing w interchangeable blocks
    for t ← 0 to T − 1 do
        PS chooses one pattern randomly from the pattern set
        PS assigns each worker with a different block in this pattern
        foreach worker_i do
            Pull the relevant P block and Q block from PS according to the R block assigned
            Read the assigned R block from memory or disk (when R cannot be held in memory )
            Execute SGD with the R block, P block and Q block
            Update P block and Q block according to Equation 6 and 7
            Push updated P block and Q block to PS
        end
    end
```

on the R blocks assigned (refer to Figure 2(b)). For example, $worker_0$ is assigned $R_{0,2}$, so it obtains P_0^0 and Q_2^0 from the PS. Similarly, $worker_1$ obtains P_1^0 and Q_1^0; $worker_2$ obtains P_2^0 and Q_0^0. Then each worker reads the rating entries of R blocks from its memory or disk, and executes SGD to update the P block and Q block. After the threshold condition is reached (e.g. each worker runs SGD with a certain number of rating entries sampled from the assigned R block), the 3 workers *push* their updated P blocks (denoted as P_0^1, P_1^1, and P_2^1) and Q blocks (denoted as Q_0^1, Q_1^1, and Q_2^1) to the PS (refer to Figure 2(c)). The PS merges the updated blocks with its local copy and completes one iteration of DSGD. Following this way, the PS again chooses another *pattern* and assigns blocks to workers for the next iteration. The updated P blocks and Q blocks from the prior iteration are redistributed to workers for further updates (refer to Figure 2(d)).

Besides DSGD, many variants are also proposed in previous works to further optimize the algorithm performance. For instance, FPSGD [20] partitions R into more blocks (say $(w+1) \times (w+1)$) to overcome locking problem. LazyTable [24] leverages Stale Synchronous Parallel model to



(a) Partition of P, Q and R ($PQ \approx R$)

(b) P blocks and Q blocks are assigned to workers

(c) Workers run SGD to update P blocks and Q blocks and *push* them to the PS

(d) PS chooses another *pattern* and starts the next iteration

Fig. 2: An illustrative example of DSGD under PS-based architecture

mitigate transient straggler effect [1]. MLGF-MF [26] partitions R into blocks of different sizes to improve the robustness to skewed matrix. These works are orthogonal to Rima and their strategies can be further integrated in Rima for enhancement. However, all these works fail to remedy the drawbacks of DSGD under PS-based architecture.

C. Drawbacks of DSGD

The drawbacks of DSGD can be summarized into four main aspects.

First, the communication cost can be expensive in PS-based systems because P and Q can also become very large, though

[1]Transient straggler problem can be considered as a special case of locking problem.

they are smaller than R. Thus the transmission will cause serious overheads and there may be network bottlenecks in the PS-based architecture [30].

Second, the PS suffers from significant consumption of working memory in large-scale MF. When there are more workers than PSes, workers keep *pushing* P blocks and Q blocks to PSes, thus PSes have to allocate sufficient working memory to hold these coming blocks. Such overheads are especially evident in *one-sided* RDMA, where the memory is registered in advance for remote read/write. With the ratio of $1 : w$ between PSes and workers, one PS needs to allocate at least w times of working memory as large as that of one worker to avoid read-write conflict or write-write conflict.

Third, the disk I/O overheads may drop down the performance. When R is very large and cannot be completely loaded into memory, workers have to read the necessary data from disk during each iteration. Since the partition and assignment of blocks are managed by the PS, the worker is unaware which block to read from disk before it receives the information from the PS. Therefore, the proper disk read operation cannot be pre-executed, or there can be lots of cache misses in the prefetched data without prior knowledge. Note that MLGF-MF [22] proposes an asynchronous I/O request mechanism to pre-read the data, however, such mechanism incurs more communication with the PS and adds more scheduling complexity. Besides, the asynchronous I/O requests from workers may cause potential conflicts, and two workers may ask for the same free block from the PS because they are oblivious to the situation of each other. Therefore, the existing works cannot reduce the cost of disk I/O without introducing additional communication overheads or conflict risk.

More importantly, RDMA/RoCE, which is considered as a promising technology to reduce communication cost, cannot be well harnessed by the PS-based architecture. RDMA relies on Queue Pairs (QPs) for data transmission. Some QP-related data, such as memory translation table (MTT), QP states and work queue elements (WQE), have to be cached in NIC memory [34], [37]. Too few QPs may limit the NIC parallelism [37], but too many QPs can run out of the cache space, thus incurring cache penalty and slow-receiver symptom [33], [34], [37]. Previous studies [35] have observed that there is a serious performance decline with Mellanox ConnectX-3 NICs as the server maintains more than 100 QPs. In PS-based systems, reliable connection is desirable, thus QPs are required to communicate exclusively and cannot be shared among connections [38]. On the other hand, each PS needs to establish connections with all workers, thus there is a large consumption of QPs on the PS, which greatly constrains the performance of RDMA.

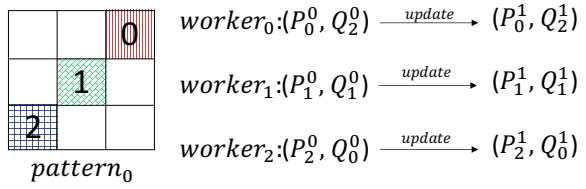
Considering the drawbacks of PS-based solutions to large-scale MF, we seek to execute MF in a decentralized way to eliminate the potential bottlenecks introduced by the PS. Meanwhile, we integrate ring-based model parallelism in Rima to reduce the communication workload and the number of connections for each server. In this way, the power of RDMA can be better leveraged to accelerate large-scale MF.

III. Design of Rima

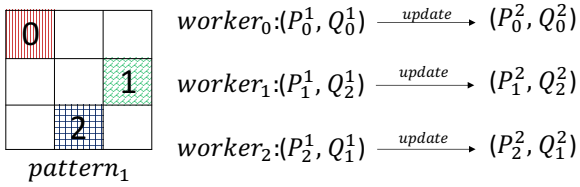
A. Halving Transmission Workload in Rima

Looking back on DSGD algorithm, we discover that some P blocks and Q blocks can be reused locally instead of *pulling* from the PS. As shown in Figure 1, each *pattern* can be generated by moving the prior *pattern* for one-block distance. For example, $pattern_1$ can be generated by moving $pattern_0$ rightwards for one-block distance; $pattern_3$ can be generated by moving $pattern_2$ upwards for one-block distance, and so on. We call this operation *one-step transformation*.

For each two consecutive iterations, if the *pattern* in the latter iteration can be generated via *one-step transformation* from the *pattern* in the former iteration, then P block or Q block updated in the former iteration can be reserved locally for the computation in the next iteration.



(a) Iteration 0



(b) Iteration 1

Fig. 3: Blocks obtained and updated for each worker

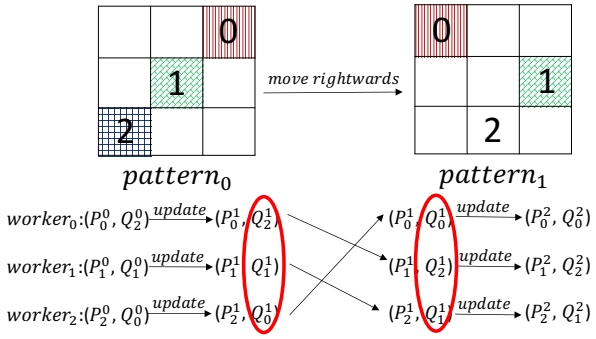
We use Figure 3 to illustrate the property. During Iteration 0, $worker_0$ is assigned $R_{0,2}$, so it runs SGD to update P_0^0 and Q_2^0, and outputs P_0^1 and Q_2^1. However, it does not need to *push* both P_0^1 and Q_2^1 to the PS, because it is assigned $R_{0,0}$ in the next iteration. In Iteration 1, $worker_0$ is responsible for updating P_0^1 and Q_0^1. P_0^1 can be obtained locally from $worker_0$ and other workers do not need P_0^1. Therefore, P_0^1 can be retained by $worker_0$ and only Q_2^1 needs to be *pushed* to the PS after $worker_0$ completes Iteration 0. The situation is similar for other workers: $worker_1$ only needs to transfer Q_1^1 and $worker_2$ only needs to transfer Q_0^1.

More generally, if the sequence order of *patterns* is specially designed, workers can always retain one block and transfer the other block for each iteration. For example, if the *pattern* sequence is designed as shown in Figure 1, then during each iteration, either P block or Q block is required to be sent and the other block can be retained by the worker. When $m \approx n$, P block and Q block have similar sizes and the retention of one block can help to reduce half of workload to transfer.

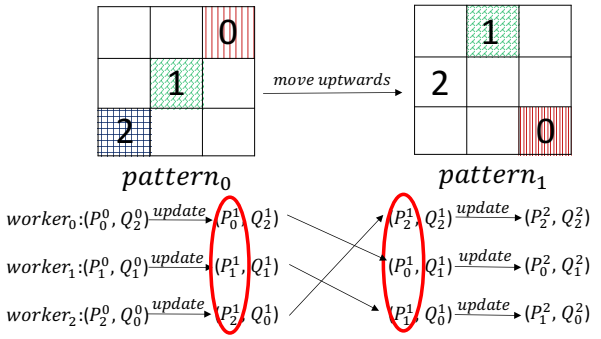
B. Ring-based Parallelism in Rima

It is worth noting that during each iteration, the block transferred by each worker is only needed by another fixed worker if the *pattern* only moves rightwards or upwards. As

shown in Figure 4, if the *pattern* moves rightwards, then $worker_0$ only needs to transfer Q block to $worker_1$; if the *pattern* moves upwards, then $worker_0$ only needs to transfer P block to $worker_1$ [2].



(a) Pattern moves rightwards, Q blocks are transferred



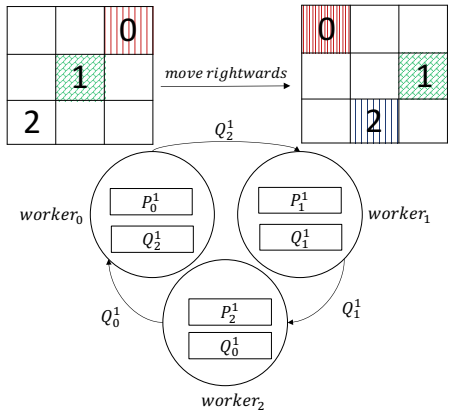
(b) Pattern moves upwards, P blocks are transferred

Fig. 4: Pattern move and block transfer

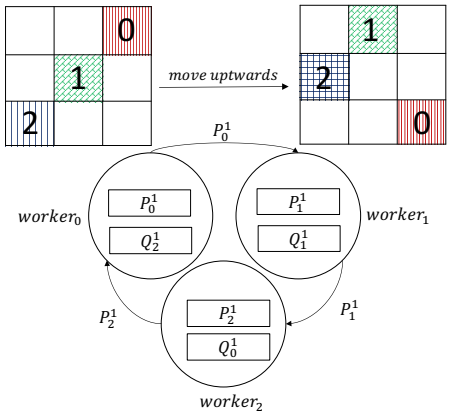
Since each worker only sends blocks to one fixed destination worker, if the *pattern* sequence is generated with *one-step transformation* towards certain directions, then there is no need to keep the role of PS for transferring blocks. As a result, we can eliminate the PS and run SGD with ring-based model parallelism. As shown in Figure 5, the three workers can be organized as a logic ring, and each worker has one predecessor and one successor. Take $worker_0$ as an example, during each iteration, it receives one P block or Q block from its predecessor (i.e. $worker_2$) and also sends one updated P block or Q block to its successor (i.e. $worker_1$). After a certain number of iterations and the threshold condition is reached, the $w-1$ workers send their P blocks and Q blocks to the other worker to form the final solution.

Compared with PS-based systems, Rima can reduce half transmission workload and it is also free from centralized network bottleneck. On the other hand, PS-based systems may suffer from scalability problem for RDMA when the number of worker increases or the QP consumption on each worker increases. By contrast, Rima can better embrace the benefit of RDMA due to two main reasons: First, Rima establishes only two connections on each server whereas PS-based systems require at least w connections on the PS. Correspondingly, to use reliable connection of RDMA for data transmission,

[2]Surely we can choose to move leftwards or downwards, but in that case, $worker_0$ only needs to transfer P block or Q block to $worker_2$.



(a) Pattern moves rightwards, and only Q blocks are transferred in the logic ring



(b) Pattern moves upwards, and only P blocks are transferred in the logic ring

Fig. 5: Ring-based model parallelism in Rima

PS-based systems require at least w QPs for each PS whereas Rima only requires two for each server. Second, PS needs to maintain w times of working memory as large as that maintained by each server in Rima. With RDMA, the PS needs to register much more memory in advance and the memory cost grows as the worker number increases. By contrast, the working memory of Rima does not grow with the number of worker and it keeps linear to the size of P blocks and Q blocks.

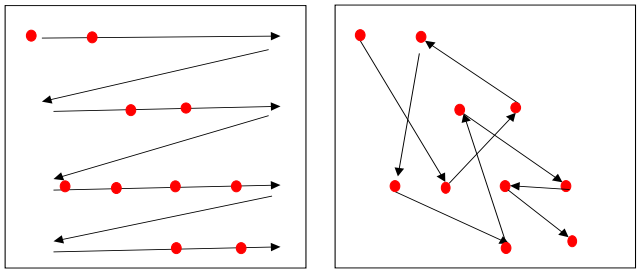
C. Partial Randomness in Rima

Since the *pattern* is transformed in a sequential way, the randomness is restricted compared with the strawman DSGD under PS-based architecture. Previous studies [18], [20] observe that the sequential method may damage the algorithm robustness and affect the convergence performance. Considering this, we design three strategies to add more randomness to Rima.

Direction randomness. Since there are two directions to choose for *pattern* transformation (i.e. rightwards and upwards, or leftwards and downwards) in the unidirectional ring-based model parallelism, we can randomly decide the direction to move after each worker has completed the computation. In other words, each worker can choose either to tranfer P block or Q block to its successor so long as all workers keep the

same action for each iteration [3].

Intra-block randomness. During the SGD process of each iteration, we randomize the selection of rating entries within the related R block (as shown in Figure 6(b)). Previous works, such as FPSGD [20], [39], select rating entries orderly in R block to avoid memory discontinuity and reduce cache misses. However, the performance benefit due to memory continuity is trivial in distributed computing for large-scale MF, where the overheads are dominated by communication and disk I/O costs [4]. Therefore, we sacrifice some memory continuity for more randomness.



(a) FPSGD orderly selects rating entries within the R block for more memory continuity

(b) Rima randomly selects rating entries within the R block for more randomness

Fig. 6: Comparison between Rima and FPSGD in rating selection within R block

Inter-block randomness. Instead of partitioning R into $w \times w$ blocks, we can generate more blocks to add randomness. As shown in Figure 7, we partition R into 6×6 blocks instead of 3×3 blocks, each worker holds two blocks in a *pattern*. Then the *pattern* transformation becomes more diversified. We use Figure 7 to illustrate inter-block randomness. Figure 7(a) is the initial *pattern* used in the iteration. For the two blocks held by one worker, they can move towards the same direction (shown in Figure 7(b)) or different directions (shown in Figure 7(c)). Besides, the step size can also be various and the *pattern* can move for two-block distance towards the same direction(shown in 7(d)) or different directions (shown in 7(e)). All these transformation will not break the unidirectional ring-based model parallelism. Note that more blocks can better overlap computation and communication, however, too many blocks also require more send/receive operations, thus incurring more start-up overheads [40]. Besides, in our experiment, we also observe that more blocks may also cause the algorithm to converge to a worse local optimum [5]. Therefore, there is expected to be a proper trade-off between the block number and randomness, and we leave this for our future work.

[3]If one worker decides to transfer P block, all the other workers should also transfer P blocks at the end of this iteration, otherwise, all workers should transfer Q blocks. So long as all workers keep the same action for each iteration, the unidirectional ring-based model parallelism will not be broken.

[4]The authors of FPSGD admit that FPSGD is designed for shared-memory systems whereas DSGD is designed for distributed systems [20], [39], therefore, the memory discontinuity problem is more important in FPSGD and they are less concerned about communication cost.

[5]If R is partitioned into 6×6 blocks, then during each iteration, it has less rating entries to update the same P block and Q block compared with 3×3 partition, because the size of one R block in 3×3 partition is four times as large as that of one R block in 6×6 partition.

D. Preloading R Blocks in Rima

When R is too large to be stored in memory, workers need to read the required R block from disk during each iteration. In previous works, because each worker does not know which R block assigned to itself in advance, the blind prefetching strategy is less effective. To be aware which block to prefetch, the worker needs to involve additional communication with the PS. The situation is even more complex for solutions like FPSGD and MLGF-MF, because these solutions partition R into more blocks to mitigate locking problem, however, in that way the *pattern* becomes undeterministic [6] and the PS has to manage more complex scheduling to guarantee each R block is used for equal times.

However, the *pattern* sequence of Rima is independent from the performance of workers in each iteration. In other words, the *pattern* sequence can be pre-determined with *partial randomness* strategies involved. During the start-up of Rima, the *pattern* sequence can be loaded to workers, thus they can know which R block to prefetch before the iteration starts.

To overcome the disk I/O overheads, we use predefined *pattern* sequence and design a preloading mechanism to load R blocks from disk to memory in advance. During the start-up period, each worker will load the maximum number of R blocks according to the memory budget for the following iterations. Then we launch a thread to keep freeing memory for used R blocks and reading fresh R blocks into memory from the disk. In this way, the disk I/O operation can be overlapped with computation/communication.

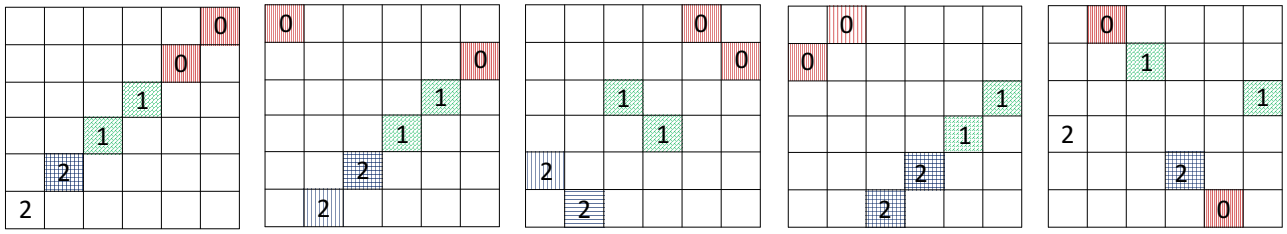
E. Summary

With the aforementioned strategies involved, we present the complete algorithm of Rima executed on each server (worker), which can be described as Algorithm 2 and Algorithm 3.

Targeting at the four main drawbacks with PS-based solutions for large-scale MF (i.e. communication cost, working memory occupation, QP consumption, and disk I/O overheads), Rima involves three main design strategies to overcome these drawbacks:

1) Rima executes with *pattern* sequences generated by *one-step transformation* to halve the transmission workload and reduce communication cost. Besides, some *partial randomness* strategies are involved to add more randomness and guarantee the algorithm robustness.
2) Rima adopts ring-based model parallelism instead of centralized PS-based model parallelism, thus reducing the working memory occupation and QP consumption to better embrace the benefit of RDMA.
3) Rima uses predefined *pattern* sequence, so workers can preload required R blocks from disks and overlap the disk read operations with communication/computation, thus disk I/O overheads are largely eliminated.

[6]Faster workers get more R blocks to calculate whereas stragglers get less blocks. When there are free blocks, they will be assigned to the fastest worker, which first asks the PS for more blocks. Therefore, we cannot completely decide the block assignment in advance.



(a) Initial *pattern* (b) Pattern blocks move towards the same direction for one-block distance (c) Pattern blocks move towards different directions for one-block distance (d) Pattern blocks move towards the same direction for two-block distance (e) Pattern blocks move towards different directions for two-block distance

Fig. 7: More blocks to add inter-block randomness

Algorithm 2: Rima for Large-Scale MF

Input:
w: The number of workers
m: The row number of R
n: The column number of R
k: The column number of P (i.e. the row number of Q)
T: The iteration number of Rima

```
Rima_Main()
    Load the predefined pattern sequence (generated with partial randomness strategies)
    Launch the data loading thread Rima_Load_Td
    Launch the send thread Rima_Send_Td
    Launch the receive thread Rima_Recv_Td
    Initialize the corresponding P block and Q block
    for t ← 0 to T − 1 do
        if R block is available in memory then
            Execute SGD with the R block, P block and Q block
            Update P block and Q block according to Equation 6 and 7
            Put P block or Q block to sendQu according to the predefined pattern sequence
            Fetch fresh P block or Q block from recvQu
        else
            Wait for the R block to be loaded by Rima_Load_Td
        end
    end
    if The server is the master then
        Gather P blocks and Q blocks from the other servers
    else
        Send its final P block and Q block to the master server
    end
```

Algorithm 3: Rima for Large-Scale MF (cont.)

```
Rima_Send_Td()
    while True do
        if sendQu is not empty then
            Fetch P block or Q block from sendQu
            Send the block to the successor
        end
    end
Rima_Recv_Td()
    while True do
        Receive P block or Q block from the predecessor
        Put the block to recvQu
    end
Rima_Load_Td()
    Read maximum R blocks allowed by the memory budget according to the predefined pattern sequence
    while True do
        Traverse R blocks in the memory according to the predefined pattern sequence
        if R block has been used then
            Free the memory and load one fresh R block from the disk to the memory
        end
    end
```

IV. Experiment Evaluation

A. Experiment Setting

We conduct comparative experiments with 5 servers directly connected to a 40GE switch. Each server is equipped with two Intel Xeon E5 CPUs (each CPU has 16 physical cores), 64GB DRAM and 40Gbps Mellanox Connectx-3 NIC. We implement both PS-based DSGD and ring-based Rima with C++ and run the prototypes in Ubuntu 16.04. As for DSGD, we choose one server as the PS and the other four servers as workers. As for Rima, we choose 4 servers as workers. We fully utilize the CPU resources of workers with multi-threading. In this way, the two prototypes possess equal computation power. Meanwhile, we use both TCP and RDMA as the transport protocol for performance comparison.

We choose both real data set and synthetic data set as benchmarks. There are three main data sets used in our experiments, i.e. MovieLen-10M [41], Yahoo!Music [16] and Jumbo. The former two are public data sets with training set and test set partitioned, thus we directly use the partitioned data set for training and testing DSGD and Rima. As for the last one, we follow the methods described in [19], [42] and synthesize the data set. More specifically, we first build the ground truth P and Q with each entry drawn from a Guassian distribution with small variance. The size of P is $1M \times 100$ whereas the size of Q is $100 \times 1M$. Then we randomly sample 20M entries from the multiplied PQ and use the 20M entries as the training set. Meanwhile, we randomly

TABLE I: Summary of experiment setting

Data Set	MovieLens-10M	Yahoo!Music	Jumbo
m	71567	1000990	1000000
n	65133	624961	1000000
Training Set	9301274	252800275	20000000
Test Set	698780	4003960	2000000
Initial Value[1]	[0,0.6]	[0,0.2] [3]	[0,0.3]
Sample Number[2]	6000	150000	30000
k	40	100	100
η	0.003	0.001	0.002
λ	0.01	0.05	0.05

[1] We set each entry of P and Q via uniform random sampling in a certain interval, e.g. As for MovieLen-10M, the initial value of entries in P and Q is randomly chosen from [0,0.6].

[2] During each iteration, each worker runs SGD on a certain number of sample entries in the corresponding R block, e.g. As for MovieLen-10M, during each iteration, the worker will randomly choose 6000 entries from R block to update P block and Q block.

[3] The value of rating entries in Yahoo!Music belongs to [0,100], which may cause computation overflow during our training process, so we use min-max scaling method to transform the original value to [0,1] for our training (and recover the original value while calculating RMSE). The hyper-parameters listed here are used for training the transformed ratings.

sample 2M entries as the test set. Afterwards, we run DSGD and Rima to factorize the $1M \times 1M$ matrix with the 20M training data and evaluate them with the 2M test data.

The details of the parameter setting can be summarized as Table I. For fairness, we use the same hyper-parameters for both DSGD and Rima. Besides, we use strawman SGD for workers in both DSGD and Rima and adopt BSP [43] as the synchronous model. Further optimization strategies, such as adaptive learning rate [21], coordinate descent optimization [42] and staleness-based synchronization [24], are not considered in the comparative experiment. In the comparison we are mainly concerned about the convergence performance and the training time under equal number of iterations. Following previous works [19], [20], [22], [42], we use RMSE (i.e. root mean square error)[7] to measure the convergence performance.

Since there are 4 workers for DSGD and Rima, we partition R matrix into 4×4 blocks and each *pattern* contains 4 *interchangeable* blocks. Besides, we also involve another baseline with 8-block-*patterns* to study the impact of more block partition on the convergence performance and training time. As for Rima's 4-block-*pattern* solution (denoted as Rima (4×4)), we adopt the first two partial randomness strategies described in Section III-C. As for Rima's 8-block-*pattern* solution (denoted as Rima (8×8)), we adopt all the three randomness strategies.

B. Experiment Result and Analysis

The comparison of RMSE convergence and training time can be illustrated in Figure 8 $\sim$ Figure 10.

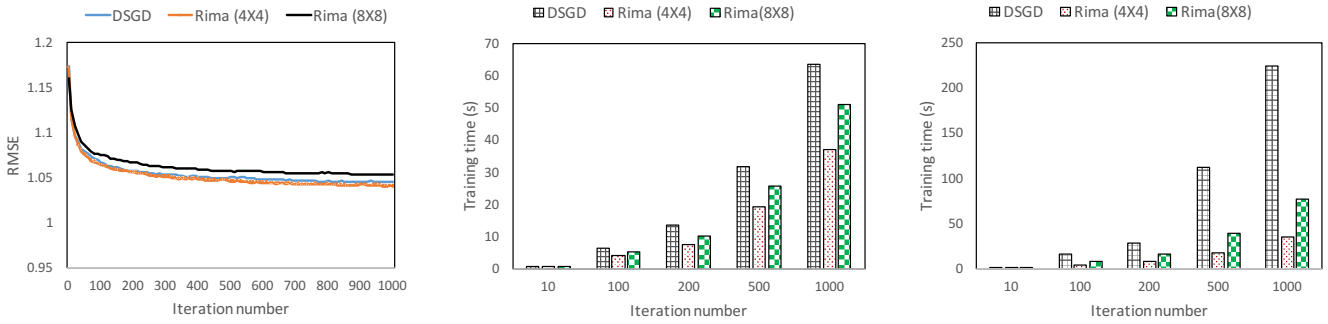
1) Comparison of Convergence Performance: The convergent result is inferior to the experiment results reported in prior works [20], [42] because we only use simple static parameter setting in our experiments and there is no additional optimization involved. However, we focus on the iteration quality comparison between Rima and DSGD under fair setting, rather than searching for the best solution that can be achieved by SGD. From Figure 8(a) $\sim$ Figure 10(a), it can be validated that the partial randomness strategies help Rima (4×4) to maintain comparable convergence performance compared with DSGD. As for MovieLen-10M, the RMSEs of Rima (4×4) and DSGD converge after about 200 iterations and the iteration qualities are comparable: After 200 iterations, the RMSE of DSGD converges from 1.173 to 1.057 whereas the RMSE of Rima converges from 1.172 to 1.057. The situation is similar for the experiment on Yahoo!Music and Jumbo. Besides, Rima (8×8) converges to a less qualified optimum under the same parameter setting. One major reason is because the total size of the two R blocks used by Rima (8×8) in one iteration is only half as large as one R block used by Rima (4×4), thus Rima (8×8) has less candidate rating entries to sample but it needs to use the sampled rating entries to update the same size of P blocks and Q blocks.

2) Comparison of Training Time: We use both TCP and RDMA for data transmission among servers, and the training time comparison is illustrated in Figure 8(b) $\sim$ Figure 10(b) and Figure 8(c) $\sim$ Figure 10(c) respectively.

As for MovieLen-10M, the matrix is relatively small (several MBs) and the non-zero entries can be wholly loaded into memory, thus causing no disk I/O overheads. Compared with DSGD, Rima (4×4) reduces the training time by 41.5% via TCP and 84.7% via RDMA; Rima (8×8) reduces the training time by 19.9% via TCP and 65.7% via RDMA. The training time reduction is mainly due to two reasons. First, the workload is reduced by nearly 50% since P and Q are close in size for MovieLen-10M. Second, the straggler problem is better mitigated by Rima. When the matrix is skew, there is expected to be imbalanced computation workload among workers, thus stragglers can become a significant factor affecting the performance [24]. Under PS-based architecture, DSGD always makes all workers wait for the slowest worker, whereas Rima only requires each worker to wait for its predecessor, thus Rima gains stronger tolerance to stragglers and better overlaps computation and communication. Since the transmission workload is relatively small, the straggler tolerance of Rima is considered to be the major cause for the outperformance.

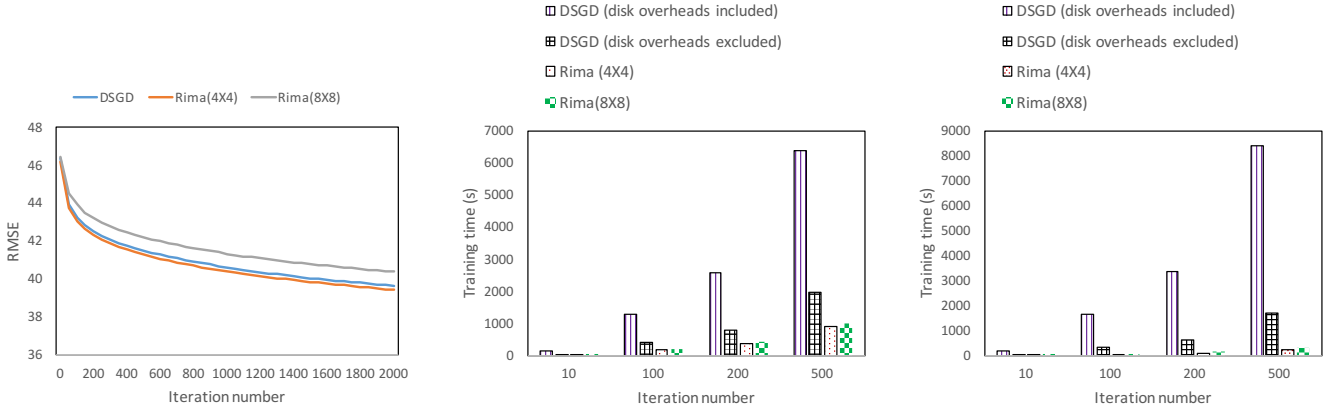
It is surprising to notice that the training time via TCP is even better than that via RDMA for MovieLen-10M. We speculate that this is mainly due to the *self-verification* mechanism [44] we implement in our prototypes. Since we use *one-sided* RDMA Write operation for data transmission. The receiver only registers the memory region in advance and it is unaware when the data will come or whether the data has been completed written by the remote sender. Therefore, the receiver needs to set some flag bits in the registered memory region and keep polling the flag until it is changed by the remote sender. More than that, the receiver also needs to guar-

[7] $RMSE = \sqrt{\frac{1}{N}\sum_0^{N-1}(x_i - x_i')^2}$, where N is the number of entries in the test set, x_i is the value of an entry in the test set, and x_i' is the predicted value of the entry by P and Q.



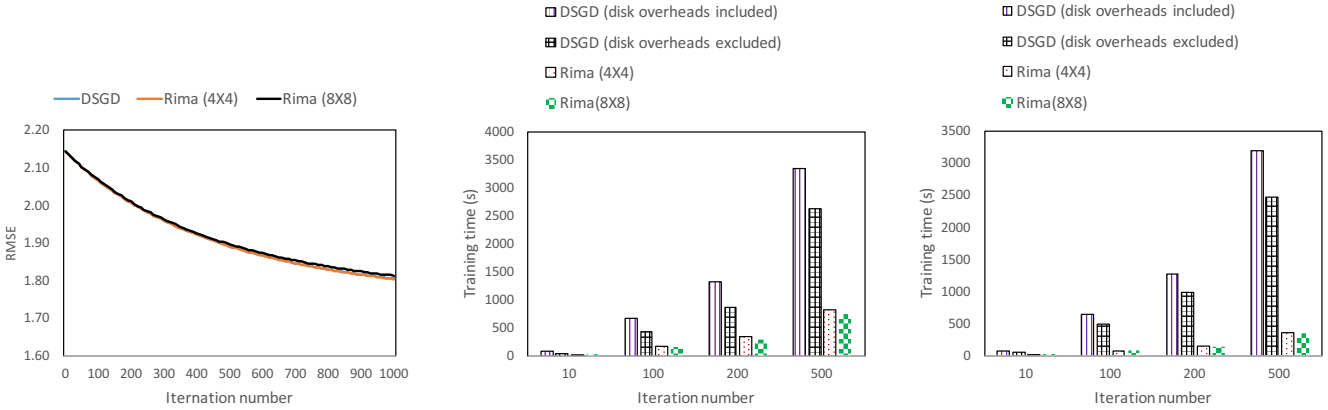
(a) RMSE convergence (b) Training time via TCP (c) Training time via RDMA

Fig. 8: Experiment result with MovieLen-10M



(a) RMSE convergence (b) Training time via TCP (c) Training time via RDMA

Fig. 9: Experiment result with Yahoo!Music



(a) RMSE convergence (b) Training time via TCP (c) Training time via RDMA

Fig. 10: Experiment result with Jumbo

antee that the data has been completely written to its memory. After it has detected the flag change, it then needs to validate the checksum involved in the data written by the remote sender. Only when the checksum is valid, will the receiver start to use the data for calculation. Since the communication cost in MovieLen-10M is not so heavy and computation is intensive in the training process, the polling and checking of our *self-verification* mechanism will compete for some CPU resources with the computation threads, thus extending the overall completion time in one iteration. The design of more efficient *self-verification* mechanism in RDMA is worthwhile and it will be considered in our future work. However, the experimental result still proves that Rima outperforms DSGD via both TCP and RDMA.

An additional finding is that Rima (8×8) has longer training time than Rima (4×4). We attribute the inferiority of Rima (8×8) to more send/receive operations. Although Rima (8×8) transfers less data and can better overlaps computation and communication, there are more frequent send/receive operations, as well as more context switches between the computation thread and communication thread, because the worker has to process two blocks in one iteration. Considering

the communication workload is not heavy with MovieLen-10M, the start-up overheads of send/receive operations [40] and the context switch cost are more significant than communication overheads. On the other hand, when the communication overheads and disk I/O overheads dominate (as shown in the training process of Yahoo!Music and Jumbo), Rima (4×4) and Rima (8×8) reach similar performance (Rima (8×8) is a little better than Rima (4×4) in some cases) and they both significantly outperform DSGD in the training time.

Yahoo!Music and Jumbo are large matrices and Jumbo is even sparser. Therefore, the communication cost and disk I/O cost dominate the overheads in the training process, and the workload reduction effect of Rima is distinct while training with the two matrices. To make a more detailed comparison, we consider DSGD with disk I/O included and excluded [8].

As for Yahoo!Music, the disk I/O operations in DSGD occupy 69.2% of training time via TCP and 79.9% via RDMA. Compared with DSGD (with disk overheads excluded), Rima (4×4) reduces the training time by 53.2% via TCP and 85.4% via RDMA for 500 iterations. As for Jumbo, the disk I/O operations in DSGD occupy 21.6% of training time under TCP and 23.0% under RDMA. Compared with DSGD (with disk overheads excluded), Rima (4×4) reduces the training time by 68.7% via TCP and 85.4% via RDMA for 500 iterations. The performance gains of Rima mainly come from two aspects. On one hand, the transmission amount is large enough during each iteration (several hundred MBs per iteration) and the benefit of half workload reduction is significant for Rima. On the other hand, the disk I/O overheads are significant for DSGD but can be greatly eliminated by Rima with the preloading mechanism described in Section III-D.

Moreover, from the last two experiment cases, we have also witnessed that RDMA can reduce the transmission time for large MF. However, it is worth noting that Rima better harnesses the benefit of RDMA than DSGD. While training with Yahoo!Music and Jumbo, Rima via RDMA reduces the training time by more than 40% compared with the counterpart via TCP. On the other hand, DSGD only gains about 10% acceleration for the training process. The reason is mainly because DSGD executes under PS-based architecture and the PS has to manage the communication with all workers. In this way, the PS becomes the network bottleneck. The incast traffic under PS-based architecture may also incur some RDMA-related problems (e.g. head-of-the-line blocking and spreading congestion) [33], [36], which can seriously damage the training performance. By contrast, Rima executes in a decentralized way and the workload is evenly distributed. Meanwhile, Rima is free from incast traffic and there is only one-to-one communication involved. Therefore, the RDMA-related problems are better avoided and Rima takes more advantage of RDMA power.

[8]The R matrices for Yahoo!Music and Jumbo are almost 1TB and they can not be completely stored in the memory of our single servers. However, since both matrices are sparse, we have compressed them before we can successfully load them into memory. Even so, there are many larger and denser matrices in practice and the consideration of disk I/O overheads is necessary.

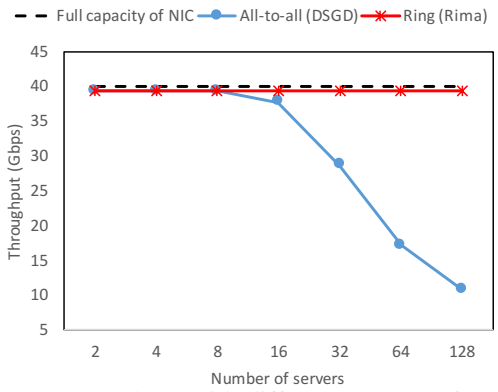


Fig. 11: Throughput with different number of servers

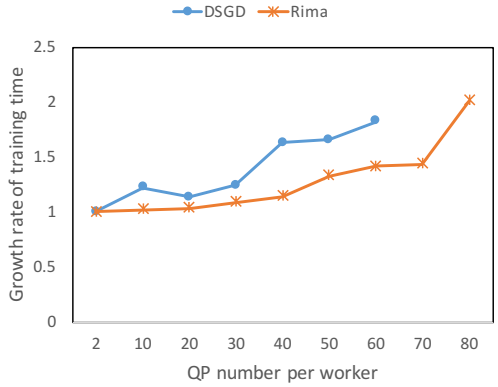


Fig. 12: Time *growth rates* with different number of QPs

3) Comparison of Scalability for RDMA: We conduct both simulation experiment and testbed experiment to demonstrate better RDMA scalability of Rima. Typically most distributed applications run on CLOS (Fat-Tree) network [45] in the data center. Therefore, we use ns-3 [46] [47] to simulate large-scale RDMA communication on Fat-Tree topology. Considering the communication patterns of PS-based DSGD and ring-based Rima, we compare the throughput performance between all-to-all communication [9] and ring-based communication. The comparative result can be shown in Figure 11. Ring-based communication (Rima) maintains nearly full-speed (40 Gbps) throughput performance as the number of server increases. On the contrary, all-to-all communication (DSGD) suffers from a serious decline because of the incast traffic and its corresponding RDMA-related problems.

Furher, we conduct testbed experiment with the 4-worker cluster to prove the scalability drawbacks of DSGD due to large QP consumption. We increase QP number on each server and compare the training time between Rima and DSGD. Every time when it comes to data transmission, the server takes a round-robin way to choose one QP to transmit data. We compare Rima and DSGD according to the completion time of 200 iterations with Yahoo!Music [10]. As what has been shown in Figure 9(c), even with the same data set and equal

[9]All-to-all communication is a special communication pattern in PS-based architecture when each server serves as both PS and worker. All-to-all communication indicates that neither senders nor receivers will become the bottleneck. Ideally, each server can obtain a full-speed throughput performance, however, the result is far from that due to the RDMA-related problems incurred by the incast traffic.

[10]The disk I/O overheads have been excluded from the completion time.

iterations, there is still a gap between Rima and DSGD in the training time and it is not suitable to directly compare the training time between the two solutions. Considering this, we compare the *growth rates* of training time for Rima and DSGD as the QP number increases.

The comparative result can be shown in Figure 12. We choose the training time as baseline with 2 QPs on each worker (i.e. The *growth rate* is set 1 with 2 QPs on each worker). Then, we increase the QP number and calculate the *growth rate* of time. As the QP number increases on each worker, the training time of DSGD is much more prolonged than Rima. When the QP number reaches 60 per worker, the training time of DSGD has been prolonged to $1.82\times$ compared with the baseline, whereas the training time of Rima is only prolonged to $1.41\times$, which is much smaller than DSGD. Besides, we also find that DSGD cannot work with more than 60 QPs per worker. When each worker maintains 60 QPs, the PS has to maintain 240 QPs, which causes RDMA registration failure due to lack of memory resources in our NIC adapters [11]. As shown in Figure 12, during the whole process of QP increase, Rima always enjoys a smaller *growth rate* compared with DSGD, which demonstrates its better scalability for RDMA.

Based on the simulation and testbed experiments, we can conclude that Rima enjoys better scalability for RDMA than DSGD. The scalability constraints of DSGD come from both the incast communication and the QP consumption. By contrast, Rima avoids incast traffic and only involves one-to-one communication, meanwhile, the ring-based communication facilitates Rima to execute MF with less QPs and achieve graceful performance degradation as the number of workers increases or the number of QPs on each worker increases.

V. Related work

Parallel or Distributed Algorithms for Large-Scale Matrix Factorization. Matrix factorization is a fundamental technique in statistical machine learning and there have been a variety of distributed algorithms proposed in previous works. Though these algorithms are usually compatible to both multi-core single machines and distributed clusters, parallel algorithms and distributed algorithms focus on different cost dimensions. Distributed algorithms(e.g. DSGD [18], Nomad [21], MLGF-MF [22], etc.) care more about the communication cost and disk I/O overheads in the training process. On the other hand, parallel algorithms (e.g. FPSGD [20], HogWild! [19], CCD++ [42], etc.) care more about the locking problem and memory discontinuity in shared-memory systems.

RDMA over Converged Ethernet (RoCE). RDMA is considered to be a promising alternative technology for TCP and has attracted much attention from both industry and academia [32]–[34], [37], [38], [44]. Recent works have applied RDMA in distributed machine learning (DML) to accelerate the communication and boost the performance [49], [50]. Currently there are three main branches of RDMA technologies, namely, RoCE, Infiniband and iWarp. Among them, RoCE is most widely applied in industry, with a multi-vendor ecosystem [32], [33]. Besides, prior works [32], [51] have also demonstrated the outperformance of RoCE compared with the other two branches. Therefore, we focus on the application of RoCE to solve large-scale MF, however, QP consumption is a common concern for all the three branches of RDMA and it is also worth considering for Infiniband and iWarp application in practice.

Parallelism and Synchronization. Data parallelism and model parallelism are two main strategies in DML to solve large-scale machine learning problems. As for data parallelism, each worker keeps the same model but trains the model with different portion of training data. As for model parallelism, both the training data and the training model are partitioned and distributed to different workers. MF is a typical case of model parallelism. Both parallelism modes require workers to synchronize their parameters after some iterations of independent training, thus communication cost cannot be ignored while designing DML algorithms and systems. Bulk synchronous parallel (BSP) model [43] is usually adopted in DML for parameter synchronization. Recent works, such as *partial barrier* [52] and SSP [24], relax the original constraints in BSP to mitigate straggler problem. In this paper, we only focus on MF under BSP model. More extensive studies with other synchronous models will be considered in our future work.

VI. Conclusion

This paper proposes Rima, which is a novel RDMA-accelerated solution to large-scale matrix factorization. Compared with existing works, Rima elaborately generates the *pattern* sequence for consecutive iterations, thus halving the transmission workload. Besides, Rima overlaps disk-based operations with computation/communication via predefined *pattern* sequence, and reduces disk I/O overheads. More importantly, Rima leverages ring-based model parallelism instead of PS-based model parallelism, which eliminates the incast traffic and saves QP consumption on each server, thus better harnessing the power of RDMA to accelerate the training process.

[11] It is possible to change the configuration of Mellanox ConnectX adapters for more memory resources to register, but that may bring more cache misses and increase the latency [48]. Therefore, we keep the default configuration for our experiments.

References

[1] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: Scalable online collaborative filtering," in *Proceedings of WWW'07*. New York, NY, USA: ACM, 2007, pp. 271–280.

[2] D. Lian, C. Zhao, X. Xie, G. Sun, E. Chen, and Y. Rui, "GeoMF: Joint geographical modeling and matrix factorization for point-of-interest recommendation," in *Proceedings of KDD'14*. New York, NY, USA: ACM, 2014, pp. 831–840.

[3] J. Kawale, H. Bui, B. Kveton, L. T. Thanh, and S. Chawla, "Efficient thompson sampling for online matrix-factorization recommendation," in *Proceedings of NIPS'15*. Cambridge, MA, USA: MIT Press, 2015, pp. 1297–1305.

[4] X. Wang, R. Donaldson, C. Nell, P. Gorniak, M. Ester, and J. Bu, "Recommending groups to users using user-group engagement and time-dependent matrix factorization," in *Proceedings of AAAI'16*. AAAI Press, 2016, pp. 1331–1337.

[5] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua, "Fast matrix factorization for online recommendation with implicit feedback," in *Proceedings of SIGIR '16*. New York, NY, USA: ACM, 2016, pp. 549–558.

[6] M. Jamali and M. Ester, "A matrix factorization technique with trust propagation for recommendation in social networks," in *Proceedings of RecSys '10*. New York, NY, USA: ACM, 2010, pp. 135–142.

[7] F. Wang, T. Li, X. Wang, S. Zhu, and C. Ding, "Community discovery using nonnegative matrix factorization," *Data Min. Knowl. Discov.*, vol. 22, no. 3, pp. 493–521, May 2011.

[8] C. Cheng, H. Yang, I. King, and M. R. Lyu, "Fused matrix factorization with geographical and social influence in location-based social networks," in *Proceedings of AAAI'12*. AAAI Press, 2012, pp. 17–23.

[9] M. T. Al Amin, C. Aggarwal, S. Yao, T. Abdelzaher, and L. Kaplan, "Unveiling polarization in social networks: A matrix factorization approach," in *Proceedings of INFOCOM'17*. IEEE, 2017, pp. 1–9.

[10] S. Li, J. Kawale, and Y. Fu, "Predicting user behavior in display advertising via dynamic collective matrix factorization," in *Proceedings of SIGIR '15*. New York, NY, USA: ACM, 2015, pp. 875–878.

[11] H. Yang, "Bayesian heteroscedastic matrix factorization for conversion rate prediction," in *Proceedings of CIKM'17*, ser. CIKM '17. New York, NY, USA: ACM, 2017, pp. 2407–2410.

[12] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec," in *Proceedings of WSDM '18*. New York, NY, USA: ACM, 2018, pp. 459–467.

[13] M. Stražar, M. Žitnik, B. Zupan, J. Ule, and T. Curk, "Orthogonal matrix factorization enables integrative analysis of multiple rna binding proteins," *Bioinformatics*, vol. 32, no. 10, pp. 1527–1535, 2016.

[14] R. Liao, Y. Zhang, J. Guan, and S. Zhou, "Cloudnmf: A mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets," *Genomics, Proteomics & Bioinformatics*, vol. 12, no. 1, pp. 48 – 51, 2014.

[15] L. Gong, T. Mu, M. Wang, H. Liu, and J. Y. Goulermas, "Evolutionary nonnegative matrix factorization with adaptive control of cluster quality," *Neurocomputing*, vol. 272, pp. 237 – 249, 2018.

[16] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup'11," in *Proceedings of KDDCUP'11*. JMLR.org, 2011, pp. 3–18.

[17] R. M. Bell and Y. Koren, "Lessons from the netflix prize challenge," *SIGKDD Explor. Newsl.*, vol. 9, no. 2, pp. 75–79, Dec. 2007.

[18] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of KDD '11*. New York, NY, USA: ACM, 2011, pp. 69–77.

[19] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of NIPS'11*. USA: Curran Associates Inc., 2011, pp. 693–701.

[20] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "A fast parallel sgd for matrix factorization in shared memory systems," in *Proceedings of RecSys '13*. New York, NY, USA: ACM, 2013, pp. 249–256.

[21] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 975–986, Jul. 2014.

[22] J. Oh, W.-S. Han, H. Yu, and X. Jiang, "Fast and robust parallel sgd matrix factorization," in *Proceedings of SIGKDD '15*. New York, NY, USA: ACM, 2015, pp. 865–874.

[23] B. Li, S. Tata, and Y. Sismanis, "Sparkler: Supporting large-scale matrix factorization," in *Proceedings of EDBT '13*. New York, NY, USA: ACM, 2013, pp. 625–636.

[24] H. Cui, J. Cipar, Q. Ho *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *Proceedings of the ATC'14*, Berkeley, CA, USA, 2014, pp. 37–48.

[25] S. Sebastian, S. Venu, and Z. Reza, "Factorbird - a parameter server approach to distributed matrix factorization," *arXiv preprint arXiv:1411.0602*, 2014.

[26] S. Ahn, A. Korattikara, N. Liu, S. Rajan, and M. Welling, "Large-scale distributed bayesian matrix factorization using stochastic gradient mcmc," in *Proceedings of SIGKDD'15*. New York, NY, USA: ACM, 2015, pp. 9–18.

[27] E. Zhong, Y. Shi, N. Liu, and S. Rajan, "Scaling factorization machines with parameter server," in *Proceedings of CIKM '16*. New York, NY, USA: ACM, 2016, pp. 1583–1592.

[28] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," *Proc. VLDB Endow.*, vol. 11, no. 5, pp. 566–579, Jan. 2018.

[29] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models," in *Proceedings of AAAI'15*. AAAI Press, 2015, pp. 79–87.

[30] P. Watcharapichat, V. L. Morales, R. C. Fernandez *et al.*, "Ako: Decentralised deep learning with partial gradient exchange," in *Proceedings of ACM SoCC '16*, New York,USA, 2016, pp. 84–97.

[31] H. Zhang, Z. Zheng, S. Xu *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *Proceedings of USENIX ATC '17*, Berkeley,USA, 2017, pp. 181–193.

[32] Y. Zhu, H. Eran, D. Firestone *et al.*, "Congestion control for large-scale RDMA deployments," in *Proceedings of ACM SIGCOMM '15*, New York, USA, 2015, pp. 523–536.

[33] C. Guo, H. Wu, Z. Deng *et al.*, "RDMA over commodity ethernet at scale," in *Proceedings of ACM SIGCOMM '16*, New York, USA, 2016, pp. 202–215.

[34] A. Dragojević, D. Narayanan, O. Hodson *et al.*, "Farm: Fast remote memory," in *Proceedings of USENIX NSDI'14*, 2014, pp. 401–414.

[35] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs," in *Proceedings of USENIX OSDI'16*, 2016, pp. 185–201.

[36] M. Radhika, S. Alex, P. Aurojit, Z. Eitan, K. Arvind *et al.*, "Revisiting network support for RDMA," in *Proceedings of ACM SIGCOMM '18*, 2018. [Online]. Available: https://people.eecs.berkeley.edu/ radhika/irn.pdf

[37] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *Proceedings of USENIX ATC '16*. Berkeley, CA, USA: USENIX Association, 2016, pp. 437–450.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of ACM SIGCOMM '14*, 2014, pp. 295–306.

[39] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Trans. Intell. Syst. Technol.*, vol. 6, no. 1, pp. 2:1–2:24, Mar. 2015.

[40] R. Rabenseifner, "Optimization of collective reduction operations," in *Computational Science - ICCS 2004*, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–9.

[41] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015.

[42] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Proceedings of ICDM'12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 765–774.

[43] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[44] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store," in *Proceedings of USENIX ATC'13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 103–114.

[45] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of ACM SIGCOMM '08*, 2008, pp. 63–74.

[46] nsnam.org, "ns-3: A discrete-event network simulator for internet systems." https://www.nsnam.org/, Dec 2017, accessed on 2018-05-30.

[47] Y. Zhu, "Ns3 simulator for rdma over converged ethernet v2 (rocev2)," https://github.com/bobzhuyb/ns3-rdma, Oct 2017, accessed on 2018-05-30.

[48] Mellanox, "How to increase memory size used by Mellanox adapters," Tech. Rep., 2013. [Online]. Available: https://community.mellanox.com/docs/DOC-1120

[49] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: Distributed data-parallelism for existing ml applications," in *Proceedings of EuroSys'15*. New York, NY, USA: ACM, 2015, pp. 3:1–3:16.

[50] B. Yi, J. Xia, L. Chen, and K. Chen, "Towards zero copy dataflows using rdma," in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos '17. New York, NY, USA: ACM, 2017, pp. 28–30.

[51] Mellanox, "RoCE vs. iWARP competitive analysis," Tech. Rep., 2017. [Online]. Available: http://www.mellanox.com/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf

[52] J. Albrecht, C. Tuttle, A. C. Snoeren *et al.*, "Loose synchronization for large-scale networked systems," in *Proceedings of the ATC '06*, 2006, pp. 28–28.