Enabling Distributed and Optimal RDMA Resource Sharing in Large-Scale Data Center Networks: Modeling, Analysis, and Implementation

Dian Shen, Junzhou Luo, *Member, IEEE, ACM*, Fang Dong, *Member, IEEE*, Xiaolin Guo, Ciyuan Chen, Kai Wang, and John C. S. Lui, *Fellow, IEEE, ACM*

***Abstract*— Remote Direct Memory Access (RDMA) suffers from unfairness issues and performance degradation when multiple applications share RDMA network resources. Hence, an efficient resource scheduling mechanism is urged to optimally allocates RDMA resources among applications. However, traditional Network Utility Maximization (NUM) based solutions are inadequate for RDMA due to three challenges: 1) The standard NUM-oriented algorithm cannot deal with coupling variables introduced by multiple dependent RDMA operations; 2) The stringent constraint of RDMA on-board resources complicates the standard NUM by bringing extra optimization dimensions; 3) Naively applying traditional algorithms for NUM suffers from scalability issues in solving a large-scale RDMA resource scheduling problem. In this paper, we present how to optimally share the RDMA resources in large-scale data center networks with a distributed manner. First, we propose Distributed RDMA NUM (DRUM) to model the RDMA resource scheduling problem as a new variation of the NUM problem. Second, we present distributed algorithms to efficiently solve the large-scale, inter-dependent RDMA resource sharing problem for different RDMA use cases. Through theoretical analysis, the convergence and parallelism of proposed algorithms are guaranteed. Finally, we implement the algorithms as a kernel-level indirection module in the real-world RDMA environment, so as to provide end-to-end resource sharing and performance guarantee. Through extensive evaluations by large-scale simulations and testbed experiments, we show that our method significantly improves applications' performance under resource contention, achieving $1.7-3.1\times$ higher throughput, and in a dynamic context, the largest performance improvement reaches 98.1% and 64.1% in terms of latency and throughput, respectively.**

Manuscript received 30 October 2021; revised 8 October 2022; accepted 20 March 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Llorca. Date of publication 7 April 2023; date of current version 19 December 2023. This work was supported in part by the Jiangsu Provincial Key Research and Development Program under Grant BE2022065-4, in part by the National Natural Science Foundation of China under Grant 6227072991, in part by the Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201, and in part by the Fundamental Research Funds for the Central Universities under Grant 2242021R41177. The work of John C. S. Lui was supported in part by the RGC GRF under Grant 14215722. *(Corresponding author: Fang Dong.)*

Dian Shen, Junzhou Luo, Fang Dong, Xiaolin Guo, and Ciyuan Chen are with the School of Computer Science and Engineering, Southeast University, Nanjing 211189, China (e-mail: dshen@seu.edu.cn; jluo@seu.edu.cn; fdong@seu.edu.cn; xlguo@seu.edu.cn; cychen@seu.edu.cn).

Kai Wang was with the School of Computer Science and Engineering, Southeast University, Nanjing 211189, China (e-mail: kwang@seu.edu.cn).

John C. S. Lui is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China (e-mail: cslui@cse.cuhk.edu.hk).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2023.3263562, provided by the authors.

Digital Object Identifier 10.1109/TNET.2023.3263562

***Index Terms*— Data center network, RDMA, distributed optimization.**

I. INTRODUCTION

REMOTE Direct Memory Access (RDMA) is a technology of high speed data transfer among applications across networks [1], [2]. Based on kernel bypass, RDMA allows applications to perform data transfers from user-space directly to RDMA Network Interface Card (RNIC) without the involvement of the operating system kernel. RDMA can achieve significantly higher throughput, lower latency, and lower CPU utilization than traditional TCP/IP based protocols, thus becoming a promising networking technology for data center applications.

Although the kernel bypassing design of RDMA is efficient for data transfer, it does not work well in a shared data center environment as reported in [3], [4], and [5]. In particular, native RDMA does not provide efficient resource management across applications [4]. When multiple applications share the RDMA-enabled network, one greedy application could monopolize the resources by issuing a large batch of requests. Thus, RDMA fails to deliver the quality of service (QoS), performance isolation, or fairness guarantee in the shared environment.

To this end, one urging question is how to design an efficient resource scheduling mechanism which optimally allocates the RDMA resources among applications. Existing proposals [3], [4], [5], [6] addressed this issue at the system level, by implementing RDMA resource scheduling solutions with performance isolation and rate allocation. However, the scheduling policies were based primarily on engineering heuristics, and they are far away from the optimality of resource allocation efficiency. Theoretically, network resource scheduling optimization can be formulated as a Network Utility Maximization (NUM) problem [7], [8], [9], [10], where the utility provides a metric to measure the optimality of resource allocation efficiency, such as QoS or fairness. While previous efforts [11], [12], [13] in TCP/IP networks have been successful in deriving optimal resource scheduling solutions based on solving some specific NUM, none has considered RDMA. In this paper, we propose a novel adoption of NUM to address the optimal RDMA resource scheduling.

However, the aforementioned NUM-based solutions require a strong assumption on the optimization variables and utility functions. Specifically, a common assumption used in previous

methods is that there is only one set of optimization variables (e.g., a rate allocation vector) and the utility function is dependent on this set of optimization variables only [8], [11], [12], [13]. Because such an assumption contradicts the complexities inherent in RDMA networks, it is challenging to adopt the NUM for RDMA resource scheduling. The challenges include the following three aspects.

Multiple dependent RDMA operations introduce many coupled variables. RDMA exposes to higher layer applications multiple low-level hardware primitives such as operations on multiple queues. These operations, with different functionalities, can be described by different utility functions. Since these operations are dependent (Section II-C), the RDMA resource scheduling problem can be formulated as a multi-block optimization problem with coupled variables. Consequently, the assumption of NUM is invalid, and the standard NUM algorithms are unable to deal with it.

Stringent constraint of RDMA on-board resources brings one extra dimension in the utility function. RDMA caches the connection information on the RNIC to achieve low latency communications. As the on-board RNIC cache is limited, when the number of connections grows, the total size of connection states will exceed the RNIC cache size and cause cache thrashing (Section II-D), impairing the performance of hosting applications. Standard NUM [7], [8] treats the number of connections as unconstrained flows. With such a stringent cache constraint, the active connections in RDMA should be carefully selected or prioritized. Thus, this is another optimization dimension which invalidates the assumption of NUM, making the standard NUM more complicated to solve.

The inherently large RDMA network scale causes the scalability issue. In a production RDMA network, the number of hosts is on the order of $O(10^4)$ to $O(10^5)$, and the number of applications on each host is of $O(10^3)$. Thus the number of variables can be up to $O(10^8)$. Naively applying traditional algorithms for NUM suffers from scalability issues in solving such a large-scale optimization, especially for the cases with coupled RDMA operations among multiple hosts. Thus, we are motivated to consider distributed optimization, and an implementation that can provide a fast and scalable solution.

In this paper, we present how to optimally schedule applications in the RDMA network with a distributed algorithm. Specifically, we first model the RDMA resource scheduling problem as a new variation of the NUM problem to characterize the complexities of RDMA networks, called Distributed RDMA NUM (DRUM). DRUM is inherently a multi-block constrained optimization problem, which jointly optimizes the sharing on multiple RDMA resources. Second, taking into account the RDMA on-board resources constraint, we design the objective function of DRUM by applying a top-$\tilde{k}$ applications selector. Considering the resource preemption scenario, we further model the resource usage of each connection with a certain preemptive range, such that the resources are allowed to be preempted by other applications. We then analyze the tractability of the proposed model by convexity analysis. Third, we present a distributed solution which splits the large-scale global optimization problem into many small local subproblems. We propose a novel distributed algorithm based on the alternating directional method of multipliers (ADMM) and prove the convergence guarantee and parallelism of our ADMM-based algorithms through theoretical analysis. In particular, we analyze in depth the difference when modeling different types of RDMA operations.

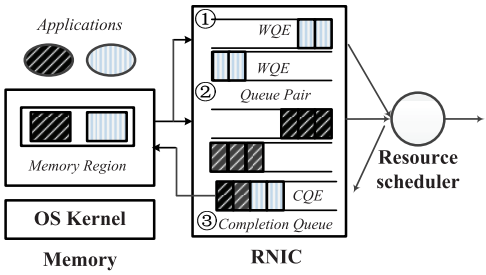


Fig. 1. RDMA abstracts resources in the semantics of queues and allows kernel bypass data transfer for applications.

To demonstrate the applicability of our proposed algorithms, we implement them as a kernel-level RDMA indirection module which manages all the privileged resources. The implemented system, DRUM-agent, is an end-to-end solution that can be deployed in a distributed manner in the current RDMA network architecture. Finally, we conduct extensive testbed experiments in the real-world RDMA environment with a comparison to the state-of-the-art methods. Experimental results show that DRUM can significantly improve applications' performance in the shared RDMA network, achieving $1.7-3.1\times$ higher throughput under heavy background traffic. In a dynamic context, the largest performance improvement reaches 98.1% and 64.1% in terms of latency and throughput, respectively. The overhead of DRUM is also mild that it only consumes less than 30% usage of one CPU core when serving over 250 connections on one host.

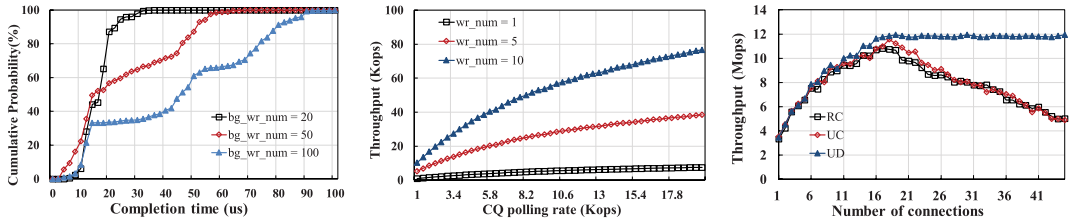
To the best of our knowledge, this paper is the first work to discuss the modeling, analysis and implementation of the RDMA resource sharing problem. Our major contributions are summarized as follows:

- We propose a new variation of the NUM model called DRUM to formulate the RDMA resource sharing problem, addressing the inherent complexities within RDMA networks.
- We present a distributed and modular algorithm to solve the problem, and prove the convergence guarantee and parallelism through theoretical analysis.
- We implement a kernel-level RDMA indirection module which manages all privileged resources and enables resource sharing with the proposed algorithms.
- We evaluate our method through large-scale simulations and testbed experiments. The experimental results show that the RDMA network achieves higher scalability and improved network performance, compared with the state-of-the-art methods.

II. Background and Motivation

A. RDMA and RDMA Resource Management

RDMA is a technology of high speed data transfer among applications across a network [1]. Different from traditional TCP/IP networks, RDMA bypasses the operating system kernel and allows applications to directly access the low-level hardware resources in RNIC. Therefore, RDMA eliminates



(a) The 80% completion time of a 1KB RDMA message degrades from 19.8 μs to 74.3 μs when the WQE requests batch size of background application increases from 20 to 100.

(b) The throughput of applications grows with the increase of CQ polling rates by different amounts, when the batch size of WQEs are set as 1, 5, and 10, respectively.

(c) Average throughput of RC and UC connections drop drastically when the number of concurrent connections grows larger than 20.

Fig. 2. Characteristics of RDMA communication: (a) The unfairness issue in the shared network; (b) dependent RDMA operations; (c) cache thrashing issue under a large amount of connections.

the packet processing overhead in the kernel so as to provide both high bandwidth and low latency.

RDMA abstracts the resources as various queues, and applications implement the communication using these queues. As Fig. 1 shows, there are primarily three queues. The "send queue" and "receive queue" are always created in pairs and are referred to as a Queue Pair (QP). To perform data transfer, an application creates a QP and places instructions on the QP. These instructions are small data structures called Work Queue Elements (WQEs), which contain the memory location where the data reside and where it wants to send. The RNIC then processes the WQEs to send the data. The third one is the Completion Queue (CQ), which is used to notify the applications when the WQEs have been completed. The completion notifications are called Completion Queue Entries (CQEs). The applications actively poll the CQEs to determine the completion of messages sent.

When multiple applications share the network, native RDMA does not provide efficient management of resources across applications [4]. As a consequence, one application can simply post a large batch of WQEs to monopolize the resources, thereby degrading the performance of other applications. As a simple illustration in Fig. 2(a), the 80% completion time of one foreground traffic degrades by 3.9× from 20 μs to 78 μs when the co-located application increases the WQE requests batch size from 20 to 100.

B. Network Utility Maximization

The theory behind network management problems is usually generalized by Network Utility Maximization (NUM). NUM is a powerful and widely-used modeling tool, the formulation of which captures network resource management objectives through utility functions and models various types of resource constraints as the constraint set. The basic NUM problem has the following formulation

$$\begin{aligned} &\text{maximize} \quad \sum_s U_s(x_s), \\ &s.t. \quad \forall s : x_s \in \mathcal{X}_s, \end{aligned}$$

where s denotes any source in the network, x_s denotes the source rate such as the bandwidth, $U_s(x_s)$ denotes the utility functions, and $\{\mathcal{X}_s\}$ is the constraint set, such as bandwidth constraints. Previous researches modeled TCP/IP networks and derived resource scheduling solutions based on solving some specific NUM with particular utility functions [12], [13].

Assumption of NUM. In a standard NUM model, the rate allocation vector $\mathbf{x} = \{x_s\}$ is usually assumed to be the only set of optimization variables and utility functions $U_s(x_s)$ are often assumed to be dependent on $\mathbf{x}$ only [8], [12], [13].

However, such an assumption is violated in the context of RDMA networks due to their inherent complexities. In the following section, we demonstrate the inherent complexities within RDMA networks through experimental results and analysis, and then present how to extend NUM with a new variation of this model.

C. The Dependency Among Multiple RDMA Operations

RDMA resource scheduling involves the management on multiple queues, e.g., allocating the WQE requesting rates on QP and deciding the CQE polling rates on CQ. At the application level, the operations on these queues are inherently dependent. Specifically, the applications usually implement the communications through a "WQE requesting-CQE polling-WQE requesting" loop that an application can only issue a batch of WQE requests after polling one CQE. The batch size is controlled by the parameter `wr_num`. For example, TensorFlow-RDMA [14] sets `wr_num` to be 1 when processing tensors and Spark [15] sets it dynamically according to the shuffle data in the buffer. In the experiment, we manually adjust the CQE polling rate of one application and measure its throughput. From Fig. 2(b), we observe that the throughput grows as the CQE polling rate increases, because the application can issue more WQE requests in one loop. Therefore, controlling one communication operation in RDMA affects the other in turn, yielding multiple dependent variables in the resource scheduling optimization. The standard NUM-oriented algorithms are unable to deal with such complexity.

D. RNIC Cache Thrashing Under Concurrent Connections

RDMA caches the connection information on the RNIC to achieve low latency communications. However, since the RNIC cache is limited, when the number of connections grows, the total size of QP states will exceed the RNIC cache size and cause cache thrashing. Cache thrashing significantly impairs the performance of hosting applications. To better understand this issue, we vary the number of concurrent connections in one host and measure the average throughput. In the experiment, RDMA connections are set with different types: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). While RC and UC need to create QPs and maintain the queue states when establishing connections, UD supports unreliable communication without establishing QPs. As Fig. 2(c) shows, when the number

of connections using RC and UC exceeds 20, the RNIC is unable to cache all the connection information and the average throughput drops significantly. The result indicates that, to maintain high communication performance, the number of QPs should be constrained to avoid cache thrashing. With such a stringent cache constraint, the active connections should be carefully selected or prioritized. Therefore, one more optimization dimension on the application selection makes the standard NUM rather complicated to solve.

E. One-Sided and Two-Sided RDMA Operations

RDMA applications transfer messages by directly manipulating RNIC through hardware primitives called RDMA Verbs. The typical Verbs consist of one-sided and two-sided operations. One-sided Verbs, including Write, Read and etc, directly access remote memory without involving the remote server's CPU. In particular, Read Verb is used to fetch data from the memory of a remote host, while Write Verb transfers data into the memory of a remote host. On the contrary, two-sided Verbs require the involvement of the remote server. For example, Send and Recv Verbs are used in pairs that when the Send Verb transfers a message to the remote host, a corresponding Recv Verb must be evoked to receive the message. As introduced earlier, all Verbs are posted, by the data structure of WQEs, to QPs that are created and maintained inside the RNIC. However, the fundamental difference brought by these two types of RDMA operations lies in the fact that one-sided operations only raise WQEs requests and completion notifications on one host, while two-sided operations raise them on both hosts. Therefore, two-sided operations introduce dependent optimization variables, in terms of resource allocation, on different remote hosts, bringing additional challenges for designing a distributed solution for RDMA resource sharing.

III. RDMA Resource Scheduling Optimization

A. Problem Formulation

We consider an RDMA-enabled data center network as illustrated in Fig. 3. It consists of a finite set of connected hosts and a finite set of applications consuming network resources. Let $host_i$ be the i^{th} physical host. There are in total n hosts in the network, i.e., $i \in \mathbb{Z}_n$, where $\mathbb{Z}_n = \{1, 2, \ldots, n\}$. Any $host_i$ runs up to m RDMA-enabled applications. The scheduler allocates the WQE requesting rates s_j to application $j \in \mathbb{Z}_m$, where $\mathbb{Z}_m = \{1, 2, \ldots, m\}$. Define $x_i \in \mathbb{R}_+^m$ as the rate allocation vector for $host_i$, we have $x_{i,j} = s_j$ and $x_i = \{s_1, s_2, \ldots, s_m\}, i \in \mathbb{Z}_n$. The problem is how to determine x_i for each $host_i$. For a straightforward adoption of NUM, the model becomes

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{n} f_i(x_i), \quad x_i \in \mathbb{R}_+^m, \\ &s.t. \quad h_i(x_i) \le q_i, \quad \forall i \in \mathbb{Z}_n. \end{aligned} \tag{1}$$

where $f_i(\cdot) : \mathbb{R}^m \to \mathbb{R}$ is the utility function. Although in general $f_i(\cdot)$ could be an arbitrary concave function, we first carefully discuss how to choose this utility function. In a shared networking environment, the basic design goal is to ensure fairness among all connections instead of blindly maximizing the total throughput, otherwise certain applications

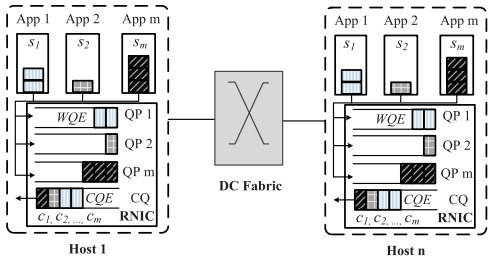


Fig. 3. RDMA resource scheduling model with n connecting hosts and m co-located applications.

could be starving that $\exists i, x_i \to \mathbf{0}^n$. That is to say, some level of balance should be considered between fairness and efficiency in the resource allocation. Addressing this issue, we use a function that satisfies the definition of fairness, which is a maximizer of α-fair utilities for all x_i, parameterized by $0 < \alpha < +\infty$ and a weight parameter w_j. w_j can be adjusted subject to the priority of connections. We define the utility function for all x_i as

$$f_i(< x_{i,1}, x_{i,2}, \ldots, x_{i,m} >) = \sum_{j=1}^{m} w_j \cdot (1-\alpha)^{-1} s_j^{1-\alpha}, \forall x_i. \tag{2}$$

Maximizing such $f_i(\cdot)$ would generate a α-fair allocation [16]. The reason we use the α-fair utility function is the generality of this function that it can satisfy the general needs of network operators. For example, in the case when $\alpha \to 1$, apply L'Hospital's rule, we get

$$\lim_{\alpha \to 1} f_i(x_i) = \sum_{j=1}^{m} w_j \lim_{\alpha \to 1} \frac{s_j^{1-\alpha} - 1}{1-\alpha} = \sum_{j=1}^{m} w_j \log s_j, \tag{3}$$

which generates the weighted proportional fairness. In the case when $\alpha = 2$, $f_i(x_i) = \sum_{j=1}^{m} w_j / s_j$, optimizing such utility function generates Minimum delay potential fairness. In the case when $\alpha \to +\infty$, it generates Max-min fairness. Furthermore, summing up the concave function $f_i(x_i)$ for all i remains its concavity, so that it can be solved easily by exiting numerical optimization algorithms.

The cost function $h_i(\cdot) : \mathbb{R}^m \to \mathbb{R}$ is an arbitrary convex function, such as physical bandwidth limitations or resource prices. We refer to $\mathbf{x}^* = \{x_1^*, x_2^*, \ldots, x_n^*\}$ as an optimal rate allocation if it solves problem (1), i.e., $\sum_i f_i(x_i^*) \ge \sum_i f_i(x_i), \forall x_i$. Traditionally, this problem can be solved via a convex programming solver or Dual Decomposition method.

However, as described in Section II, the complexities of RDMA make the direct transformation from NUM impractical. First, RDMA abstracts the resources by multiple queues. In addition to QP and associated WQE requesting rates, scheduling CQ and associated CQE polling rates should also not be overlooked. We define another rate vector $z_i = \{r_1, r_2, \ldots, r_m\}, i \in \mathbb{Z}_n$, where r_j is the allocated CQE polling rate for each application j and $z_{i,j} = r_j, j \in \mathbb{Z}_m$. In RDMA, the CQE polling rate is closely related to the application-perceived latency, we hereby define a different utility function for CQ.

Definition 1 (CQ Utility): The function is characterized by a multiplier $g_i(z_i) = -\beta \cdot \sum_{j=1}^{m} (1/z_{i,j})^2$, where $z_i \in \mathbb{R}_+^m$.

In this definition, β is the weighting factor that captures the relative importance of latency-related utility. The application-

perceived latency for application j on $host_i$ is captured by $(1/z_{i,j})^2$, which is inversely proportional with its CQE polling rate $z_{i,j}$. Then the overall utility function $g_i(\cdot)$ depends on the application-perceived latency. Clearly, $g_i(\cdot)$ is a concave function and achieves its maximum value when $z_i \to +\infty$.

In the RDMA design, due to the on-board cache size limitation, the number $\tilde{k}$ of active connections using QPs on each $host_i$ should be bounded, i.e. $\tilde{k} < m$. In practice, $\tilde{k}$ is set as $\tilde{k} = 20$ and homogeneous across the data center. Therefore, the selection of $\tilde{k}$ active applications should be considered. Therefore, we rewrite the original $f_i(x_i)$ as $\widetilde{f}_i(x_i)$ using the following definition.

Definition 2 (QP Utility): We define the utility as a multiplier $\widetilde{f}_i(x_i)$, where $x_i \in \mathbb{R}_+^m$. The multiplier prioritizes $\tilde{k}$ applications with the largest utilities.

$$\widetilde{f}_i(x_i) = \sum_{j=1}^{\tilde{k}} (1-\alpha)^{-1} s_{[j]}^{1-\alpha}, \quad \forall s_{[j]} \in x_i, \tag{4}$$

where $[j]$ is the application with j-th largest utility value. Note that $\widetilde{f}_i(x_i)$ is not restrictive to a top-$\tilde{k}$ utility selection. It can be formulated to describe any prioritizing mechanism such as FIFO or smallest job first. Therefore, the utility function of QP is more complicated by applying the functionality of application selection. We theoretically analyze the complexity through convex analysis and derive the following lemma.

Lemma 1: The function $\widetilde{f}_i(x_i)$ is strictly concave.

Proof 1: First, order the element j in vector x_i by the value $(1-\alpha)^{-1} s_j^{1-\alpha}$ in the descending order. Then equation (4) is mathematically equal to sum the top $\tilde{k}$ largest ones up from m elements, which is

$$\widetilde{f}_i(x_i) = max\{(1-\alpha)^{-1} s_{[i_1]}^{1-\alpha} + \ldots + (1-\alpha)^{-1} s_{[i_{\tilde{k}}]}^{1-\alpha} \\ |i_1, i_2, \ldots, i_{\tilde{k}} \in \mathbb{N}, 1 \le i_1 \le \ldots \le i_{\tilde{k}} \le m\}. \tag{5}$$

Thus, there are $C_m^{\tilde{k}}$ combinations of such selection. We can define a binary vector $t \in \mathbb{R}^m$, where the corresponding element of an accepted application is set to 1, while the others are 0. Transform equation (5) to $\sup_t\{f_i(t \cdot x_{[i]}) | \forall t\}$. This means that we conduct a supremum operation on $C_m^{\tilde{k}}$ combinations of $f_i(\cdot)$, which is $\widetilde{f}_i(\cdot) = \mathrm{g}(f_i(x_i))$, where $\mathrm{g}(\cdot) = \sup_t\{\cdot\}$. Because $f_i(x_i)$ is concave and $\mathrm{g}(\cdot)$ is the operation that maintains concavity, then $\widetilde{f}_i(\cdot)$ is still a concave function. Therefore, the lemma is proved. □

Other than the objective function, we also carefully consider the cost function $h_i(\cdot)$, which should be an arbitrary convex function that restricts the physical RNIC capabilities, such as the logarithm function used in [17], or a more general piecewise linear function with increasing slopes. For computational convenience, we define $h_i^q(x_i) = \sum_j log(x_{i,j}) \le q_i$ as the physical constraints for x_i, and $h_i^c(z_i) = \sum_j log(z_{i,j}) \le c_i$ for z_i, where q_i and c_i are set according to the hardware specifications. Note that we use a trick that the formulation constrains the physical usages of RDMA on-chip resources with logarithmic functions. When applying this trick, we define q_i and c_i as $q_i = \beta_i \dot{q}_i^m$ and $c_i = \gamma_i \dot{c}_i^m$, where $\dot{q}_i$ and $\dot{c}_i$ are the original hardware specifications, β_i and γ_i are the scaling factors. The rationale behind the scaling is that summing up the logarithmic of $x_{i,j}$ and $z_{i,j}$ is the product of their multiplications. Therefore, the scaling of $\dot{q}_i$ and $\dot{c}_i$ with exponentials approximates the original constraints. This transformation will result in eliminable variables in the derivation of the closed-form solution of optimization, and the details will be described in the proof of Theorem 2 and 3.

We further consider the scenario where preemptive scheduling is allowed in the RDMA network. Preemptive scheduling is especially important for RDMA use cases where there are many real-time or time-sensitive applications. To model this feature, we extend the rate allocation vector x_i with an ellipsoidal region, denoted as $\boldsymbol{a}_i^T x_i, \forall i$, where $\boldsymbol{a}_i \in \mathcal{A}_i$. $\mathcal{A}_i$ is an ellipsoidal region that $\mathcal{A}_i = \{\boldsymbol{a}_i : \boldsymbol{a}_i = \overline{a}_i + \mathfrak{q}_i u, ||u||_2 = 1\}$. The physical meaning of $\mathcal{A}_i$ is that we allow preemption on the RDMA resources in cases when some latency bounded applications need to schedule immediately. In this setting, the preemption range is controlled by $\mathfrak{q}_i$ that if the application is not very urgent, it can declare that it needs a minimum resource guarantee $\overline{a}_i$, and $\mathfrak{q}_i u$ amount of resources can be preempted by other applications. For time sensitive applications, it can declare that all the resources should be strictly guaranteed. In this case, $\overline{a}_i = 1$ and $\mathfrak{q}_i = 0$ and such that x_i degrades to the standard modeling. Therefore, with this extension of x_i, we can model different preemption tolerance for various kinds of applications. Note that the same technique applies to the CQE polling rate z_i as well.

With the above definitions, we formulate a variant of the basic NUM to characterize the RDMA resource scheduling problem, denoted as DRUM, the objective of which is to jointly optimize the utility of x_i and z_i.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} \widetilde{f}_i(x_i) + \sum_{i=1}^{n} g_i(z_i), \quad x_i, z_i \in \mathbb{R}_+^m, \\ s.t. \quad & h_i^q(x_i) \le q_i, \quad \forall i \in \mathbb{Z}_n, \\ & h_i^c(z_i) \le c_i, \quad \forall i \in \mathbb{Z}_n, \\ & F(\mathbf{x}) = \mathbf{z}. \end{aligned} \tag{6}$$

Problem (6) is inherently a large-scale multi-block optimization problem with equality and inequality constraints and dependent optimization variables, which is hard to solve in a timely manner.

B. Problem Analysis and a General Algorithm

Traditionally, to solve problem (6) is based on the method of multipliers, in which there is an augmented Lagrangian parameter ρ, and Lagrangian dual variable u. The optimal value $\mathbf{x}^*$, $\mathbf{z}^*$ and dual optimal value u^* are computed iteratively by solving the following equation:

$$(\mathbf{x}^{k+1}, \mathbf{z}^{k+1}) := \underset{\mathbf{x},\mathbf{z}}{\operatorname{argmax}} \mathcal{L}_\rho(\mathbf{x}, \mathbf{z}, u^k),$$
$$(u^{k+1}) = u^k + \rho(F(\mathbf{x}^{k+1}) - \mathbf{z}^{k+1}).$$

Here, the augmented Lagrangian $\mathcal{L}_\rho(\mathbf{x}, \mathbf{z}, u^k)$ is maximized jointly with respect to the two jointly dependent primal variables. To solve this problem, we leverage the Alternating Direction Method of Multipliers (ADMM) [18] method. In ADMM, x_i and z_i are updated in an alternating fashion; that is, in each iteration k, x_i^k is computed as an intermediate result from the previous state (z_i^{k-1}, u_i^{k-1}). z_i^{k+1} is a function of (x_i^{k+1}, u_i^k), and the Dual variable u^{k+1} is updated from u^k by collecting all the x_i^{k+1} and z_i^{k+1}. Following the ADMM

framework, we obtain an algorithm schema with iterations on the order of the x-minimization step, z-minimization step and u-update step. We illustrate the schema as Algorithm 1:

Step 1, x-minimization step:

$$\min_{x_i} -\widetilde{f}_i(x_i) + u^k(F(\mathbf{x}) - \mathbf{z}^k) + (\rho/2)||F(\mathbf{x}) - \mathbf{z}^k||_2^2,$$
$$s.t. \quad h_i^q(x_i) \le q_i. \tag{7}$$

Step 2, z-minimization step:

$$\min_{z_i} -g_i(z_i) + u^k(F(\mathbf{x}^{k+1}) - \mathbf{z}) + (\rho/2)||F(\mathbf{x}^{k+1}) - \mathbf{z}||_2^2,$$
$$s.t. \quad h_i^c(z_i) \le c_i. \tag{8}$$

Step 3, u-update step:

$$u^{k+1} = u^k + \rho \cdot (F(\mathbf{x}^{k+1}) + \mathbf{z}^{k+1}). \tag{9}$$

By design, the alternating update in ADMM reduces the computing complexity, and the extra regulation term $(\rho/2)||F(\mathbf{x}) - \mathbf{z}||_2^2$ stabilizes the iterations. One important property of the ADMM-based solution is its theoretical guarantee of convergence. The convergence of our algorithm is formally guaranteed by the following theorem:

Lemma 2: The strong duality holds for problem (6), i.e., suppose that problem (6) (P) has an optimal solution x^*, z^*, there exists and optimal solution to $\mathcal{L}_0(\mathbf{x}, \mathbf{z}, u)$ (D), such that the optimal values for (P) and (D) are equal, $V_P = V_D$.

Proof 2: The proof ban be found in Appendix.

Theorem 1: When a feasible solution to the DRUM *exists, the ADMM-based solution converges to the optimal value p^*. In particular, the followings are true:*

1. The consistency is achieved: $\mathbf{x}^k \to F^{-1}(\mathbf{z}^k)$ as $k \to +\infty$.

2. The solution is optimal: $\sum_{i=1}^{n} \widetilde{f}_i(x_i^k) + \sum_{i=1}^{n} g_i(z_i^k) \to p^*$ as $k \to +\infty$.

3. An optimal dual value is found: $\rho u^k \to \rho u^*$ as $k \to +\infty$.

Proof 3: Let $(\mathbf{x}^*, \mathbf{z}^*)$ be the primal optimal solution to problem (4). In particular, $F(\mathbf{x}^*) = \mathbf{z}^*$. Let u^* be a dual optimal solution. Because of the condition that a feasible solution to the DRUM exists, along with the strong duality theorem, u^* exists. Then we have $(\mathbf{x}^k, \mathbf{z}^k, u^k)$ as an arbitrary set of results generated in iteration k. Let $v^k = u^k/\rho$ and $\mathbf{z}^k = F(\mathbf{x}^k) + v^{k-1} - v^k$, so we have

$$V^k = \sum_{i=1}^{n}(||\mathbf{z}^k - \mathbf{z}^*||_2^2 + ||v^k - v^*||_2^2). \tag{10}$$

This equation leads to convergence if $\exists \varepsilon, D^k = V^{k+1} - V^k < \varepsilon$. [18] has proved that D^k is Lipschitz continuous and ε exists $\forall k$ with any start point $\{\mathbf{x}^0, \mathbf{z}^0, u^0\}$, so ε is

$$\varepsilon = \rho||r^{k+1}||_2^2 + \rho||F^{-1}(\mathbf{z}^{k+1} - \mathbf{z}^k)||_2^2, \tag{11}$$

where r^k is the primal residual after k-th iteration. Equation (11) holds with the saddle point theorem when the objective functions $\widetilde{f}_i(\cdot)$ and $g_i(\cdot)$ are closed, proper, and convex. Note that it does not require the combination $\widetilde{f}_i(\cdot) + g_i(\cdot)$ to be convex. With Lemma 1, $\widetilde{f}_i(x_i) = \sum_{j=1}^{k}(1-\alpha)^{-1}s_{[j]}^{1-\alpha}$ is strictly concave, so $-\widetilde{f}_i(x_i)$ is a convex function. By definition, $g_i(z_i) = -q \cdot \sum_j (1/z_{ij})^2$. $g_i(\cdot)$ is concave when $dom(g_i) \in \mathbb{R}_+^m$. Since $z_i \in \mathbb{R}_+^m$, $-g_i(z_i)$ is a convex function. Then we check the convexity of the constraints. We evaluate the function $h_i^q(\cdot)$ and $h_i^c(\cdot)$ by definition. Specifically, for the preemptive setting, we rewrite $h_i^q(x_i)$ by

$$h_i^q(x_i) = (\overline{a}_i + \mathfrak{q}_i u)^T x_i \le q_i \Leftrightarrow \overline{a}_i x_i + u^T \mathfrak{q}_i^T x_i \le q_i$$
$$\Leftrightarrow \overline{a}_i x_i + ||\mathfrak{q}_i^T x_i||_2 \le q_i. \tag{12}$$

Equation (12) indicates that the feasible set of x_i is the sublevel set of $h_i^q(\cdot)$, where $\mathcal{S}_i = \{x_i : \overline{a}_i x_i + ||\mathfrak{q}_i^T x_i||_2 \le q_i\}$. $\mathcal{S}_i$ is a second-order cone (SOC) over x_i, which is closed and convex. The above analysis also applies to $h_i^c(\cdot), \forall i$ as well. It is worth noticing that the modeling of the preemption will not influence the convergence of this algorithm, which demonstrates the feasibility and generality of this algorithm in solving various kinds of RDMA resource allocation problems. When the above conditions hold, we can derive the convergence of V_k. Assume the inverse of function $F^{-1}(\cdot)$ exists, we can derive that $F^{-1}(\mathbf{z}^{k+1} - \mathbf{z}^*)$ is bounded and both r^{k+1} and $F^{-1}(z^{k+1} - z^k)$ go to 0, when $k \to +\infty$. Thus $F^{-1}(\mathbf{z}^*) = \mathbf{x}^*$, and Theorem 1.1 holds.

Since the inequality holds that $p^{k+1} - p^* \le u^{*T} r^{k+1}$, we have $lim_{k\to+\infty} p^k - p^* = 0$, i.e., the objective convergence. Thus Theorem 1.2 is proved.

Finally, with Lemma 2, the duality gap is proved to be zero for DRUM problem, so $u^k \to u^*$ when $p^k \to p^*$. Thus Theorem 1.3 is proved. □

Theorem 1 indicates that the ADMM-based method will converge to the optimality with enough k. Note that it implicitly assumes that the function $F(\cdot)$ is convex and invertible. Essentially, $F(\cdot) : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ correlates $\mathbf{x}$ and $\mathbf{z}$ and the physical meaning behind this function is how the generation of RDMA CQEs is related with WQEs. In the following sections, we will show that, for different RDMA operations, the relationship between CQEs and WQEs differs, yielding different modeling of $F(\cdot)$. Furthermore, different versions of $F(\cdot)$ lead to divergent designs and analyses of DRUM solutions.

C. DRUM for One-Sided RDMA Operations

We first evaluate the one-sided RDMA operations. In this case, the RDMA operations only generate CQEs on the local host where they are evoked. In this case, as described in Section II-C, the CQ polling rate and QP requesting rate have an approximately linear relationship that, after polling each CQE, the application generates multiple WQEs according to the WQE batch size. Thus, $F(\cdot)$ can be defined as

$$F(\mathbf{x}) = A\mathbf{x} = \mathbf{z}, A \in \mathbb{R}^{m\times m}, \tag{13}$$

where $A = diag\{a_1, a_2, \ldots, a_m\}$, and a_1 varies for applications with different WQE batch sizes. With the definition of such $F(\cdot)$, we analyze how to solve the local subproblem on $host_i$ under Algorithm 1. First, for the x-minimization step, we explore whether such a subproblem has a closed-form expression. The computation of each x-minimization step is guided by the following theorem.

Theorem 2: $\forall h_i$, the solution for x-minimization step is

$$x_i^{k+1} = \begin{cases} x_{i,j} = -\frac{1}{2}c_j + (c_j^2 - 4b_j d_j)^{1/2} b_j^{-1}, \forall x_{i,j} \notin I \\ \{x_{i,j}^{1/2} | b_j x_{i,j}^4 + c_j x_{i,j}^2 + x_{i,j} + d_j = 0\}, \\ \qquad\qquad \forall x_{i,j} \in I, \alpha = \frac{1}{2} \\ \{x_{i,j} | x_{i,j} = (b_j x_{i,j}^2 - c_j x_{i,j} - d_j)^{1/3}\}, \\ \qquad\qquad \forall x_{i,j} \in I, \alpha \neq \frac{1}{2} \end{cases} \tag{14}$$

where $b_j = -\rho a_j^2$, $c_j = \rho a_j z_{i,j}^k + a_j u_j^k$ and $d_j = \lambda_i^q$.

Guided by Theorem 2, we have a closed-form solution of x_i^{k+1}, and now we consider the CQ polling part $g_i(z_i)$, with the following theorem to guide the update of vector z_i^{k+1}.

Theorem 3: $\forall h_i$ and $\forall z_i$, the solution for z-minimization is

$$z_i^{k+1} = \{z_{i,j} | z_{i,j} = \lambda_i^c((\rho + u_j^k) z_{i,j} - 2\ z_{i,j}^{-3} - \rho a_j x_{i,j}^{k+1})^{-1}\}. \tag{15}$$

Due to space limitation, the proofs of Theorem 2 and Theorem 3 are attached in the Appendix. With the above theorems, we have demonstrated that all the subproblems of x and z minimization can be solved very efficiently by closed-form expressions or simple numerical methods.

Parallelism analysis. Such an ADMM-based method can be viewed as a variant of the method of multipliers where Gauss-Seidel iterations over x_i and z_i are used. Consequently, it naturally decomposes the joint minimization into two independent steps and breaks down the dependency between the steps into altering programming directions. Then, we focus on each minimization step to see whether it can be computed in parallel. For x-minimization step, as the objective $\sum_i^n \widetilde{f}_i(x_i)$ is the summation of decomposed functions $\widetilde{f}_i(x_i)$, which is separable. For the regulation term $(\rho/2)||F(\mathbf{x}) - \mathbf{z}||_2^2$, as $A = diag\{a_1, a_2, \ldots, a_m\}$ and assumed to be identical across all hosts, $F(\mathbf{x})$ can be easily decomposed to $F(x_i) = Ax_i = \sum_{j=1}^m a_j \cdot x_{i,j}, \forall i$. Then the regulation term can be rewritten to $(\rho/2)||F(x_i) - z_i||_2^2$. Therefore, x-minimization controls only a subset of variables x_i on each *host*$_i$, and observes its local constraints q_i and c_i, it can be computed locally on *host*$_i$. For the z-minimization step, applying similar decomposition, $g_i(z_i)$ and the constraints are also independent and local on *host*$_i$. Therefore, the optimization of each x_i and z_i can work in a distributed manner. For the dual update step, it collects the value x_i and z_i from hosts, conducts a simple summation computation and then broadcasts the updated dual value to hosts. Therefore, the algorithm can work independently for each host i to calculate x_i^{k+1}, z_i^{k+1} in each iteration k, so that it can work in a distributed manner.

D. DRUM for Two-Sided RDMA Operations

We now explore the resource sharing problem for two-sided RDMA operations. Theoretically, it can be viewed as an extension of the problem (6), where the main difference is the modeling of $F(\cdot)$. Revisiting the definition of $F(\cdot)$, it captures the relations between WQEs and CQEs. As for two-sided RDMA operations, the WQEs and associated CQEs will be generated on both the sender and receiver sides and consume the queue resources on both hosts. First, we analyze the partial of z_i incurred by receiving WQEs, denoted as z_i'. On each host l, we define a matrix $B^l \in \mathbb{R}^{m \times n}$, where each element B_{ji}^l represents the proportion of traffic that application j on host l send to the destination host i. Such traffic raises the CQEs processing on the destination host. Therefore, for each application j on host i, it can receive traffic r_j from other host, which can be defined as $r_j = a_j \sum_{l=1}^n s_j \cdot B_{ji}^l$. Expanding r_j to the host i, we can derive z_i' as

$$z_i' = A^i \sum_{l=1}^n x_l \circ b_i^l, \quad \forall i, \tag{16}$$

where the operator $\circ$ is an element-wise multiplication to compute the Hadamard product of two vectors and $b_i^l = [B_{1i}^l, B_{2i}^l, \ldots, B_{mi}^l]$ is the ith column vector of B^l. Besides, z_i in the two-sided RDMA operations cases should also include the local CQEs incurred by sending WQEs, denoted as z_i''. z_i'' is the same as in the one-sided case, that $z_i'' = A^i \cdot x_i$. Combining them together, we can derive the correlation between x_i and z_i by the following:

$$z_i = z_i' + z_i'' = A^i \sum_{l=1}^n (x_l \circ b_i^l) + A^i x_i, \quad \forall i. \tag{17}$$

Therefore, for the two-sided operation case, the original DRUM optimization problem becomes:

$$\begin{aligned} \min \ & \sum_{i=1}^n \tilde{f}_i(x_i) + \sum_{i=1}^n g_i(z_i), \\ s.t. \ & h_i^q(x_i) \leq q_i, \quad \forall i \in \mathbb{Z}_n, \\ & h_i^c(z_i) \leq c_i, \quad \forall i \in \mathbb{Z}_n, \\ & A^i \sum_{l=1}^n (x_l \circ b_i^l) + A^i x_i = z_i, \quad \forall i \in \mathbb{Z}_n. \end{aligned} \tag{18}$$

Intuitively, this problem can still be solved by the standard ADMM method. However, due to the dependence of variables in the constraint $A^i \sum_{l=1}^n (x_l \circ b_i^l) + A^i x_i = z_i$, we can see that the x-minimization step requires all VMs v_i to jointly solve a global optimization problem due to the penalty term $(\rho/2)||A^i \sum_{l=1}^n (x_l \circ b_i^l) + A^i x_i - z_i^k||_2^2$. To facilitate the analysis, we reorganize the traffic matrix B to:

$$P^i = \begin{pmatrix} a_1^i B_{11}^i & a_1^i B_{12}^i & \ldots & a_1^i B_{1i}^i + a_1^i & \ldots & a_1^i B_{1n}^i \\ a_2^i B_{21}^i & a_2^i B_{22}^i & \ldots & a_2^i B_{2i}^i + a_2^i & \ldots & a_2^i B_{2n}^i \\ \vdots & \vdots & \ddots & \vdots & & \\ a_m^i B_{m1}^i & a_m^i B_{m2}^i & \ldots & a_m^i B_{mi}^i + a_m^i & \ldots & a_m^i B_{mn}^i \end{pmatrix}$$

Note that in the definition of P^i, only the ith column have extra a_1^i summation. Then we can rewrite the last constraint of problem (18) into a more compact manner:

$$\sum_{l=1}^n (x_l \circ p_i^l) = z_i, \tag{19}$$

where p_i^l is the ith column of P^l. Such that we can see the computation of x_i relies on the summation of x_i on all other hosts. Such coupling is undesirable for large-scale networks. Ideally, the x-update should be conducted independently by each v_i. Therefore, we aim to find a distributed solution for this problem.

The trick here is to introduce an auxiliary variable y_l^i and make a wise use of the structure of dual variables. Let $x_l \circ p_i^l = y_l^i$ and substitute it into problem (18). For clarity, we change

the subscript of x from i to l, then the original optimization problem can be rewritten:

$$\begin{aligned} \min \quad & \sum_{l=1}^{n} \tilde{f}_l(x_l) + \sum_{i=1}^{n} g_i(\sum_{l=1}^{n} y_l^i), \\ s.t. \quad & x_l \circ p_i^l = y_l^i, \quad \forall i \in \mathbb{Z}_n, \\ & h_l^q(x_l) \le q_l, \quad \forall l \in \mathbb{Z}_n, \\ & h_i^c(z_i) \le c_i, \quad \forall i \in \mathbb{Z}_n, \\ & \sum_{l=1}^{n} y_l^i = z_i, \quad \forall i \in \mathbb{Z}_n. \end{aligned} \tag{20}$$

After reformulating this optimization problem, we can derive the following steps of Algorithm 2 for solving the two-sided RDMA resource sharing problem.

Step 1, x-minimization step:

$$x_l^{k+1} = arg\min_{x_l}(\tilde{f}_l(x_l) + \frac{\rho}{2}\|p_i^l \circ (x_l - x_l^k) + d^k\|_2^2), \tag{21}$$

where $d^k = \frac{1}{n}(\sum_{l=1}^{n}(p_l^i \circ x_l^k) - z_i^k + u^k)$.

Step 2, z-minimization step:

$$z_i^{k+1} = arg\min_{z_i}(g_i(z_i) + \frac{\rho}{2n}\|z_i - \sum_{l=1}^{n}(p_i^l \circ x_l^{k+1}) - u^k\|_2^2). \tag{22}$$

Step 3, u-update step:

$$u_i^{k+1} = u_i^k + \sum_{l=1}^{n}(p_i^l \circ x_l^{k+1}) - z_i^{k+1}. \tag{23}$$

Algorithm 2 is a variation of standard ADMM algorithm described in equation (7) to (9)and its correctness is guaranteed by the following theorem.

Theorem 4: Algorithm 2 complies with the standard ADMM framework.

Proof 4: A preliminary of the proof is the lemma described in [18] that the dual variables are equal for all hosts during iteration. Let v_l^k be the dual variable of the subproblem on each host l for each iteration k. This lemma implies $\forall l, v_l^k = v^k$. With this lemma we have $\forall i, u_i^k = \sum_{l=1}^{n} v_l^k = nv^k$. By standard ADMM framework, the dual update step is formulated by:

$$v_l^{k+1} = v_l^k + p_i^l \circ x_l^{k+1} - y_l^{ik+1}. \tag{24}$$

Then we sum up equation (24) by l and substitute the constraint $\sum_{l=1}^{n} y_l^i = z_i$ into it, the dual update step becomes:

$$\begin{aligned} u_i^{k+1} &= u_i^k + \sum_{l=1}^{n}(p_i^l \circ x_l^{k+1}) - \sum_{l=1}^{n}(y_l^{ik+1}) \\ &= u_i^k + \sum_{l=1}^{n}(p_i^l \circ x_l^{k+1}) - z_i^{k+1}, \end{aligned} \tag{25}$$

which is exactly the same as the dual update step described in equation (23). Therefore, the dual step is proven to comply with the standard ADMM. Besides, with equation (24), we can also derive that the dual on each host is given by:

$$\begin{aligned} v_l^k = v^k = \frac{1}{n}u^k = \frac{1}{n}u^{k-1} + \frac{1}{n}(\sum_{l=1}^{n}(p_i^l \circ x_l^k) - z_i^k) \\ = v^{k-1} + \frac{1}{n}(\sum_{l=1}^{n}(p_i^l \circ x_l^k) - z_i^k). \end{aligned} \tag{26}$$

We then analyze the x-update step. Applying the standard ADMM framework to problem (18), we have:

$$x_i^{k+1} = arg\min_{x_i} \tilde{f}_i(x_i) + \frac{\rho}{2}\|p_i^l \circ x_l - y_l^{ik} + v_l^k\|_2^2. \tag{27}$$

In equation (27), we focus on the regulation term, which is $p_i^l \circ x_l - y_l^{ik} + v_l^k$. By substituting equation (26) into it, we have the following derivation:

$$\begin{aligned} & p_i^l \circ x_l - y_l^{ik} + v_l^k \\ &= p_i^l \circ x_l - y_l^{ik} + v_l^k - p_i^l \circ x_l^k \\ &\quad + y_l^{ik} + \frac{1}{n}(\sum_{l=1}^{n}(p_i^l \circ x_l^k) - z_i^k) \\ &= p_i^l \circ x_l - P_i^l \circ x_l^k + d^k, \end{aligned} \tag{28}$$

in which we define $d^k = \frac{1}{n}(\sum_{l=1}^{n}(p_l^i \circ x_l^k) - z_i^k + u^k)$.

Substituting the equation (28) into equation (27), we can see that it is exactly the same as the x-update step described in equation (21). Therefore, the x-update step is proved. Finally, for the z-update step, as we introduce an auxiliary variable y_l^i, we have:

$$\begin{aligned} y_l^{i,k+1} &= arg\min_{y_l^i} \\ &= g_i(\sum_{l=1}^{n} y_l^i) + \frac{\rho}{2}\sum_{l=1}^{n}\|y_l^i - p_i^l \circ x_l^{k+1} - v_l^k\|_2^2. \end{aligned} \tag{29}$$

Then, we focus on the term inside the norm operator $y_l^i - p_i^l \circ x_l^{k+1} - v_l^k$, together with the derivation in equation (24), we have:

$$\begin{aligned} y_l^i - p_i^l \circ x_l^{k+1} - v_l^k = -v^{k+1} = -\frac{1}{n}u^{k+1} \\ = \frac{1}{n}(z_i - \sum_{l=1}^{n}(p_l^i \circ x_l^{k+1}) - u^k). \end{aligned} \tag{30}$$

Thus, the Euclidean norm part of equation (29) becomes:

$$\sum_{l=1}^{n}\|y_l^i - p_l^i \circ x_l^{k+1} - v_l^k\|_2^2 = \frac{1}{n}\|z_i - \sum_{l=1}^{n}(p_l^i \circ x_l^{k+1}) - u^k\|_2^2.$$

Then, substituting $\sum_{l=1}^{n} y_l^i = z_i$, we have the z-update step:

$$z_i^{k+1} := argmin(g_i(z_i) + \frac{\rho}{2N}\|z_i - \sum_{l=1}^{n}(p_l^i \circ x_l^{k+1}) - u^k\|_2^2),$$

which is the same as the z-minimization step described in Equation (22). By now, we show that all three steps of Algorithm2 are mathematically transformed from the standard ADMM framework. Thus, Theorem 4 is proved. □

Parallelism analysis. First, for x-minimization step, we can analyze it as two components. In terms of the objective function, $\sum_i^n \tilde{f}_i(x_i)$ is the summation of decomposed subproblem $\tilde{f}_i(x_i)$, which is the same as the one-sided operation case. The difference lies in the regulation part which incorporates the dependent constraints among x_i and z_i on multiple hosts. With Theorem 4, we successfully decompose the coupled regulation term $(\rho/2)||A^i\sum_{l=1}^{n}(x_l \circ b_l^i) + A^i x_i - z_i^k||_2^2$ into a new one $\rho 2\|p_i^l \circ (x_l - x_l^k) + d^k\|_2^2)$. For the new regulation

function, the optimization relies on the variable x_l only and as $d^k = \frac{1}{n}(\sum_{l=1}^{n}(p_l^i \circ x_l^k) - z_i^k + u^k)$, the summation of $\sum_{l=1}^{n}(p_l^i \circ x_l^k)$ is the values already computed in the last round of iteration. For the z-minimization step, $g_i(z_i)$ are also independent among each host and its regulation part contains no coupling of variables. So $g_i(z_i)$ can be calculated by each host. Therefore, the computation of two terms x_i and z_i can be distributed across each host. For the dual update step, it collects the value x_l and z_i from hosts, conducts a simple summation and then broadcasts to all the hosts. From the u-update step described by equation (23), we notice that the u-update step can further be distributed to each host i, and collects only the intermediate values from host l where they have mutual two-sided communications. It is worth noticing that, in the former computation of the dual variable u^k for the one-sided operations, the global u^k is computed by collecting all the intermediate variables among all the hosts. Now, also with the fact that $\forall i, v_i^k = v^k$, u^k can also be computed by the distributed hosts. Therefore, for DRUM problem, we conclude that all the three key steps can be computed in a distributed manner, for both one-sided and two-sided operations.

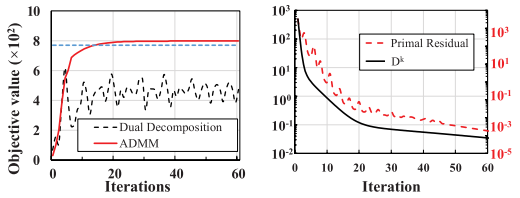
E. Simulations

To illustrate the basic behavior of the proposed algorithms, we report the simulation results for some large-scale instances of the DRUM problem. We implement our proposed algorithms in Python with CVXPY [19] package and embed them in multiple parallel processes to simulate the distributed network scenario. The difference with a real-world RDMA network is that the communication between multiple processes is using cross-process communication instead of real data transferring on the substrate network. In the simulation, the utility and cost functions of problem (6) are set as the concrete context of RDMA communication. For different applications, we randomly generate the coefficients that correlate the CQ rates and QP rates, and encode them in matrix A. We set the number of host in data centers $n \in \{10^2, 10^3, 10^4\}$ and each host can run up to m applications, $m \in \{50, 5 \times 10^2, 5 \times 10^3\}$. With this setting, we simulate a large-scale data center environment. Each application's communication is randomly generated. The maximum WQE processing rate q_i is set to be 20 to 100 million operations per second and the CQE processing rate c_i is set to be 15 to 50 million operations per second. The maximum concurrent number of RDMA connections $\tilde{k}$ is set to be 20 to 50. These numbers represent state-of-the-art RDMA configurations in real-world data centers.

For the numerical part, we set the penalty parameter $\rho = 10^{-3}$ of the ADMM algorithm after an empirical sweep of $\rho \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. The initial point of u^0 and z^0 are set to be zero. The evaluation metric is the number of iterations, which is platform-independent.

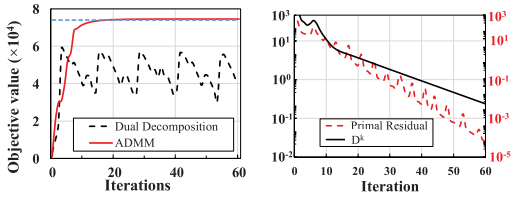
For comparison, we also implement the conventional Dual Decomposition approach with sub-gradient methods to solve the standard NUM described in problem (1). The step size ρ_k is chosen following the commonly accepted diminishing step size rule [17], with $\rho_k = 10^{-3}/\sqrt{k}$.

Convergence. We first evaluate the convergence of DRUM algorithms under varying n and m. From Figs. 4(a) and 5(a), we observe that our algorithm converges very fast in less than 20 iterations for all cases. Figs. 4(b) and 5(b) depict the trajectory of the primal residual defined in the proof of Theorem 1. From the figures, we observe that the primal residual becomes less than 1 when $n = 100$, $m = 50$, and less than 10 when $n = 10^3$, $m = 500$ after 10 iterations. We also observe that D^k is indeed non-increasing and Lipschitz continuous. From the simulation results, we also find out that the convergence rate is independent of the problem size by the fact that the convergence behaviors of the proposed algorithm are the same in all cases when $n \in \{10^2, 10^3, 10^4\}$, $m \in \{50, 5 \times 10^2, 5 \times 10^3\}$. This means that our algorithm is scalable for solving large-scale problems. The blue dotted line in Fig. 4(a) and Fig. 5(a) is an 99.5% approximated optimal value for the objective. The algorithm stops at 17-th iteration when $n = 10^3$, $m = 500$. These results confirm that our algorithm generates a solution with high accuracy even faster.



(a) Comparison of objective value. (b) Comparison of the primal residual and D^k.

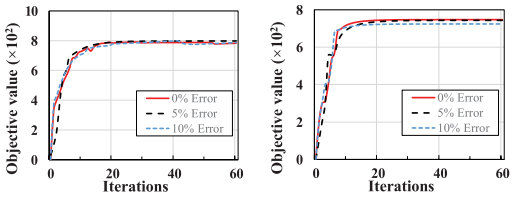
Fig. 4. Convergence analysis when $n = 100$, $m = 50$.



(a) Comparison of objective value. (b) Comparison of the primal residual and D^k.

Fig. 5. Convergence analysis when $n = 10^3$, $m = 500$.

Fault-tolerance. In the real-world RDMA environment, since our algorithm runs in a distributed manner on each host, the missing of intermediate results between the iterations can happen. For example, packet losses will cause intermediate results missing as they are contained in the RDMA messages. This is especially common when RoCE (RDMA over Converge Ethernet) is used without Priority Flow Control (PFC) [2]. In this RDMA usage scenario, zero packet loss is no longer guaranteed. Another example is that, when the host is overloaded, some intermediate results can not be computed within a certain deadline. In this case, such straggler results should be discarded. Therefore, the fault-tolerance capability is important that even if some intermediate results are missing, the algorithm can still converge to the optimal solution. In this simulation case, we demonstrate the fault-tolerance ability of our algorithm. We randomly introduce the failure by manually dropping the intermediate results with different probabilities. Fig. 6 plots the convergence with different failure probabilities from 0% to 10%. From the figures, we observe that the impact of failures is insignificant. Although the random failure causes the solution quality to degrade at the early stage (before the 10-th iteration), when $n = 100$, $m = 50$, the optimization gap is at most 1.5% compared to the solution with $p = 0\%$, and ceases



(a) $n = 100$, $m = 50$ (b) $n = 1000$, $m = 500$

Fig. 6. Fault-tolerance analysis.

to 0.2% after 20 iterations. Fig. 6(b) demonstrates the situation when $n = 1000$, $m = 500$, our algorithm has less than 0.1% optimality loss and essentially the same convergence speed with 10% failure rate. This fact indicates that our algorithm is robust against the faults. This feature is especially desirable when deploying the algorithm in real-world RDMA data centers.

Stopping criterion. In practice, it is usually unnecessary to derive an exact optimal value for the traffic allocation problem, so we discuss a practical way to stop the iteration when finding a high accuracy solution. Following [18], we can record the primal and dual residuals r^k and s^k in each iteration k, which measures the primal feasibility and the difference between the current and previous iterations. We can define a simple criterion for terminating iteration when

$$||r^k||_2 <= \epsilon^{pri}, \quad ||s^k||_2 <= \epsilon^{dual} \tag{31}$$

where ϵ^{pri} and ϵ^{dual} are primal and dual tolerances, respectively. We can normalize both of these quantities to network size by $\epsilon^{pri} = \epsilon^{dual} = \epsilon^{abs} n$, for some absolute tolerance $\epsilon^{abs} > 0$. The choice of ϵ^{abs} is usually a heuristic in practice with sensitivity tests.

IV. System Design and Implementation

Inspired by the host-based software design for RDMA resource management [4], [20], DRUM designs and deploys distributed agents on each RDMA-enable host, so as to provide end-to-end resource sharing and performance guarantee. The essential of DRUM agent is a kernel-level RDMA indirection module and it acts as a delegator for the applications on the host, and manages the low-level RDMA resources, i.e., QPs and CQs. As illustrated in Fig. 7, the DRUM agent is a lightweight software module that lies between RDMA applications and the RNIC driver. It handles applications' traffic through 3 main modules: an IO monitor, a DRUM scheduler, and a DRUM poller.

IO monitor is designed to enable the resource sharing among multiple RDMA applications. It is the key module of the DRUM indirection layer. Essentially, it is a dedicated thread that maintains a virtual request pool shared by all the hosted applications. In order to support scalable and efficient transport, the IO monitor follows a lock-free design. Instead of allowing applications using the `mutex_lock` to access the RDMA on-board resources, when each connection generates WQEs, IO monitor maps the WQEs in the application memory space to the virtual request pool. When the WQEs are generated, rather than directly accessing the underlying QPs, they are redirected to the virtual request pool. These WQEs are further processed by the asynchronous model that they will be actively fetched by the DRUM scheduler. Such a lock-free design is implemented by listening to the `fd` which contains

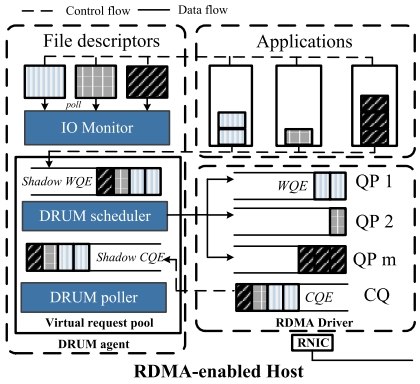


Fig. 7. Architecture of DRUM indirection module.

the WQEs by the I/O multiplexing technique `epoll`. Based on `epoll`, the IO monitor is able to listen up to 65536 `fd`s. This design is different from the past practice [3], [4], where developers allocate one QP for each application. The reason behind is that, for some specific type of RDMA operations, there are some limitations of QP sharing. For example, QPs of RC type only support one-to-one communication. To facilitate this, DRUM implicitly converts the shared QP to UD type, and such conversion is transparent to applications. Then we arrange different connections to the same target node to share one QP and the multiplexing ratio is set between 4 to 10 in our settings.

Another important resource handled by IO monitor is the RDMA Memory Region (MR). In native RDMA, each application registers a dedicated memory region to manipulate WQEs for transmission. In DRUM design, the IO monitor module pre-registers a global memory buffer to shadow the memory regions of applications. When applications need to register a memory region, the IO monitor allocates a privileged memory space from the global memory buffer, remaps the virtual pages of application buffer to the physical pages of system buffer. Such an MR sharing mechanism further eliminates the frequent memory allocation and mapping overhead, such as CPU circle waste and memory fragmentation.

DRUM scheduler decides the rates and the dequeue sequences of WQEs in the virtual request pool. The first fundamental enabler of resource sharing is to identify the WQEs for each application, and to further manage the state of each connection. DRUM scheduler allocates a Connection ID for each application. The Connection ID is extracted from `fd` associated with each application when creating the QP. Then the Connection ID is stored in the `wr_id` domain in each WQE. For two-sided operations, the IO monitor should also be notified with the Connection ID for remote applications, so as to facilitate the DRUM algorithm. The recording of a remote Connection ID is implemented by exploring the usage of the `imm_data` domain in Receive Completion elements. The `imm_data` domain is a 32 bits number in network order which can be sent along with the payload of Send Work Request to the receiver and be placed in the Receive Completion at the side of the receiver. Therefore, the remote scheduler can write Connection ID in the `imm_data` domain and the local Poller is able to find the remote Connection ID in the `imm_data` domain of the Receive Completion. The IO monitor then works collaboratively with the DRUM poller to identify each Receive Completion according to the remote

Connection ID. Then, the DRUM scheduler is dedicated to fetching the WQEs based on the computed rates for each application. Specifically, it first computes the rates of fetching WQEs for different applications, fetches the WRs actively and sends them to the corresponding physical QPs on RNIC.

DRUM poller maintains and manages the completion queue of CQEs. In our design, all connections on the same host share one CQ. The implementation of sharing CQ is by using direct memory mapping, which is similar with that of QP sharing. When their messages finish transferring, a CQE is generated into the shared CQ. The DRUM poller has one dedicated thread which actively polls the CQEs. Since the CQE contains the connection identifier, the poller is able to know which QP and which application it is associated with. Finally, the poller pushes the received CQEs to the corresponding applications by the decided rates. This design utilizes the CQ more wisely and hereby maximizes the resource sharing.

Deployment considerations In the RDMA-enabled data center, the DRUM agents are deployed on each host. On each host, the agent first executes independently to solve a small subproblem with only local information, determining the initial queue usage rates for the co-resident applications. In many cases, the subproblem is solved by closed-form formulations given in Section III. Furthermore, we have explored the internal parallelism to solve x_i so as to implement Algorithms 1 and 2 in a more computation-friendly manner. After each iteration, they get feedback from the coordinator and update the rates for the next decision round. The coordination is also not computation-intensive, because it simply accumulates the collected values. As we have analyzed, the coordinator can be also deployed in a distributed manner. Thus, it either collects the intermediate results from the local host or from hosts where they have mutual two-sided communications. The intermediate values of x_i^k, z_i^k, u^k are stored as floating numbers and their sizes are usually several Bytes. These values are encoded as special control packets for network management purposes. If the operators sacrifice some precision, these values can be further encoded in the header of regular data packets. Furthermore, the DRUM algorithm is inherently fault-tolerant, so that it works even when some of the intermediate results are missing. In practical deployment, the coordinator is set with a window where it collects and sums up the intermediate results, and for any timeout control messages, it reuses the last-round values. In general, DRUM works very similar with most existing network protocols, e.g. TCP/IP congestion control, where all the senders initialize a congestion window size and gradually converge to a stationary according to the network congestion signals issued by the switch, e.g. packet loss or ECN marking [21]. In the TCP congestion control context, the ECN signaling can be viewed as a variation of the dual coordination in DRUM. Besides, the computational iterations can be further accelerated by emerging network hardware such as FPGA NIC and P4 switches [22], [23], enabling the network to make wiser resource management decisions.

V. Experiment

A. Experiment Settings

We evaluate DRUM's performance in a real-world RDMA network environment through extensive testbed experiments.

Testbed. The testbed is built with 7 nodes, each of which is equipped with two 2.2GHz Intel Xeon E5-2650 v4 processors and 64 GB of memory. All these servers are installed with Ubuntu 16.04 and are equipped with a Mellanox ConnectX Series RNIC (40 Gbps over InfiniBand) to enable RDMA.

Application. We implement an RDMA-enabled Key-Value (KV) store service. On client nodes, each client creates multiple connections requesting data in the KV store through RDMA network. Each connection requests for data with random data sizes ranging between $[32B, 1MB]$. In our implementation of KV, as the stored values are of small sizes, it is recommenced [24] to use one-sided RDMA operations to avoid frequent notifications on both sides. For comparison, we also implement KV with two-sided RDMA operations.

Redis [25]is a widely-used open-source memory cache software. We rewrite its communication module. The original communication of Redis is done mostly with Redis.c in the message event handling function processEventOnce(void* task_id_) and processEvents(void* connectionhandler). For comparison, we replace its previously hardcoded write_with_imm() functions by one-sided Write() and two-sided Send/Rcv() verbs, respectively. To synthesize the workloads, we modify the code of benchamark.c to generate the desired workloads and the data cached in Redis is in the range of [10B, 512MB].

Benchmark. For comparison, we have implemented the following benchmarking mechanisms:

(1) *RAW*: It represents the raw RDMA-based primitives and the standard operation in the RDMA (same as in FaRM [26]). In this case, each connection manages its own QP and WQEs.

(2) *SHARE*: It denotes the connection grouping mechanism proposed in [4] and [3], where the connections designated for the same host share one QP for resource multiplexing. As the authors suggest, the multiplexing ratio of QP is set to be 5.

(3) *DRUM*: The implementation of our method, with controlled RDMA resource sharing.

Some key experimental results are highlighted as follows:

- For individual 32B packet under varying background traffic, *DRUM* completes the WQE $3.1\times$ faster than *RAW* and $1.7\times$ than *SHARE*. (Test case 1)
- In a setup with the dynamically changing context, *DRUM* proactively allocates the RDMA resources and provides a consistent performance, where the largest performance improvement reaches 98.1% and 64.1% in terms of latency and throughput, respectively. (Test case 2)
- For real-world applications and workloads, *DRUM* is able to achieve 39.2% higher throughput for KV and up to 142.6% latency reduction for Redis. (Test case 3)
- *DRUM* only introduces less than 30% overhead of one CPU core when serving over 250 connections on one host. (Test case 4)

B. Experimental Results

Test Case 1 (**Packet level performance**). To evaluate whether our model could provide fine-grained resource scheduling, we test the completion time of a WQE to continuously request for 32B data from the KV store server, while the background connection varies its WQE batch size from 20 to 100. Fig. 8 demonstrates the distribution of the WQE completion time.

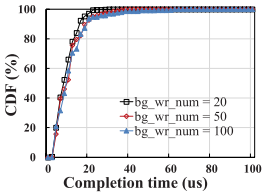


Fig. 8. WQE completion time of a 32B message for KV.

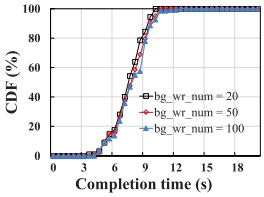
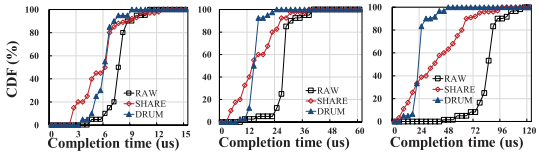
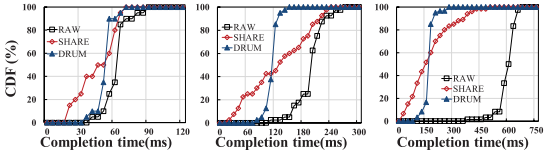


Fig. 9. 1GB flow completion time of a sequence of WQEs for KV.



(a) $cc = 20$ (b) $cc = 50$ (c) $cc = 100$

Fig. 10. Comparison on 32B packet completion time.



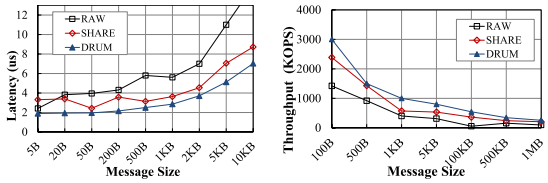
(a) $cc = 20$ (b) $cc = 50$ (c) $cc = 100$

Fig. 11. Completion time of 1MB flows (32B/packet).

We observe that the single WQE completion time is unaffected by the background traffic that with the number of background connections increasing from 20 to 100, the 80% completion time of the packets remains within 20 μs. This case demonstrates that DRUM can protect the RDMA flow from starving by HOL blocking, even with heavy background traffic.

We then compare *DRUM* with the benchmarking mechanisms. We adjust the number of concurrent connections cc in the host. cc is set as $\{20, 50, 100\}$. The evaluation results are depicted in Fig. 10. When $cc = 20$, the 80% completion time of the single WQE are 7.1 μs, 7.2 μs, 8.8 μs under *RAW*, *SHARE* and *DRUM*, respectively. When cc increases, *RAW* does not work properly. For example, the 80% completion time under *RAW* degrades from 26.5 μs to 90.4 μs, when cc increases from 50 to 100. The reason is that when the number of connections increases, RNIC will be occupied by storing and replacement of QP states. Although *SHARE* employs the QP sharing and performs better than *RAW*, it suffers from the long tail distribution of the completion time. From our further analysis, this is because of the lock contention on the shared QP. On the contrary, *DRUM* accesses QP by the single DRUM agent in a lock-free manner. It completes the WQEs in 14.5 μs and 24.0 μs, when $cc = 50$ and $cc = 100$. It is worth noting that when $cc = 100$, *DRUM* completes the WQEs 3.1× faster than *RAW* and 1.7× than *SHARE*.

Test Case 2 (**Flow level performance**). In this test case, we evaluate the performance of *DRUM* when transferring a large data file. The foreground traffic transmits 1MB message through one WQE each, and continuously transfers 1GB of data in total. The background connections request for Redis data at random sizes between $[1KB, 10MB]$ and the batch size parameter is set between $[20, 100]$. We repeat the flow transfer for 1,000 times, and Fig. 9 illustrates the distribution of the flow completion time. We see that the completion time of foreground traffic is unaffected by the background traffic that the distributions of flow completion time remain stable. The reason is that DRUM jointly controls the resource scheduling on CQ and QP, providing an enhanced resource scheduling.



(a) Throughput comparison (b) Latency comparison

Fig. 12. Performance comparison with varying message sizes.

We then evaluate *DRUM*'s performance by transferring a 1MB flow that is composed of a sequence of 32B WQEs. The evaluation results are depicted in Fig. 11. When $cc = 20$, the 80% completion time are 53.1 ms, 82.8 ms, 62.9 ms under *RAW*, *SHARE* and *DRUM*, respectively. When the number of concurrent connections increases, where the contention on CQ polling is more severe, *DRUM* outperforms all other mechanisms. When $cc = 100$, *DRUM* completes the WQEs 3.75× faster than *RAW* and 1.4× than *SHARE*.

We then dynamically adjust some context features, i.e., average message sizes. In this experiment, we measure the average latency of all messages and the system throughput under *DRUM* against *RAW* and *SHARE*. The result with changing message sizes can be seen in Fig. 12. In general, *DRUM* can provide a stable performance that the latency increases incrementally subjected to the message size growth. When the packet size is small in the range [5B, 200B], *DRUM* has an advantage of 43% to 98.1% improvement. When the packet sizes are medium, *SHARE* achieves similar performance with *DRUM*. However, when packet sizes go large, *SHARE* becomes less effective, having an overhead of 14.7%- 27.8% than *DRUM*, in terms of the average latency.

In the aspect of system throughput, we measure the amount of operation the RNIC could process in a second. As is shown in Fig. 12(b), the throughput of *DRUM* decreases gradually and consistently when the packet sizes grow from 100B to 1MB. When the message size is medium from 1KB to 5KB, the throughput of *SHARE* is close, which means that QP sharing is an effective method. In particular, when packet sizes grow to larger than 500KB, the throughput of *RAW* drops dramatically to 10.3 to 15.6 KOPS, while *DRUM* achieves 14 to 25.6 KOPS in the same scenario, which is a 35.9% to 64.1% performance improvement.

Test Case 3 (**Overall system performance**). We evaluate *DRUM* by measuring the overall system performance, in terms of throughput and latency. We repeat the experiments for ten times and record the average performance, associated with the 90th percentile performance (upper whiskers) and the 10th percentile performance (lower whiskers). In this test case,

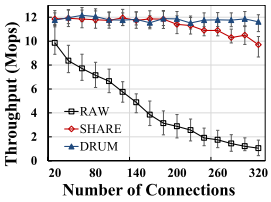


Fig. 13. Throughput with increasing number of connections.

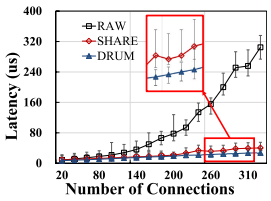
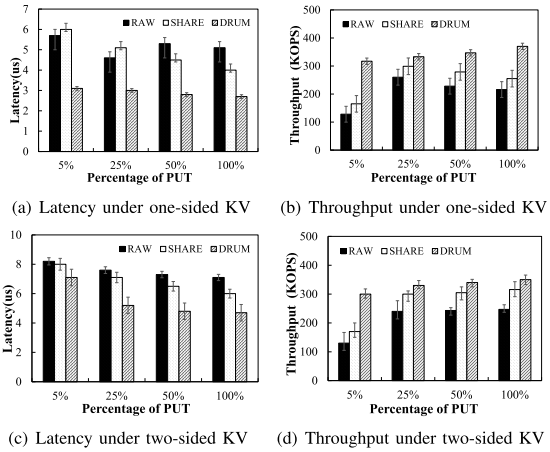


Fig. 14. Latency with increasing number of connections.



(a) Latency under one-sided KV (b) Throughput under one-sided KV

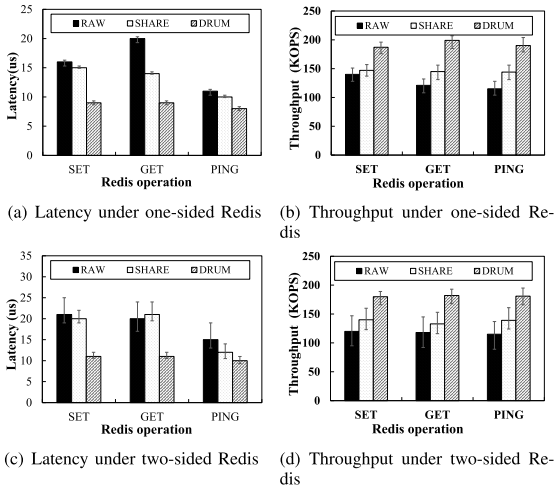
(c) Latency under two-sided KV (d) Throughput under two-sided KV

Fig. 15. Performance comparison of the KV application.

we vary the concurrent number of clients requesting data from the KV store from 20 to 400, which are distributed evenly to 6 physical servers. The clients request data at random sizes between $[32B, 1MB]$. As shown in Fig. 13, for KV application, *DRUM* significantly outperforms all other mechanisms in terms of the average throughput. *SHARE* performs better than *RAW*, but when connections increase, the lock contention on shared QP becomes the bottleneck, thus hurting the performance of applications. When the total number of connections is larger than 300, the average throughput under *DRUM* is $1.8\times$ greater than *SHARE*. In Fig. 13, we observe that the performance variation under *DRUM* is much smaller, demonstrating that our method can deliver consistent and stable performance enhancement. Fig. 14 shows the average latency of messages. We record the latency of each batch of WQEs. From the results, we observe that *DRUM* is capable of keeping most of the requests within extremely low latency of less than 18 μs. When the number of connections increases, the network is fully saturated and the performance of *SHARE* degrades, the average latency is 67.3% higher than *DRUM* when the total number of connections is larger than 300. Meanwhile, when taking a closer look at the variation of the latency, we also observe that the 90th percentile tail latency is much higher under *SHARE* than that under *DRUM*. The reason is that *SHARE* does not consider the impact of CQE contention issue in the CQ that the completion signals of some messages are delayed. Consequently, the application-perceived latencies of these messages become high. The tail latency issue is especially harmful to latency-sensitive applications in data centers. On the contrary, as the results in Fig. 14 indicate, *DRUM* has the benefit of providing stable and consistent low latency for all messages.

To further evaluate the performance of *DRUM* under application-specific workloads, we implement two representative RDMA applications, KV and Redis, by using one-sided and two-sided verbs, respectively. Throughout the evaluation, we adjust the workloads for different applications. As for KV, we change the workloads to be read-intensive (95% GET, 5% PUT) or write-intensive (50% GET, 50% PUT). Fig. 15(a) and (b) demonstrate the latency and throughput of one-sided KV. Under *DRUM*, the latency remains as low as 3.1 μs, while the performance under *SHARE* fluctuates between [4.2μs, 6μs]. We observe that both *SHARE* and *RAW* perform worse than *DRUM*. In detail, the latency under *SHARE* is 30.1% higher than that under *DRUM* and latency under *RAW* is 52% higher. In particular, for the 100% PUT workloads, the latencies of all KV operations under the three benchmarking mechanisms are 5.2 μs, 4.1 μs and 3 μs, respectively. Similar results are also shown in Fig. 15(b) in terms of throughput. Under *DRUM*, both read-intensive and write-intensive workloads achieve more than 320Kops throughput regardless of the workload composition. By contrast, throughputs under *SHARE* and *RAW* are correlated with the workload composition. For the read-intensive workloads, *SHARE* only achieves 165 Kops throughput which is only 48.9% of the *DRUM*'s throughput. The reason is that the sender side RNIC resource contention also negatively impacts the performance. As shown in Fig. 15(b), *DRUM* achieves on average 28.1% higher throughput than *SHARE* and 39.2% higher throughput than *RAW* for different types of workloads. Fig. 15(c) and (d) demonstrate the latency and throughput of two-sided KV. First, for the aspect of latency, we observe that the overall latency is higher than that with one-sided KV. This is because, for the KV application, PUT and GET operations need just one-shot communication, which is suitable for one-sided verbs like Write. Two-sided RDMA operations like SEND/RECV in this use case inherently incur higher latency. The results are consistent with the design guidelines for RDMA-enabled KV applications [1]. Moreover, we observe that the benefit of *DRUM* degrades in the two-sided KV case that it has only 10% to 21% latency reduction. The reason is that, with two-sided verbs, the resource allocation algorithm of *DRUM* also becomes more complicated by incurring more computation in each iteration, resulting in a smaller performance gain.

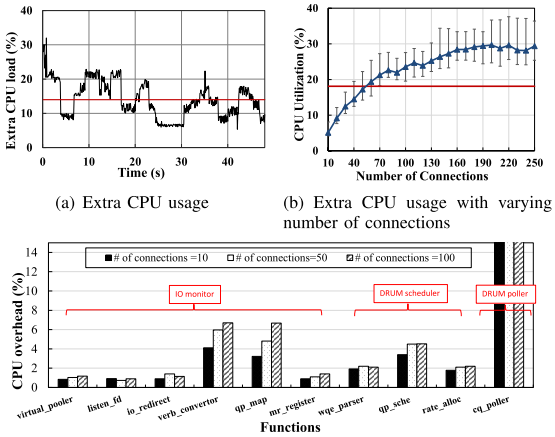
Then, we turn to a more complicated application Redis. In this experiment, we evaluate Redis with one-sided and two-sided RDMA implementations, respectively. We test the basic Redis operations to continuously push data with size uniformly distributed in the range [10B, 512MB] into the remote memory cache, pull data from the remote server, or just probe the cache for data validation. The results are shown in Fig. 16. When using *RAW*, the average latency for SET, GET and PING are 16, 21.8 and 12.4, respectively. For comparison, the average latencies under *DRUM* are 9.1, 9 and 7.1 μs, respectively, with an improvement of 75.9.7%, 142.6% and



(a) Latency under one-sided Redis (b) Throughput under one-sided Redis

(c) Latency under two-sided Redis (d) Throughput under two-sided Redis

Fig. 16. Performance comparison of Redis application.



(a) Extra CPU usage (b) Extra CPU usage with varying number of connections

(c) Detailed CPU overhead

Fig. 17. Evaluations on extra CPU usage.

74.6%, respectively. The reason is that *DRUM* reacts to the changing of data size by adjusting their requesting rates. Even for PING operations with less variation, *DRUM* is still better, because it is able to avoid large messages from blocking small messages, which is beneficial for control messages such as probing packets. A similar phenomenon can be seen for the throughput comparison results, where *DRUM* achieves 37.8% to 48.1% higher throughput under various workloads. We then evaluate the case with two-sided Redis. From Fig. 16(c) and (d), we observe that the performance gain of *DRUM* is similar with the one-sided case. For each query message, Redis has additional operations like hashing, and the data stored in Redis can be larger than that in KV. Therefore, the time budget for each message is large enough for *DRUM* algorithm to compute the resource allocation. Consequently, the extra computational overhead of *DRUM* for two-sided operations can be overlooked. Moreover, because *DRUM* splits the large message into small ones, two-sided verbs perform similarly with one-sided for medium-sized messages. Therefore, for Redis applications, either the one-sided or two-sided RDMA implementation is suitable.

Test Case 4 (**Overhead**). We evaluate the overhead of *DRUM* by collecting the CPU usage with the CPU profiling tool `perf top` which measures the CPU usage of each function. Before analyzing the evaluation results, we note that in the design of *DRUM*, the busy-polling mode of RDMA is required. With this setting, the system actively polls and schedules the CQE in the completion queue. The benefit of this setting is to reduce the latency caused by CQE event triggering and handling, but the overhead is that one CPU core will be delicately occupied for polling. This setting is also enabled in many real-world RDMA use cases [1]. Therefore, in the test case, we measure the extra CPU usage on other CPU cores except for that dedicated core. The measurement results are shown in Fig. 17. First, we randomly generate new RDMA connections, and each connection is active for sending a random amount of messages. We record the CPU usage for a few seconds. As illustrated in 17(a), the average CPU usage is 13.5%. Since *DRUM* algorithm is triggered whenever a new connection is established, so the peak usage is when connections are initializing. In Fig. 17(a), at the initiation phase, the CPU cost is high for computing the resource allocation and allocating the resource mapping for shared memory region. During the transmission, the CPU usage gradually decreases, since the computation of *DRUM* is simple by closed-form equations and the algorithm can converge very fast. The fluctuation is due to some new connections being established. We then measure the CPU usage with a varying number of connections. From Fig. 17(b), we can see that the usage is going higher with the increasing number of connections. This is reasonable because more connections would cost the algorithm to solve larger problems. Another observation is that the increment of curve gradually slows down. This is because *DRUM* serves only the top-k connections to avoid RNIC cache thrashing. Finally, we show how much CPU time each specific function uses in Fig. 17(c). From the results, we observe that the DRUM poller consumes 100% usage for one core. This is because *DRUM* uses the busy-polling mode to schedule the CQEs. For the other two main components, the IO monitor consumes more CPU resources. The most resource consuming functions are `verb_convertor` and `qp_map`. `verb_convertor` is responsible for accessing the `imm_data` domain of each request and identifying the corresponding connections. `qp_map` consumes CPU because it maintains a shared memory region and is responsible for mapping and managing this region for different connections. For DRUM scheduler, the main overhead lies in the allocation rate computation by using our proposed algorithm. The core function `qp_sche` consumes approximately 4.22% CPU usage. The low CPU usage is due to the closed-form computation and fast convergence of our algorithm in many cases. Note that, from the results, we also observe that the CPU overhead does not increase much when the number of connections increases, which shows the scalability of our method.

VI. Related Work

RDMA suffers from the unfairness issue and performance degradation in the shared data center network as reported in [3], [4], [5], and [27]. Addressing this issue, various resource management mechanisms for RDMA are proposed. Freeflow [28] proposed a virtualization based software solution to isolate the performance of RDMA-enabled applications.

Chen et al. [3] designed the connection grouping mechanism, allowing multiple applications to share one QP. LITE [4] allowed applications to share resources by establishing a shared memory pool in the kernel. Qiu [5] et al. proposed a pooling mechanism between applications and RNIC driver to manage RDMA resources. FaRM [26] applied QP sharing in the kernel. Wang [24] optimized parameter settings for applications when sharing the same network. Lu et al. [29] explored the capability of RDMA in the multipath network scenario, and addressed the issue of strict on-chip resource of RNIC. 1RMA [30] presented a connection-free design of new RDMA architecture where software handles congestion control, and applications handle failure recovery and inter-operation ordering as needed. Recently, Wang et al. [31] proposed a remote-side offloading method to tackle the RDMA scalability issue. Wang et al. [20] proposed Nem which alleviates the resource contention at RNIC cache through asynchronous resource sharing. While existing works provide substantial system design insights to the RDMA resource sharing issue, existing proposals lack sufficient theoretical analysis, such that they are unable to guarantee the optimality of resource scheduling.

Theoretically, the network resource scheduling optimization was modeled as a NUM problem [9], [11], [12]. Previous efforts [11], [12], [13] in resource management for TCP/IP networks have been successful in deriving NUM-based resource scheduling solutions. For example, Guo et al. [32] proposed an instance of NUM to optimize the bandwidth allocation. Also, [8] provided a relaxation-based approach to solve the NUM with non-convex utility functions. While none of prior methods considered RDMA, we have proposed a preliminary version of NUM model and designed a distributed and optimal solution [33] based on ADMM [18], characterizing the inherent complexities of RDMA networks. Compared with traditional Dual Decomposition [11] based method to solve NUM, ADMM is easier to tune and guarantees fast convergence both theoretically and in practice. In this paper, we analyze more comprehensively on both one-sided and two-sided RDMA operations and address the theoretical challenges of the underlying problem.

VII. Conclusion

In this paper, we have introduced a distributed solution to optimal RDMA resource scheduling and presented how to model, analyze and implement it with the real-world RDMA network deployment considerations. Characterizing the inherent complexities in RDMA networks, we have abstracted the problem as a multi-block utility maximization problem with coupling variables. To efficiently solve it, we have presented a distributed and modular algorithm based on ADMM, and demonstrated the parallelism and convergence guarantee through theoretical analysis. We have implemented a kernel-level software module to enable the end-to-end optimal resource sharing in the RDMA network and considered its real-world deployment. Through large-scale simulations and testbed experiments, we have shown that, compared with the state-of-the-art methods, our method has a number of unique advantages, such as achieving higher network utility and higher overall throughput when multiple applications share the RDMA network. Such advantages are highly desirable for resource management in large-scale shared RDMA networks.

References

[1] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 437–450.

[2] R. Mittal et al., "Revisiting network support for RDMA," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 313–326.

[3] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on reliable connection with efficient resource sharing," in *Proc. 14th EuroSys Conf.*, Mar. 2019, p. 19.

[4] S.-Y. Tsai and Y. Zhang, "LITE kernel RDMA support for datacenter applications," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 306–324.

[5] H. Qiu et al., "Toward effective and fair RDMA resource sharing," in *Proc. 2nd Asia–Pacific Workshop Netw.*, Aug. 2018, pp. 8–14.

[6] *Mellanox OFED for Linux User Manual*. Accessed: Oct. 7, 2022. [Online]. Available: http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v4_3.pdf

[7] M. Zhang and J. Huang, "Mechanism design for network utility maximization with private constraint information," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 919–927.

[8] M. Ashour, J. Wang, C. Lagoa, N. Aybat, and H. Che, "Non-concave network utility maximization: A distributed optimization approach," in *Proc. IEEE INFOCO Conf. Comput. Commun.*, May 2017, pp. 1–9.

[9] S. Ramakrishnan and V. Ramaiyan, "Completely uncoupled algorithms for network utility maximization," *IEEE/ACM Trans. Netw.*, vol. 27, no. 2, pp. 607–620, Apr. 2019.

[10] L. Vigneri, G. Paschos, and P. Mertikopoulos, "Large-scale network utility maximization: Countering exponential growth with exponentiated gradients," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 1630–1638.

[11] D. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE J. Sel. Areas Commun*, vol. 24, no. 8, pp. 1439–1451, Aug. 2006.

[12] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle, "Layering as optimization decomposition: A mathematical theory of network architectures," *Proc. IEEE*, vol. 95, no. 1, pp. 255–312, Jan. 2007.

[13] W.-C. Liao, M. Hong, H. Farmanbar, X. Li, Z.-Q. Luo, and H. Zhang, "Min flow rate maximization for software defined radio access networks," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 6, pp. 1282–1294, Jun. 2014.

[14] *TensorFlow-RDMA*. Accessed: Oct. 7, 2022. [Online]. Available: https://github.com/tensorflow

[15] *Spark-RDMA*. Accessed: Oct. 7, 2022. [Online]. Available: https://github.com/Mellanox/SparkRDMA

[16] J. Mo and J. Walrand, "Fair end-to-end window-based congestion control," *IEEE/ACM Trans. Netw.*, vol. 8, no. 5, pp. 556–567, Oct. 2000.

[17] C. Feng, H. Xu, and B. Li, "An alternating direction method approach to cloud traffic management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2145–2158, Aug. 2017.

[18] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, Jan. 2011.

[19] *Convex Optimization for Python*. Accessed: Oct. 7, 2022. [Online]. Available: https://www.cvxpy.org/

[20] X. Wang, H. Song, C.-T. Nguyen, D. Cheng, and T. Jin, "Maximizing the benefit of RDMA at end hosts," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2021, pp. 1–10.

[21] T. Li et al., "Revisiting acknowledgment mechanism for transport control: Modeling, analysis, and implementation," *IEEE/ACM Trans. Netw.*, vol. 29, no. 6, pp. 2678–2692, Dec. 2021.

[22] D. Firestone et al., "Azure accelerated networking: Smartnics in the public cloud," in *Proc. 15th USENIX Symp. Networked Syst. Design Implement.*, 2018, pp. 51–66.

[23] D. Kim, S. Lee, and K. Park, "A case for SmartNIC-accelerated private communication," in *Proc. 4th Asia–Pacific Workshop Netw.*, Aug. 2020, pp. 30–35.

[24] K. Wang, F. Dong, D. Shen, C. Zhang, J. Zhang, and J. Luo, "Towards tunable RDMA parameter selection at runtime for datacenter applications," in *Proc. IEEE 24th Int. Conf. Comput. Supported Cooperat. Work Design (CSCWD)*, May 2021, pp. 49–54.

[25] *Redis*. Accessed: Oct. 7, 2022. [Online]. Available: https://redis.io/
[26] A. Dragojevi, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Symp. Networked Syst. Design Implement.*, 2014, pp. 401–414.
[27] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement.*, 2016, pp. 185–201.
[28] D. Kim et al., "FreeFlow: Software-based virtual RDMA networking for containerized clouds," in *Proc. NSDI*, 2019, pp. 113–126.
[29] Y. Lu et al., "Multi-path transport for RDMA in datacenters," in *Proc. 15th USENIX Symp. Networked Syst. Design Implement.*, 2018, pp. 357–371.
[30] A. Singhvi et al., "1RMA: Re-envisioning remote memory access for multi-tenant datacenters," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 708–721.
[31] X. Wang et al., "StaR: Breaking the scalability limit for RDMA," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2021, pp. 1–11.
[32] J. Guo, F. Liu, J. C. S. Lui, and H. Jin, "Fair network bandwidth allocation in IaaS datacenters via a cooperative game approach," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 873–886, Apr. 2016.
[33] D. Shen, J. Luo, F. Dong, X. Guo, K. Wang, and J. C. S. Lui, "Distributed and optimal RDMA resource scheduling in shared data center networks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Jul. 2020, pp. 606–615.

Dian Shen received the bachelor's, master's, and Ph.D. degrees from Southeast University, China, in 2010, 2012, and 2018, respectively. He was a Visiting Researcher with The Chinese University of Hong Kong from 2021 to 2022. He is currently an Associate Professor with the School of Computer Science and Engineering, Southeast University. His research interests include cloud computing, virtualization, and data center networks.

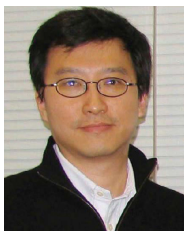
Junzhou Luo (Member, IEEE) received the B.S. degree in applied mathematics and the M.S. and Ph.D. degrees in computer network from Southeast University, China, in 1982, 1992, and 2000, respectively. He is currently a Full Professor with the School of Computer Science and Engineering, Southeast University. His research interests include network security and management, cloud computing, and wireless LAN. He is a member of ACM. He is the Co-Chair of the IEEE SMC Technical Committee on Computer Supported Cooperative Work in Design.

Fang Dong (Member, IEEE) received the B.S. and M.S. degrees in computer science from the Nanjing University of Science and Technology, China, in 2004 and 2006, respectively, and the Ph.D. degree in computer science from Southeast University, China, in 2011. He is currently a Professor with the School of Computer Science and Engineering, Southeast University. His current research interests include cloud computing, task scheduling, and big data processing.

Xiaolin Guo received the bachelor's degree from the Nanjing University of Science and Technology, China, in 2018. She is currently pursuing the Ph.D. degree in computer science with Southeast University, China. Her main research interests include edge computing and in-network computing.

Ciyuan Chen is currently pursuing the integrated master's and Ph.D. degree with the School of Computer Science and Engineering, Southeast University. Her research interests include data center networks, cloud (edge) intelligence, and optimization theory.

Kai Wang received the master's degree from Southeast University, China, in 2020. He has worked with Alibaba Cloud. He is currently working with NetEase Service Mesh Group. His main research interests include data center networks and cloud computing.



John C. S. Lui (Fellow, IEEE) received the Ph.D. degree in computer science from the University of California at Los Angeles. He is currently the Choh-Ming Li Chair Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK). His current research interests include quantum networks, machine learning, online learning (e.g., multi-armed bandit and reinforcement learning), network science, future internet architectures and protocols, network economics, network/system security, and large-scale storage systems. He is elected as a member of the IFIP WG 7.3, a Fellow of ACM, and a Senior Research Fellow of the Croucher Foundation. He has received various departmental teaching and research awards, including the CUHK Vice Chancellor's Exemplary Teaching Award and the Research Excellence Award from CUHK.