Accelerating Distributed Deep Learning using Multi-Path RDMA in Data Center Networks

Feng Tian
University of Minnesota Twin Cities
Minneapolis, Minnesota
tianx399@umn.edu

Yang Zhang
University of Minnesota Twin Cities
Minneapolis, Minnesota
zhan3248@umn.edu

Wei Ye
University of Minnesota Twin Cities
Minneapolis, Minnesota
ye000094@umn.edu

Cheng Jin
University of Minnesota Twin Cities
Minneapolis, Minnesota
jinxx339@umn.edu

Ziyan Wu
University of Minnesota Twin Cities
Minneapolis, Minnesota
wu000598@umn.edu

Zhi-Li Zhang
University of Minnesota Twin Cities
Minneapolis, Minnesota
zhzhang@cs.umn.edu

ABSTRACT

Data center networks (DCNs) have widely deployed RDMA to support data-intensive applications such as machine learning. While DCNs are designed with rich multi-path topology, current RDMA (hardware) technology does not support multi-path transport. In this paper we advance *Maestro* – a purely *software-based* multi-path RDMA solution – to effectively utilize the rich multi-path topology for load balancing and reliability. As a “middleware” operating at the user-space, Maestro is *modular* and *software-defined*: Maestro decouples path selection and load balancing mechanisms from hardware features, and allows DCN operators and applications to make flexible decisions by employing the best mechanisms as needed. As such, Maestro can be readily deployed using existing RDMA hardware (NICs) to support distributed deep learning (DDL) applications. Our experiments show that Maestro is capable of fully utilizing multiple paths with negligible CPU overheads, thereby enhancing the performance of DDL applications.

CCS CONCEPTS

• **Networks** → **Data center networks**; • **Hardware** → **Networking hardware**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

Multi-Path RDMA, Distributed Deep Learning, Data Center Networks

ACM Reference Format:
Feng Tian, Yang Zhang, Wei Ye, Cheng Jin, Ziyan Wu, and Zhi-Li Zhang. 2021. Accelerating Distributed Deep Learning using Multi-Path RDMA in Data Center Networks. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR ’21), October 11–12, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3482898.3483363

1 INTRODUCTION

Remote Direct Memory Access (RDMA) over Converged Ethernet [23] (*RoCE*) has been widely deployed in data center networks (DCNs) to support data-intensive applications such as distributed machine learning and deep learning applications [14]. RDMA/RoCE employs kernel bypassing and zero-copy to allow applications to directly read or write large chunks of data from the (*user space*) main memory of one server to that of another server. This is achieved by “offloading transport” to the dedicated RDMA network interface card (RNIC for short) which directly transfers data between the NIC hardware buffer and the (designated) application user space memory regions, thereby significantly reducing the kernel processing overhead and (remote) memory access latency. RoCE utilizes the standard stack (TCP/IP protocol) to establish the control path, including establishing the so-called Queue Pairs (QPs) on both ends of two communicating applications and registering the memory regions (see §2 for more details). For data path operations, applications use RDMA *Verbs* (e.g., `SEND`/`RECV`, `READ`/`WRITE`) for data transfer: the corresponding RNIC would automatically fetch data chunks to be transferred from the user space memory region, segment and encapsulate them into UDP/IP packets, and deliver them over the network. By design, RDMA is a *point-to-point* transport, where each RDMA connection (Queue Pair) is mapped onto a single network path (as specified by the IP address pair of the RNICs or traced by the hardware generated UDP flows).

Data-intensive RDMA workloads are often characterized by *periodic* transfers of large data chunks. For example, distributed deep learning applications, e.g., implemented using PyTorch [25] or TensorFlow [2] which has built-in RDMA support, require periodic parameter synchronizations among multiple instances running in a DCN, which often involve transfer 10 times or 100 times MBs or several GBs of data at a time. During such periods of time, the network link utilization increases significantly: the “elephant flows” associated with such data transfers not only compete for network bandwidth among themselves, creating congestion, but also severely affect the (tail) latency of “mice flows” associated with, e.g., the RDMA control messages, or other applications. On the other hand, modern DCNs are designed with a “spine-leaf” topology with rich multi-path diversity. The current *point-to-point* RDMA transport, unfortunately, cannot take advantage of such rich multi-path topology for intelligent load balancing to mitigate congestion (especially along the core paths which are oversubscribed) and reduce the overall flow completion time.

In this paper, we advance a purely *software-based* multi-path RDMA transport framework, dubbed *Maestro*. Our solution employs *three key innovations* by addressing the above challenges. First, we propose a novel *virtual* RNIC based user space path control mechanism. We create multiple virtual interfaces – each with a different (virtual) IP address of our choice – and bind them to the same physical RNIC (effectively creating multiple virtual RNICs). Second, we develop a user-space *middle-ware* layer that intercepts and split (large) messages of RDMA operations into multiple (smaller) messages, and dynamically maps them onto different paths at the sender side, and judiciously merge them together at the receiver side by passing them to the applications without introducing extra memory copy. Performing these operations correctly and incurring as little overheads as possible (especially, maintaining zero-copy) is nontrivial; they involve careful designs and some clever tricks (see § 3). Third, we decouple the path mapping (and selection), path monitoring, and load balancing mechanisms to allow flexibility and achieve application-aware load balancing.

Our framework is *software-defined* and *modular*: In contrast to the earlier hardware-based RDMA multi-path solution [20] and connection-based multi-path solution [40], both of which rely upon router ECMP support for multi-path load balancing by manipulating UDP source ports only when encapsulating RDMA data into UDP packets, our solution can not only take advantage of router ECMP for multi-path load balancing – *but with far more flexibility* through the assignment of virtual IP addresses to virtual RNICs, but also work with an SDN controller in a (software-defined) DCN to *explicitly* select and map (virtual) RDMA connections (virtual QP pairs) to specific paths. Furthermore, unlike a fixed congestion control algorithm that is built in the hardware NIC as in [20], our framework allows a DCN operator to implement different path monitoring and load balancing mechanisms with varying complexity. For example, we have implemented a user space path monitor and a load balancer running on end hosts that leverages the existing RDMA congestion control (DCQCN [44]) algorithm. However, instead of relying on the end-to-end congestion control and load balancing, our framework enables a DCN operator to utilize an SDN controller to perform intelligent load balancing by setting up appropriate flow rules and dynamically adapting them based on the (global) network conditions. *Maestro* can also control the number of virtual Queue Pairs that are set up for multi-path load balancing as well as select the granularity of load balancing by adjusting data chunk sizes. In other words, Maestro enables *application-aware* load balancing by allowing DCN operators to select appropriate paths and deploy the right algorithms in accordance with application requirements and metrics (e.g., bandwidth or latency) depending on the context. These capabilities are all made possible due to the modular design of *Maestro*.

Last but not the least, our purely software-based RDMA multi-path transport framework does not require any modifications to the existing RDMA NICs and thus can be readily deployed in today's DCNs. To evaluate the effectiveness of our framework in accelerating the performance of DDL applications in DCNs, we have incorporated *Maestro* in PyTorch [25] as a portable collective communication library. Our experiments demonstrate that Maestro can decrease up to 66.7% transport time in DDL by leveraging the rich multi-path DCN topology, when compared with the conventional (single-path) point-to-point RDMA, while incurring negligible CPU overheads.

2 MOTIVATION

2.1 RDMA and RoCE

RDMA allows applications to directly access remote memory with zero-copy and kernel bypassing. It offloads the transport logic in hardware RNICs. RDMA over Converged Ethernet v2 (*RoCE v2*) has been widely deployed in data center networks to support compute- & data-intensive applications, e.g., distributed deep learning, where RDMA packets are encapsulated with packets with UDP/IP headers. As shown in Fig. 1, RDMA is an end-to-end transport mechanism where control path and data path are decoupled. On control path, applications connect with each other using send and receive Queue Pairs (QPs). On data path, an application initiates RDMA operations (or *verbs*) by posting *Work Requests (`WRs`)* (or *Work Queue Element* (WQE)), e.g., `SEND/RECV`, `WRITE` or `READ`) to the QP using APIs from RDMA libraries (e.g., `ibv_post_send` or `ibv_post_recv`), which commands the RNIC to transfer data to the memory of a remote host. For each application, there is also one (or more) completion queue (CQ); upon completing a `WR`, a completion queue element (CQE) is delivered to CQ for notification purposes. RDMA is a message based transport, where RDMA messages are divided into segments and encapsulated in UDP packets (in *RoCE v2*) that are transported along a single path, and assigned with IP addresses and randomly selected UDP source port numbers by RNIC (using cached context on hardware)

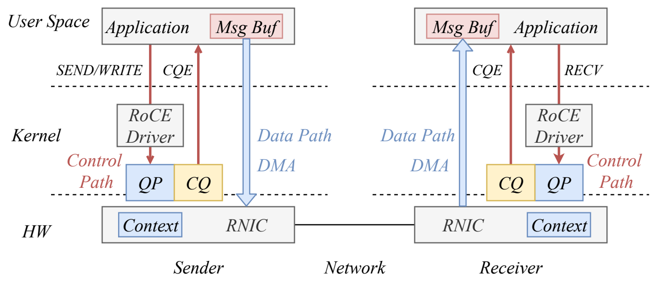


Figure 1: RDMA Transmission Mechanism

2.2 DCNs and Multi-Path RDMA

The "leaf-spine" topology in modern data center networks (DCNs) offers rich path diversity. Switches and routers often support "native" multi-path load balancing via *built-in* Equal-Cost Multi-path (ECMP) routing based on hashes of 5-tuples (⟨*src IP*, *dst IP*, *src port*, *dst port*, *protocol number*⟩) in packet/flow headers. The current *point-to-point* RDMA however cannot take advantage of ECMP. This is because once a QP connection is established, the five tuples used by UDP flows for the RDMA data path are fixed. Hence a large data transfer (encapsulated in a single UDP flow) can only be delivered via a single path. In addition, if the path fails, the current RDMA transport cannot automatically steer traffic to another path for continued transmission.

MP-RDMA [20] is the first to consider multi-path RDMA transport. It proposes a hardware-based solution with a "built-in" congestion control mechanism. The proposed solution is implemented and evaluated using FPGA-based emulation. The key challenge it focuses on is the limited memory in RNICs. By using the source port to encode "virtual path" id (*VP id*) and influence the path traversed by the RDMA UDP packets, it leverages but heavily relies on the underlying routers' ECMP mechanisms for load balancing among multiple paths. As MP-RDMA requires replacing existing RNICs with new MP-RDMA capable NICs, it cannot be readily deployed in DCNs. Moreover, the hardware-based solution also lacks flexibility at the application level e.g., an application has heterogeneous requirements on paths. Thus, a software-based multi-path solution is needed.

2.3 Multi-Path RDMA Benefits DDL

Distributed deep learning applications running in DCNs are often implemented using platforms such as PyTorch [25] and TensorFlow [2]. They use collective communication (e.g., *allreduce* & *allgather*) libraries, e.g., Horovod [32] and Gloo [8], for model (parameter) synchronization. These libraries have incorporated (point-to-point) RDMA transport for more efficient large data transfer. In the following, we will use an experiment we have conducted to illustrate that DDL applications based point-to-point RDMA transport do not fully utilize the network resource. In this experiment, we train the VGG19 [33] model (includes 143M floats, total 572Mbyte) using CIFAR10 [17] dataset (including 50K training images) by distributed PyTorch (Gloo is used for communication). Our testbed consists of 4 servers connected by 8 network paths using 40Gbps links with ECMP enabled. We measure the traffic load on each path using sFlow [24].

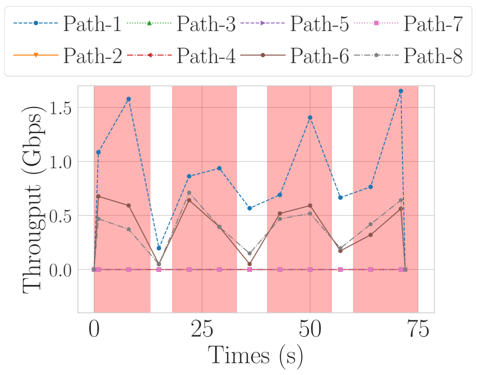


Figure 2: Traffic Load on Paths in DDL Training

As shown in Fig. 2, it has an obvious *On-Off* traffic pattern (e.g., the red shadows represent *On* transmission period where servers are exchanging data), which is caused by alternate computation & communication. Most importantly, the *On* period is wider than the *Off* period since the network transport dominates the time cost. We observe that the network bandwidth is not fully utilized. In particular, the loads along the paths are not well-balanced. For instance, only 3 of 8 paths are used for traffic delivery, where *Path-1* has about twice the traffic load than the other 2 paths. We further observe that DDL applications introduce period large data transfers ("elephant" DDL flows). For example, each epoch of this VGG19 training involves 782 iterations of data exchange with 858Mbyte per iteration when batch size is set as 64 and *ring_allreduce* algorithm is used.

Intuitively, we can accelerate distributed learning by using more light-loaded paths to transmit data faster which means compressing the *On* period. Moreover, there are amounts of DDL applications running in a single data center. Their flows can also compete with each other for the network resources (especially the bandwidth) if they are at the *On* periods at the same time. If all these applications can compress their *On* period, the possibility of transmission conflict (which may cause congestion in the network) can also be decreased.

3 MAESTRO DESIGN

In this section, we first lay out the design goals of Maestro and discuss the key challenges in designing a multi-path RDMA solution. We then provide an overview of Maestro and the key ideas behind its design.

3.1 Design Goals & Challenges

We seek a purely *software-based* multi-path RDMA solution operating in the user space for multiple reasons. First of all, without requiring modifications to existing RNICs that have been widely installed on servers in modern data centers, such a solution can be readily adopted and deployed to enhance the performance of data-intensive applications such as distributed deep learning (DDL). Second, we recognize that applications running in modern DCNs and their mixtures are often rapidly evolving, and service objectives and performance requirements can be highly diverse and dynamic. Hence instead of fixing a specific load balancing mechanism for all DCNs and all applications, we want a *software-defined* solution that provides DCN operators and applications with the flexibility in the multi-path routing and load balancing decisions, including deciding and specifying i) whether to utilize the multi-path RDMA transport; ii) if the multi-path RDMA transport is activated, what and how many paths should be used for load-balancing; and iii) what load balancing algorithms (including the information and metrics used for decision making) should be employed. For example, it has been shown that global congestion avoidance and traffic scheduling [11, 21, 27] are essential in DCNs, and applications are best aware of traffic load distribution for adaptive load balancing [16]. In particular, our solution should be able to take advantage of software-defined networking (SDN) capabilities for path selection, network monitoring, and load balancing. Third, the software-based solution cannot incur too many overheads so as not to reduce application performance and defeat the benefits of multi-path transport. Fourth, the solution should not require modifications to existing applications (or software platforms) that employ the current point-to-point RDMA transport.

Achieving all these goals simultaneously poses many challenges. For instance, RDMA "offloads" the transport layer functions to hardware RNIC, which restricts the packet-level operations we can perform. In particular, the UDP source port number in RDMA packets is assigned by RNIC randomly (between 49152 to 65535), while the destination port number is fixed as 4791. As a result, applications cannot control the UDP port number in RDMA packets. This forces the earlier solutions to either modify the RNIC hardware [20] or make an unrealistic assumption that applications can directly assign the UDP port number [40]. Both solutions also reply on router ECMP for multi-path load balancing. A key enabling idea of our proposed solution is to create multiple *virtual* NICs (vNICs) and binding them to the same hardware RNIC, thereby allowing multiple IP addresses (of our choice) to be assigned to the same RNIC. This not only enables us to circumvent the above challenge but also affords us the capability to explicitly specify or select paths for load-balancing. For example, by setting up an SDN flow rule in a core DCN switch with the appropriate (virtual) source IP address and (virtual) destination IP address corresponding to a pair of (virtual) RNICs (or virtual QP pair, see §3.2)) and the destination UDP port number = 4791, the corresponding RDMA packets (of the UDP flow) will then be forwarded along a selected path. Furthermore, we adopt a *modular* design by decoupling path selection and mapping, network monitoring, and load balancing mechanisms. As RDMA is message-based (where for DDL applications, each message can be as large as several GBs), we decompose the messages into smaller data chunks for load-balancing across multiple paths. To minimize the CPU overheads, we avoid any data copying; instead, we introduce several clever techniques and tricks to create multiple (virtual) QP pairs, segment and register appropriate memory regions for each QP pair, and so forth while at the same time maintaining the same RDMA verb semantics. Both the number of virtual QP pairs (thus the number of paths) and the chunk sizes for load-balancing can be specified by a DCN operation or an application, and dynamically modified if needed. We have implemented our solution as a "middleware" sitting on top of the existing RDMA library while providing the same standard APIs such as RDMA verbs to applications. As such, Maestro is completely transparent to applications.

3.2 System Overview

Employing a purely software-based solution, Maestro is designed as an *open, software-defined, and modular framework* to support multi-path RDMA transport at the user space. The overall architecture of Maestro is illustrated in Fig. 3.

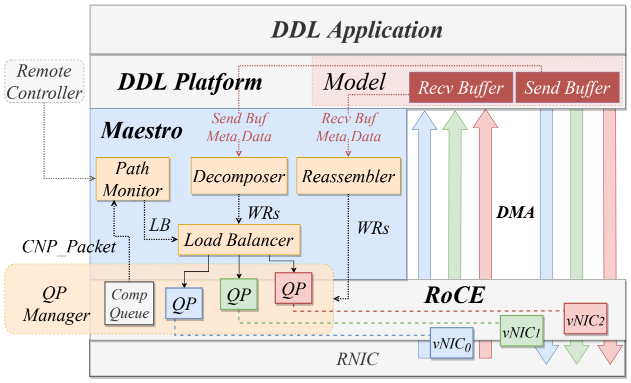


Figure 3: System Architecture of Maestro

As alluded to earlier, a key enabling innovation of Maestro is to create multiple virtual NICs (vNICs) using `ifconfig`, each assigned with a distinct IP address (of our choice), and bind them to the same hardware RNIC. RDMA uses a Global ID (GID) to identify each host, and RoCEv2 binds GIDs to IP addresses of the interfaces using the IP table. Using the vNICs, Maestro is able to create multiple Queue Pairs (QPs) using the standard RDMA library `rdma_cm`. Maestro maps each QP to a different (*virtual*) path (VP), using the IP address associated with each vNIC as a VP id. By using (virtual) IP addresses to identify (virtual) paths, Maestro is endowed with more flexibility in selecting and mapping paths for load balancing. It can not only more effectively utilize router ECMP for load balancing as the earlier solution; but perhaps more importantly, take advantage of SDN capabilities of modern DCNs to explicitly control path selection via policy routing. The modular design of Maestro enables us to employ different load balancing algorithms that better meet application performance requirements.

Maestro is built on top of the standard RDMA libraries, operating as a middleware running in the user space. In particular, it provides (and "replaces") the same RDMA APIs (and RDMA verbs) that applications use to invoke RDMA functions, thus is completely transparent to applications. For example, an application invokes `connect()` to establish an RDMA connection, which Maestro transparently sets up multiple virtual connections and QPs. The application uses `READ/SEND` and `WRITE/RECV` to post work requests, `WR`s. On the sender side, Maestro will automatically decompose a large RDMA message (thereafter simply a "flow") contained in a `WR` into smaller data chunks ("sub-flows"), and distribute them across different (virtual paths) QP's by generating the corresponding constituent `WR`'s for the sub-flows using the standard RDMA verbs. The sub-flows are "merged" at the receiver side. These are illustrated in Fig. 3. Maestro consists four major components, *QP Manager*, *Decomposer* (on the sender side), *Reassembler* (on the receiver side), and *Path Monitor & Load Balancer* (PM&LB), which are described in more details in §4.

Maestro assumes that there is at least (but not limited to) one single port of RDMA NIC (RNIC) connected to the access (ToR) switch as well as multiple paths in the core-layer of data center networks. The load balancing configuration on switches can be either ECMP (with known hash function), static routing, or policy routing using SDN. For (fine-grained) congestion control and flow control, Maestro can leverage DCQCN which is already implemented on modern RNICs.

4 MAESTRO COMPONENTS

In this section, we will present the core functionality of each component and describe the key mechanisms employed to enable Maestro to achieve our design goals.

4.1 User Space Path Control

As a purely software-based solution, path control is the most essential but challenging goal in RDMA. The key idea of Maestro is to create multiple Queue Pairs (QPs) to map multiple paths as *virtual* paths (*QP-Path Mapping*), then select the identical paths (*Disjoint Path Selection*). Maestro proposes a *virtual* NIC (vNIC)-based mechanism to assign distinct IP addresses to sub-flows so that they can be loaded on disjoint network paths to achieve parallel transmission If ECMP enables 5-tuple hashing, Maestro introduces a probing-based solution to help select as disjoint paths as possible *a posteriori*.

QP-Path Mapping. First, we discuss how Maestro realizes QP-based virtual path mapping. We recall that RDMA uses a global identifier (GID) to identify each host. *RoCE* binds GIDs to the IP addresses assigned to the RNIC interfaces and caches the context on RNIC hardware. During the data transmission (the *data path*), the packets are appended with the IP address by RNIC directly using the cached context without the kernel & CPU involvement. Meanwhile, the port numbers are randomly selected by RNIC and assigned to packets' header without the application's awareness. Therefore, it is hard to manipulate the values in 5-tuples (e.g., IP addresses or UDP port numbers) of RDMA flows in user-space and also maintain copy-avoidance and kernel bypass on the *data path* for path control purposes.

Innovatively, we propose a *Virtual NIC* (VNIC)-based QP-Path mapping method to control the IP address in sub-flows from user space. As illustrated in Fig. 4, it sets up multiple vNICs on each physical RNIC using the NIC virtualization technique, e.g., `IP alias`, each assigned with a distinct IP address. Then, Maestro creates QPs upon the vNICs, which are used as the *virtual* paths (VPs) in the future. These are all accomplished on the control path. On the data path, when applications transmit messages by posting `WR`s to QPs, the RNIC directly DMAs data from user space and assigns cached IP addresses to packets without any CPU/Kernel involvement. When the packets arrive at the switches, the IP addresses are used as the *VP id* of flows to perform multi-path transport. For instance, in SDN enabled network, forwarding rules can be added by the operator using IP addresses as identifiers of flows; in ECMP enabled legacy network, IP addresses can be used as a controllable parameter in 5-tuples to identify flows thereby they can be hashed to separate output paths. To this end, application in user space is capable of mapping the flows to physical paths as needed.

The advantage of this design is that Maestro not only relies on port numbers as the identical value for hashing in ECMP but can also implicitly identify paths by IP addresses [42]. For example, if one of the paths is broken or congested, Maestro can switch to another path by simply using another vNIC with a different IP address to initiate switches to steer the traffic to an alternative path, no matter which multi-path routing is configured. In addition, Maestro requires no modification on both hardware RNIC and software RDMA libraries, and can also work with multiple ports RNICs.

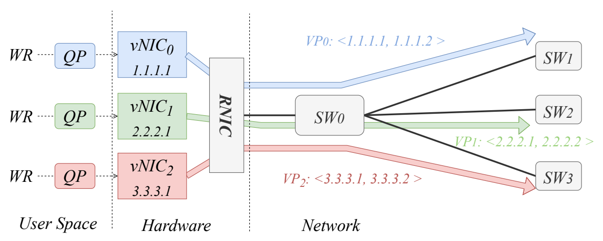


Figure 4: vNIC based QP-Path Mapping

Disjoint Path Selection. However, distinct QPs may be mapped to the identical physical path. For better network utilization, Maestro is supposed to select paths as disjoint as possible. A DCN operator can accomplish this by simply establishing a set of static routes to assist Maestro in mapping RDMA (UDP) flows to a set of (target) core paths for load balancing. If deployed in a "software-defined DCN", Maestro can instruct the SDN controller to install a set of load-balancing flow rules [39]. In the legacy network, where ECMP is enabled for load balancing, Maestro employs a hybrid path selection method by combining the probing and hash function calculation to filter out (virtual) paths with shared links. The idea is to utilize the (known) router ECMP hash algorithms (e.g., XOR algorithms on our testbed) to compute a set of hash results *a priori* (where the IP addresses are controllable) and filter out those that are mapped to the same (core) network paths. More specifically, the procedure works as follows: 1) If UDP port number is disabled in hash calculating, Maestro can simply use the IP addresses to calculate hash results and map the path (because IP addresses are the only variables in 5-tuples of RDMA traffic in this scenario). 2) Otherwise, if the UDP port number is enabled, we start by establishing several (virtual) QP connections, e.g., twice as large as the number of (target) paths in the core. 3) To check whether these QP connections are mapped to all paths in the core, we send a small probe message (e.g., 2Byte) along with each QP. Using a traffic sniffer tool, e.g., tcpdump [34][1], we can obtain the packet headers of each RDMA sub-flow identified by the source IP address. 4) After computing the hash results using 5-tuples in the headers, if the QP connections are mapped to all core paths, we filter those that are mapped to the same QP connection, leaving only one. Otherwise, we increase the number of QP connections (each with a different hash value calculated using the (known) router ECMP hash function) and repeat the process. The goal is to select numerous disjoint (virtual) paths that equal the number of (target) core paths. Note that this procedure is only required at the connection establishment phase. After that, the selected QPs are reused during DDL training.

QP Manager. Maestro implements the *QP Manager* component to handle the multiple QP connections, disconnections, and path selection. Besides, the *QP Manager* also creates a shared *Completion Queue* (CQ) to receive *Completion Queue Element* (CQE) for notification purposes. Moreover, this CQ is shared by QPs within the same Protection Domain (PD) where the registered memory region can be accessed by any member (especially QPs) to avoid hidden overhead caused by duplicated buffer registrations [10]. In this way, Maestro introduces the primary overhead on the control path rather than the computation-intensive data path that involves frequent completion queue queries and QP operations.

4.2 Multi-path Transport

Maestro decomposes a single message from the application into multiple "chunks" and performs parallel transmission using multiple paths on the sender side. Maestro proposes a "per-chunk" instead of conventional "per-packet" or "per-message" level transport and load balance design by considering both transmission efficiency (copy avoidance in RDMA) and flexibility (rapid response to network events).

Chunk-based Decomposition. RDMA uses *verbs* (`SEND/RECV`, `WRITE` or `READ`) to perform various transmission operations. In practice, application posts *Work Request*s (`WR`s) (to QPs) to command the RNIC for transmitting a specific message by providing the metadata (e.g., address and data size of corresponding memory buffer) of the message. The memory buffer has to be registered as a memory region to a PD in advance so that RNIC can directly

[1]libpcap library v1.9 or above is installed on and above support RDMA packet sniffer by default tcpdump from 4.9.2-4 and above

write and read data using Direct Memory Access (DMA) using pre-fetched authentication keys (`rkey` for remote access & `lkey` for local accesses).

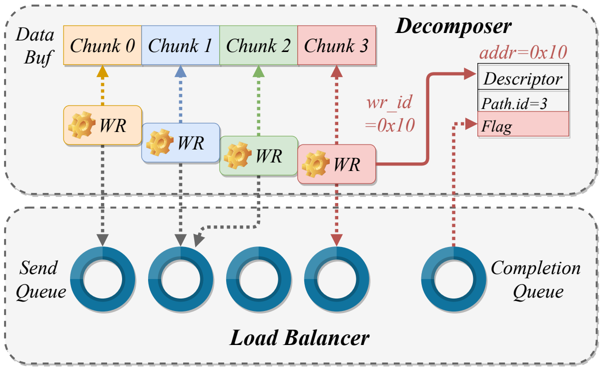


Figure 5: Chunk-based Multi-path Transport

As illustrated in Fig. 5, Maestro uses the `WR`s to decompose a message into *chunks* and perform the load balancing on *chunk* level. The *Decomposer* component is responsible for this task. The key idea is to partition the message into multiple blocks and each is represented by a `WR` with smaller and disjoint memory region. The message originally transmitted by a single `WR` is now using multiple `WR`s. These `WR`s are posted to multiple QPs in parallel. After RNIC processing `WR`s and transmitting the packets, a single message can be delivered via multiple paths simultaneously. From the application's point of view, though sole RNIC exists, multiple paths can be utilized for transmission. Maestro also performs load balancing on *chunk* level. For example, if one path is congested or broken during the transmission, applications can steer the traffic to alternative paths by posting involved `WR`s to other QPs. In addition, applications can tune the configurable *chunk* size to make a trade-off between the flexibility and efficiency to achieve the optimal performance regarding the customized setup. For an arbitrary message, 1) the smaller chunk size means more `WR`s. It requires Maestro to call *ibv_post_send()* (on multiple QPs) more frequently, which increases the CPU usage (as overhead). Besides, a large number of `WR`s also exacerbates the Out-Of-Order (*OOO*) issue (discussed in § 4.3) since the `CQE`s of the `WR`s may not be read together by a single *ibv_poll_cq()* call. Meanwhile, more alive `WR`s also consume extra RNIC resources (e.g., limited onboard memory) to get processed. 2) On the other hand, if the chunk size is too large, each `WR` has to deliver a large amount of data. Consequently, Maestro cannot be aware of or react to any networking event (congestion or failure) until the fairly large *chunk* accomplished transmission or the `WR` is timeout, which lacks flexibility and impacts the performance.

WR State Tracking. When multiple `WR`s are multiplexed in parallel, tracking the status of alive `WR`s can be CPU consuming in RDMA. To achieve efficient status tracking and less CPU involvement, Maestro introduces a novel structure, namely *WR Descriptor*. For each `WR`, a corresponding *WR Descriptor* is used to record multi-path transport required metadata as shown in table 1. The metadata is used to tracking the state of *chunk*s and notify the corresponding application on completion.

Maestro leverages the `wr_id` field to link the *WR Descriptor* to its `WR`. The `wr_id` is a 64-bits integer resident in `WR` structure that also residences in Completion Queue Element (CQE) when the transmission is accomplished. Maestro replaces the `wr_id` with the memory address (64-bits on most of the servers) of the *WR Descriptor*, which can fit into the `wr_id` field. And the original `wr_id` is stored in the *WR Descriptor* structure. Thus, both *WR Descriptor* and `WR` can be retrieved in $O(1)$ time.

Now we explain how exactly this mechanism works. For example, when the corresponding data transmission of a `WR` is accomplished, a CQE is generated by RNIC to notify applications where the `wr_id` is used for identification. Since we have already replaced the `wr_id` with the memory address of *WR Descriptor*, Maestro can directly retrieve this *WR Descriptor* without delay (compared with using storage structure such as a hash map). Then the metadata inside it can be used to check the status of the original message (i.e., whether all related `WR`s are completed or not). The benefit of this design is that first, we can bind multiple `WR`s to a single message without introducing much memory overhead; second, *WR Descriptor* is extendable as needed by applications.

Table 1: WR Descriptor Meta Data

Item	Description
WR ID	ID of the original WR
VP ID	ID of the virtual path posted to
Message ID	ID of the original message
Chunk ID	ID of the chunk

Multi-QP WR Posting. To deliver traffic via multiple paths in parallel, Maestro uses multiple worker threads to post `WR` to QPs simultaneously, one worker per QP. Maestro adopts the lock-less ring buffer design of `kfifo` in Linux kernel as the *Work Queue* between *Decomposer* and worker threads. *Decomposer* decomposes each message, generates multiple `WR`s, and then distributes them to distinct workers by inserting the corresponding *WR Descriptor* to the *Work Queue*. Then worker threads post the standard RDMA `WR` to its QP using standard RDMA API, i.e., `ibv_post_send`. Then RNIC takes over to perform kernel bypassed transmission.

We also remark that while Maestro performs the additional tasks of dividing a large message ("flow") contained in a `WRITE` or `SEND` `WR` into smaller messages ("sub-flows") by generating a sequence of `WR`s, these `WR`s are distributed across multiple QPs, and are performed using the standard RDMA verbs (`READ`, `WRITE` or `SEND`). In other words, the RNIC will directly read/write the corresponding data from or into the remote memory area of applications. As a result, Maestro incurs *no additional memory copying* on the data path, and is compatible with the existing RDMA system.

4.3 Multi-flow Reassembling

On the receiver side, traffic from distinct sub-flows is re-assembled to reconstruct the original message in the memory buffer, and also, applications are notified by the completion of transmission of the entire message. Particularly, the *OOO* issue in multi-path transport has to be carefully dealt with. Maestro introduces the *Reassembler* component to address above requirement.

Out-Of-Order: Due to various speeds of paths, sub-flows may arrive in arbitrary orders. Received data is supposed to be cached on the receiver side to avoid re-transmission. This is the so-called *OOO* issue. Maestro resolves the *OOO* issue by caching the received *chunk*s directly into the receiver's buffer in user space instead of the limited RNIC on-board memory. Maestro proposes a sender-oriented solution to proactively decide the destination memory address of each *chunk* by leveraging WRITE paradigm in RDMA. The key idea is to let the sender uses WRs to command the RNICs to directly put the data of each *chunk* into the right memory address in the receiver's buffer. Thus, on the receiver side, extra memory copies can be prevented when all *chunk*s are received, and eventually, the correct mapping of the original message can be reconstructed.

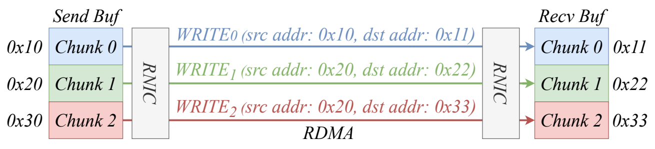

Send Buf
0x10
Chunk 0
0x20
Chunk 1
0x30
Chunk 2
RNIC
WRITE0 (src addr: 0x10, dst addr: 0x11)
WRITE1 (src addr: 0x20, dst addr: 0x22)
WRITE2 (src addr: 0x20, dst addr: 0x33)
RDMA
RNIC
Recv Buf
Chunk 0
0x11
Chunk 1
0x22
Chunk 2
0x33

Figure 6: *Reassembler*: Sender-oriented Reassembling.

To discuss the feasibility, we consider (one-sided) WRITE or READ and (two-sided) SEND/RECV verbs respectively: (1) In one-sided WRITE or READ verb case, RNIC uses pre-obtained rkey to access registered memory region on remote host. Besides, RDMA also exchanges the memory address of each other's receive buffer which can be used as its own virtual memory space. Thus, Maestro can directly apply a one-to-one mapping between local and remote memory buffer by including source and destination memory address in WRITE WR as illustrated in Fig. 6. Then RNIC performs standard RDMA WRITE on each chunk despite *OOO* issue.
(2) SEND/RECV is a two-sided verb that is usually for control message exchange where SEND and RECV work together. It typically transmits small messages which can be delivered by a single path. However, in rare situations, SEND/RECV may be used for relatively large messages. In this case, *Decomposer* translates the SEND WRs into WRITE; then uses WRITE verbs as described above to perform data transmission. Additionally, to consume the posted RECV on receiver side, an empty SEND or WRITE WR with imm_data (both require a SEND in receive queue) can be leveraged.

Nevertheless, it is also possible that the SEND verbs are supposed to be decomposed into SEND WRs as required by applications. This is also practicable as long as sufficient RECV WRs are posted to the receive queue before sending (which may also introduce more alive WRs). Other than that, imm_data is also needed for synchronizing purposes. However, the detailed design is out-of-scope of this work and will be addressed in the future.

4.4 Path Monitor & Load Balancer

To avoid congestion and achieve optimal parallel transmission, traffic loads on each path are supposed to be allocated according to its network condition, which is so-called congestion control & load balancing. Maestro, as an open platform, allows DCN operators to design and select various path monitoring and load balancing mechanisms as needed. Additionally, we provide the example *Path Monitor* and *Load Balancer* components for monitoring path congestion & failures and performing resilient routing & dynamic load balancing.

Congestion Control & Resilience: Maestro relies on the DCQCN [43] implemented on RNIC to achieve packet-level congestion control at user space. The DCQCN algorithm is the state-of-art mechanism of RDMA in the data center network which combines Data Center TCP (DCTCP) [3] and Quantized Congestion Notification (QCN) [1] algorithms and relies on Explicit Congestion Notification (ECN) marking on the switch. More specifically, 1) When congestion happens, switch (as Congestion Point, CP) marks the packet using ECN bits, which is propagated to the receiver RNIC. 2) Receiver RNIC (as Notification Point, NP) creates Congestion Notification Packet (CNP) and sends it to the sender. 3) When CNP is received in the sender RNIC (as Reaction Point, RP), the transmission rate of the QP is going to be throttled based on the DCQCN algorithm considering a pre-configured α value. Meanwhile, system-level counters are also updated by RNIC (e.g., number of received CNP packets (CNP_Packet_Handled)).

By leveraging the DCQCN counters, *Path Monitor* is aware of any congestion in previously used virtual paths. If any (indicated by increased CNP_Packet_Handled counter), *Path Monitor* marks the used paths as congested and notifies the *Load Balancer* to steer the traffic to other paths in the next transmission.

Path Monitor can also detect the path failure by monitoring CQE or probing (as defined) for failure recovery. A failed path can be congested or broken, which can be indicated by RDMA counters or errors in CQE. When path failure is detected, *Path Monitor* notifies *Load Balancer* to steer the traffic to other paths by selecting alternative QP to post WRs.

Dynamic Load Balancing: The *chunk*s decomposed from a single message are transmitted via multiple QPs by considering the network status of paths to realize dynamic load balancing in user space. To monitor network path status (e.g., bandwidth and latency), *Path Monitor* adopts a modular design so that various run-time subsystems can be employed. For instance, end-to-end probing-based mechanisms like qperf [18] can be directly used. Moreover, centralized SDN-based zero-queue network mechanisms such as [27] can also be integrated.

The *Load Balancer* that residents in user space (as in Fig. 5) can leverage comprehensive information to apply dynamic load balancing. For example, an application can directly use end-to-end path statistics (e.g., end-to-end bandwidth) to distribute different numbers of *chunk*s from a single message. Moreover, in zero-queue network design, the *Load Balancer* can work as the local agent to acquire the controller scheduling command for any transmission task.

Furthermore, applications can command the *Load Balancer* to transmit various types of messages based on their requirements. For example, for big "elephant" flows that transmit large messages, more paths can be used to aggregate more bandwidth; for small "mice" flows such as control messages, the path with minimum latency is usually selected. In the first case, Maestro can adopt the "flow-let" mechanism to apply load balancing between heterogeneously loaded virtual paths. The reason why we can use this design is as follow: 1) In RDMA, transport is offloaded to hardware RNIC. Packet-level load balancing cannot be achieved in user space since it tremendously delays the data path. 2) RDMA is a message-based transport mechanism where packets are transmitted in burst where "flow-let" is feasible [37].

5 IMPLEMENTATION & SETUP

In this section, we present integrating the Maestro with DDL platforms to support RDMA-enabled DDL applications in user space and describe the experimental setup of our testbed.

5.1 Integrating Maestro with DDL Platforms

We have implemented a prototype of Maestro as a user-space "middleware" library on top of the standard RDMA libraries, `librdmacm` and `libibverbs`, and have been working on integrating it with PyTorch [25] to support distributed learning applications. Maestro is implemented in C language, and can be ported by Python-based DDL platforms as a stand-alone library (e.g., via `ctypes` library [5]). Distributed deep learning consists of process group initialization, model computation, and model synchronization phases. The communications are involved in both initialization and model synchronization stages. In the rest of this section, we introduce the implementation details of Maestro from these two aspects respectively.

Connection Establishment in Initialization: First, we introduce how Maestro establishes multiple QPs among multiple processes. Initially, in collective communication-based distributed deep learning, multiple processes connect to form a communication group where `Rand ID` is used as the identification of each process. For instance, in PyTorch, processes call the the `init_process_group` API to initiate the communication group as a full-mesh topology where `Rand ID` and `World Size` are passed as parameters. We implement Maestro version of `init_process_group` API in Maestro library so that applications can simply call it using the same parameters. Within the Maestro's `init_process_group` API, the process with (`Rand ID`=0) is selected as the server process to listen to the connection requests from other processes as in original `init_process_group` function to form the full-mesh topology. After that, the *QP manager* exchanges the IP address information using the default connection between each pair of processes to establish extra QP connections.

Collective Communication in Model Synchronization: The second phase in data-parallel distributed deep learning is model gradient computation and model synchronization, where each node uses part of the dataset to compute local model copy and exchange the model parameters for synchronization. During the synchronization stage, DDL platforms adopt the collective communication algorithm (e.g., state-of-art *ring_allreduce*) to exchange the model group-wide. For instance, in PyTorch, processes first compute the model locally using allocated data batch. After each iteration of computation on batched data, the `all_reduce` API is called to synchronize parameter values of each layer of the neural network model (stored as Tensors) among nodes within the connected communication group. In PyTorch or TensorFlow, Tensors are used to store the models in user space memory. The memory space of the tensors can be accessed by querying the corresponding `storage` object. Then, these memory buffers can be used in Maestro as messages.

Maestro replaced the `all_reduce` API by implementing a "build-in" *ring_allreduce* algorithm (which can be easily extended to other collective algorithms, e.g., *allgather* or *broadcast*) to perform collective communication. Applications call Maestro's API to perform model synchronization using the multi-path transmission in each iteration as the original API. However, since memory buffers are required to be registered before use, Maestro introduces an extra phase in the first iteration to query the memory address of each tensor and register them with the protection domain. However, this can be optimized for transparency. In future iterations, when `all_reduce` API is called (tensor is passed as the parameter), Maestro directly uses the registered memory region handler to access the buffer and performs the multi-path transmission. Additionally, Maestro uses the qperf [18] as the run-time *Path Monitor*.

Maestro also introduces the following optimizations in implementation. (1) Since the tensor buffer of the model is reused in PyTorch, *Decomposer* also reuses the `WR`s for each chunk to avoid dynamic memory allocation. (2) Maestro is implemented using an event-based mechanism to handle multiple QPs management and transport to avoid introducing extra CPU overhead. (3) Moreover, Maestro uses the model buffer as the sender buffer to apply in-place collective computation to avoid memory copy.

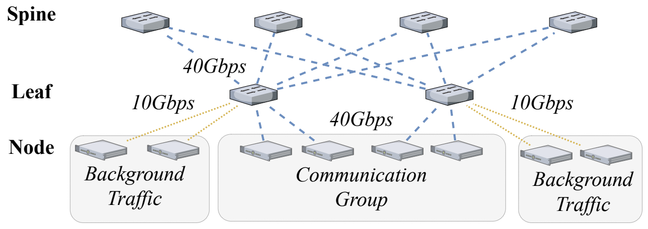


Figure 7: Test Bed Setup

Table 2: Testbed Hardware Specifications

Component	Specification
CPU	24x Xeon(R) E5-2620v3 @ 2.40GHz
DRAM	10x 16GB DDR4 @ 2133 MHz
RNIC	4x Mellanox ConnectX-5, 50Gbps
RNIC	4x Mellanox ConnectX-3 Pro, 10Gbps
Switches	Dell Z9100-ON (Dell OS 10)

5.2 Experimental Testbed Setup

As shown in Fig. 7, we use 8 servers connected by 6 switches to form a 2-tier spine-leaf topology where 4 servers serve as a communication group and the other 4 servers are background traffic generators. All the links are set as 40Gbps as default, except synthetic congestion configurations in specific experiments. As shown in Table 2, RNICs are Mellanox ConnectX-5 on computation nodes and Mellanox ConnectX-3 Pro on traffic generators. The software drivers used are *MLNX_OFED_LINUX-5.3-1.0.0.1* (*MLNX_OFED_LINUX-4.9-3.1.5.0* for ConnextX-3 Pro). The operating system on severs are all Ubuntu 18.04 with 4.15.0-142-generic kernel. On both ToR switches, ECMP with XOR algorithms is enabled to perform multi-path routing using seed value ($\langle srcIP,\ dstIP,\ srcPort,\ dstPort,\ Protocol\rangle$) configuration. The MTU in the entire network is 4096Bytes. RoCE traffic requires a lossless network, so we enabled WRED with $(Min, Max, Drop\ Rate, ECN) = (50KB, 100KB, 1.0, enabled)$ and PFC, to configure it as lossless converged Ethernet. α value in the DCQCN is set as the default value, 1023. RoCE traffic goes through a high-priority queue while TCP traffic goes via the default-priority queue.

6 EVALUATION

We conduct a comprehensive evaluation of Maestro in performance of communications in DDL from multiple perspectives as presented below.

6.1 Multi-path Utilization in DDL

In this section, we evaluate the performance of Maestro in utilizing multiple paths in collective communications of DDL. We start by evaluating the end-to-end throughput performance of Maestro using RDMA Perftest [26]. We also investigated how the chunk size impacts transmission efficiency and flexibility. Furthermore, we demonstrated that Maestro improves the performance of collective communications.

End-to-end Throughput. Since end-to-end communication is the most primitive communication in DDL data transmission, we first evaluated the throughput of Maestro to show that Maestro can effectively utilize multiple paths between spine and leaf (i.e., core-level) by bandwidth aggregation.

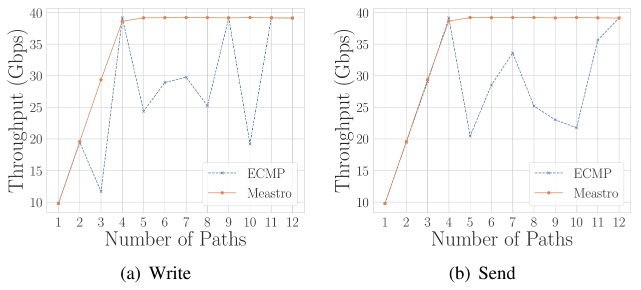


Figure 8: End-to-End Bandwidth Performance

To compare with multi-thread single-path RDMA with ECMP, We run the benchmark tools of the Perftest [26] toolkit with Maestro (*ib_send_bw* and *ib_write_bw*) to measure end-to-end throughput for `SEND`/`WRITE` verbs. For each experiment, we ran 112089 iterations with a message size of 65536 Byte. To generate synthetic congestion, we set the link speed in core as 10Gbps while access links (the links between RNIC and ToR switches) remain 40Gbps. Thus, the core-level network can be the bottleneck.

As demonstrated in Fig. 8, Maestro outperforms ECMP multi-path routing in both `SEND`/`WRITE` verbs cases. As shown in Fig. 8(a), the total throughput increase linearly as the number of used paths increasing until the core-level aggregated bandwidth reaches the maximum bandwidth of the access link (40Gbps). When throughput reaches the maximum throughput of RNIC, Maestro stabilizes the performance while ECMP does not. This fact guarantees that Maestro achieves an effective utilization of core-level bandwidth while there are more available paths. On the contrary, the ECMP solution suffers from hashing conflicts. This result holds the same in `SEND` verb scenario as shown in 8(b).

Chunk Size. Maestro decomposes a large message into multiple chunks which are posted to multiple QPs simultaneously. As a result, the chunk size is the key factor that determines the trade-off between granularity of load balancing and transport efficiency. For instance, given a message, a smaller chunk size increase the number of `WR`s used which increases the overhead (on both CPU & RNIC as discussed in §4.2) but achieve a finer grain of dynamic load balancing. In this experiment, we select the proper chunk size on our testbed and evaluate its impact on transport efficiency.

For the setup, we synchronized the VGG19 model with 1600Kbyte message size (the maximum size of a single message), by running *ring_allreduce* algorithm among 4 servers. The chunk sizes are set as 12.5, 25, 50, 100, 200, & 400K bytes in experiments. We used the standard *libibverbs* APIs, i.e., *ibv_query_rt_values_ex()*, to query raw (Host Channel Adapter) HCA clock cycles for communication time acquisition.

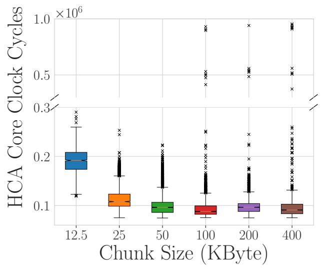


Figure 9: Chunk Size

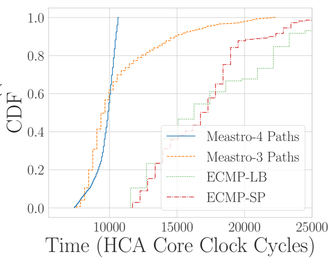


Figure 10: Transport Time

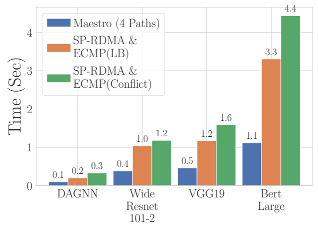
We calculated the completion time of each block to evaluate the impact of the chunk size on transport efficiency. Fig. 9 shows the statistical distribution of the completion time. With chunk size increasing, the median completion time decreases. As the network is configured in lossless mode, even if there is congestion, traffic can always be delivered without packet loss but with a larger flow completion time, which is represented as outliers. Thus, when a larger chunk size is used, Maestro is weakened in flexibility that manifests as outliers increase. The reason is that larger chunk size delays the response of Maestro towards congestion, consequently, more congestion happened. Thus, the number of outliers increased with the chunk size increasing. In conclusion, to balance the transport efficiency and granularity of dynamic load balancing, 50KB is the optimal chunk size in our setup, which also meets the results as in [10, 19]. Thus, we selected 50K bytes as the default value of chunk size in the following experiments.

Collective Communication Performance. Collective communication is critical for cluster-based distributed applications since communication is the bottleneck. In this experiment, we evaluate how Maestro performs in real DDL communication scenarios. We run the state-of-art collective communication algorithm, *ring_allreduce* [12], to show that Maestro can improve the collective communication performance by utilizing multiple paths in model synchronization.

To simulate the synthetic congestion in DCN, we set the speed of core-level links as 25Gbps while the links between RNIC and ToR switches remain 40Gbps. We test multiple models (DAGNN [35], Wide_Resnet101_2 [41], VGG19 [33] and Bert_large [6]) to comprehensively evaluate the performance of Maestro. The message size selected to apply *ring_allreduce* is set as 1600Kbyte. The chunk size used in this experiment is 50Kbyte. For communication time cost, we querying the HCA raw clock on RNIC on each `CQE` for accurate hardware time acquisition. Meanwhile, the standard system API, i.e., *clock()*, is used for the application running time acquisition. To measure the network congestion, we leverage the statistic counters, particularly ECN marks, on switches. Note that we do not consider the computation cost on CPU for the following two reasons. 1) CPU computation is constant and not the bottleneck in collective communication. 2) For large message cases, the GPU is usually used for computation.

a) Message Transmission: We evaluate Maestro's performance in the transmission of messages. As shown in Fig. 10, we use the VGG19 model as an example to illustrate that Maestro can decrease the transmission time in the transmission of messages (data blocks). The red dot line is the result when the application uses a single path RDMA and encountered hash conflicts, where all flows are hashed on the same path. On the contrary, the green line is the ideal case of ECMP where all four flows are hashed on disjoint paths. An essential observation is that although load-balanced ECMP slightly outperformed conflicted ECMP, they have similar overall performance. The main reason is that RNIC still cannot reach its full performance due to the limited bandwidth on congested core-level paths. Thus, when the core-level path is the bottleneck, ECMP cannot resolve the congestion issue effectively.

On the other hand, Maestro can resolve this issue by aggregating available bandwidth on multi-paths. As illustrated in the figure, Maestro outperformed ECMP by decreasing up to 60% of the completion time in transmission. Maestro has the minimum completion time (approximately 5000 to 10000 HCA clocks) when ECMP has a maximum of 25000 HCA clocks. A corner case is a 3-paths situation, where only 3 paths with accumulated 150Gbps are used. In a lossless network, 4 nodes cluster requires a totally 160Gbps to avoid congestion. Consequently, Maestro encountered congestion but still outperforms ECMP.


(a) Transport Time

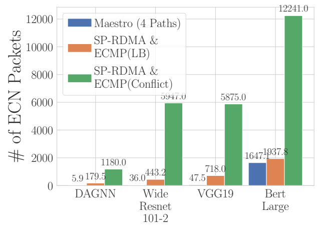
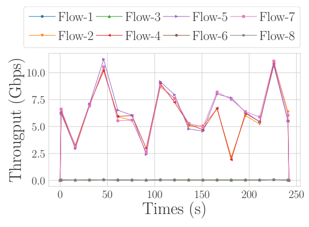

(b) Congestion Avoidance

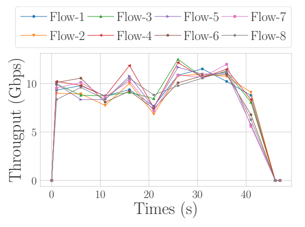
Figure 11: Collective Communication in DDL

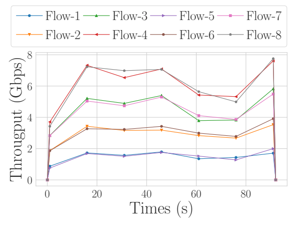
b) Collective Communication: Then we run the same experiment to evaluate the overall performance of Maestro in DDL collective communication. i) *DDL Communication:* As shown in Fig. 11, Maestro outperforms ECMP at least 2 times. Meanwhile, Maestro performs better in larger model cases since communication dominates the costs. ii) *Congestion Avoidance:* As shown in Fig. 11(b), Maestro also achieves congestion avoidance when the core-level path is the bottleneck. Compared with the ECMP, Maestro could balance traffic better among all paths. Since Maestro can effectively aggregate available bandwidth of every single path, fewer congestion events (ECN marked packets) and *DCQCN rate decrease* are triggered. In conclusion, when the core-level network is the bottleneck, Maestro can fully utilize multiple paths to decrease transport time (up to 66.7%) in DDL, and also effectively avoid congestion in the network.

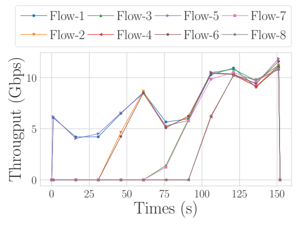
6.2 Dynamic Load Balancing

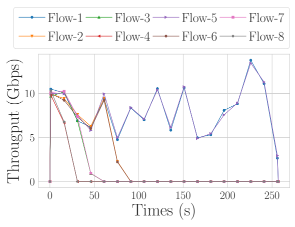
In this subsection, we evaluated the performance of Maestro in terms of "user-space" dynamic load balance and resiliency. We run the *ring_allreduce* algorithm within 4-node communication groups. We use the VGG19 model as an example to illustrate the results. For traffic load measurement upon paths, we use the sFlow [24] tool with 1 out of the 4096 sample rate on switches.


(a) Single-Path RDMA & ECMP


(b) Path Utilization


(c) User Space Load Balance


(d) Active Path Detection


(e) Failure Detection & Recovery

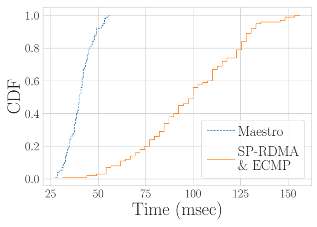

(f) Latency Aware Path Selection

Figure 12: Dynamic Application-aware Load Balancing

Load Balance. First, we disabled Maestro and rely on ECMP for load balancing. Then, we run Maestro to run the same application without any background traffic. Last, we load background traffics on 3 of 4 paths, which are 10Gbps, 20Gbps, and 30Gbps, using other 4-pair of servers respectively. We measure the sub-flow traffic on each path from the application to show that Maestro can detect the network bandwidth and load the traffic accordingly. Fig. 12(a) and Fig. 12(b) show that Maestro can initiate 8 flows which can be loaded on 4 paths in two directions. ECMP only initiated 4 flows and loaded on 2 paths, so the other 2 paths are left unused. Moreover, Maestro performed better because less congestion was observed, and thus DCQCN initiated less *transmit rate decrease*. Fig. 12(c) shows that Maestro can allocate traffic among paths based on the network conditions. The distribution of sub-flows adapts to the available bandwidth measured by the *Path Monitor*, which means Maestro loads traffic on paths according to the network condition. For example, the traffic load on the path with 20Gbps available bandwidth has 2 times the traffic load of a path with 10Gbps available bandwidth.

Path Detection. In this experiment, we show that Maestro can detect and utilize new paths. Initially, we enable only two physical paths. Gradually, we enable a new path per 30 seconds, to show whether Maestro can detect and utilize it. As illustrated in Fig. 12(d), new flows are joining periodically, which means Maestro detects the new paths and steers traffic to them. In particular, in every 30 seconds, a new flow is initiated (e.g., at the 30s, 60s, and 90s), which means Maestro detects and utilizes the new path rapidly.

Resiliency. In this experiment, we show that Maestro can failover traffic to healthy paths (*resiliency*). Initially, we enable all 4 physical paths (8 logical paths) and Maestro synchronizes data between nodes. Then, we gradually disable paths to simulate path failure, as the results illustrated in Fig. 12(e). Another important feature is that Maestro dynamically modifies the load of paths by allocating chunks (`WRs`) to QPs proportionally. Thus, after detecting the failure, Maestro immediately load traffic among paths in balance. Moreover, this whole procedure is done during *off* traffic intervals (computation period), so that RDMA transmission is not degraded.

Adaptive Path Selection. DDL, as a typical distributed system, synchronizations among nodes play an important role. For instance, some Message Passing Interface (MPI)-based mechanisms use *MPI_barrier* operation to synchronize nodes inside the communication group. It exchanges small messages (e.g., 2Byte) and is latency-sensitive. In this experiment, we show that Maestro can adaptively select a low-latency path for synchronization operations.

We run a token-based *MPI_barrier* algorithm among 4 nodes where a 2Byte size token message is passed along the ring topology to synchronize processes. To simulate the latency on paths, we set the latency on 4 paths with 10, 20, 30 & 40ms values (3ms jitters with normal distribution) in each direction using *tc* command. Then we measure the completion time of the synchronization operation on the first node (the rank with id 0 who initiates the token passing). We compare Maestro with ECMP based path selection. Fig. 12(f) shows that Maestro can always detect the path with the lowest latency to send out the message on each node. Compared with ECMP, Maestro achieves a maximum 2 times lower synchronization time, so that DDL applications can avoid blocking caused by high-latency paths.

6.3 Overhead & Scalability

In this section, we evaluate the overhead of Maestro on both host (CPU overhead) and RNIC hardware along data path and control path respectively, to show that Maestro introduces most overhead on the control path but negligible overheads on the data path, thus it is feasible for large-scale deployment in DCN.

Data Path Overhead: First, we evaluate the CPU usage introduced by Maestro during data transmitting which is the data path of Maestro. We ran *ring_allreduce* algorithm on the 4-server clusters to synchronize the VGG19 model that is the size of 245Mbyte. For CPU usage acquisition, we use CPU cycles via system API, *clock()*. As shown in Fig. 13, with more paths are using, the CPU usage is slightly increasing. The main reason is that we initiated a worker thread for each QP which is the primary CPU hot-spot in Maestro. For each QP worker thread, it introduces about 6% CPU overhead compared with single path RDMA. However, this overhead is acceptable in DDL scenarios because of the following three reasons: 1) Compared with the benefits gain introduced by Maestro (up to 2 times by using one extra path), the CPU overhead can be negligible. 2) It is noteworthy that in GPU training, the CPU is idle. Thus it is worth making this trade-off for communication benefit. 3) Most importantly, though the core-level path is the bottleneck, it only requires a limited number of paths (4 at maximum on our testbed) to fully utilize the RNIC. Therefore, the CPU overhead will not increase infinitely. Nevertheless, this is also a potential direction for optimization in future implementation. For instance, we could adopt batch-based posting mechanisms or event-based polling mechanisms to avoid CPU usage.

In terms of RNIC hardware, the lower boundary of the number of alive `WRs` is decided by the number of used paths, while the higher boundary can be configured by the application. In the end-to-end communication, at least 4 alive `WRs` exist on our testbed since 4 paths are used. Meanwhile, we also decrease the RNIC overhead by limiting the maximum alive number of `WRs` to 8 which is sufficient for pipeline optimization. Besides, 4 paths are sufficient to fully utilize the RNIC in our experiments. Thus, in the end-to-end communication, 4 alive `QPs` is sufficient to utilize these paths. Furthermore, in widely used ring-based collective communication, 8 `QPs` at maximum on each rank are sufficient where each rank only has to communicate with its left and right neighbors, which introduces negligible overhead on the RNIC (since an RNIC can support about hundreds of QPs before performance drop on throughput [15]).

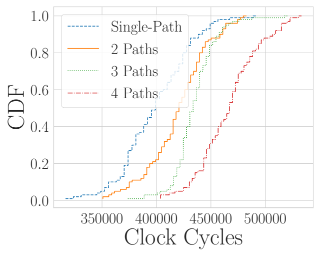


Figure 13: CPU Overhead

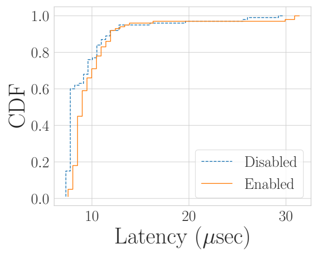


Figure 14: Aggressiveness

Control Path Overhead. The primary overhead introduced by Maestro is during the connection establishment phase (control path) where Maestro connects each pair of nodes using multiple QPs. To break down the overhead, we collect the number of the API calls of Maestro during this phase.

During connection establishment, for each pair of nodes, one works as a client while the other works as the server. On server-side, Maestro is listening to multiple connecting requests; on client-side, Maestro generates multiple QP connecting requests. As shown in Table 3 (N is the number of QPs used), on the client-side, the number of API calls increases linearly with the number of used paths increasing. On server side, Maestro calls *rdma_listen()* APIs once to accept multiple connection requests, and *rdma_accept()* is called multiple times to process the connection request. These operations introduce primary CPU usage and latency on the control path, however, are not to DDL communication. Because QPs are connected in process group initialization stages and reused during training.

Impact on Mice Flows. In this experiment, we validated whether Maestro is non-aggressive to other traffic when initiates traffic on more paths. In a PFC-enabled lossless network, latency-sensitive (especially tail-latency) mice flow (e.g., SDN control messages) usually are set to high priorities. Thereby, RDMA traffic (normally uses priority 3) may not influence such traffics. However, flows

Table 3: Control Path Overhead Breakdown

Item	API	# of calls
common	*ibv_create_cq*	1
	rdma_create_qp	*N*
Client	*rdma_resolve_addr*	*N*
	rdma_resolve_route	*N*
	rdma_connect	*N*
Server	*rdma_listen*	1
	rdma_accept	*N*

that share the same priority queue with RDMA traffic on the same path can be affected by extra sub-flows from Maestro. To explore such influences on mice traffic flows, we ran *ring_allreduce* with Maestro to synchronizing the VGG19 model between four servers. Then, we used *ib_send_lat* to send 2Byte messages between the other two servers to simulate a mice RDMA control traffic that also uses priority 3.Then we measure the maximum flow completion time of mice flows as the tail-latency. As shown in Fig. 14, the latency introduced by sub-flows form Maestro is negligible which means that Maestro is not aggressive to these mice flows. We can state that Maestro utilizes available bandwidth on extra paths but does not affect the mice-flows on these paths.

7 RELATED WORK

RDMA plays an important role in data center [22] to boost DDL. Existing multi-path RDMA transportation heavily relies on ECMP hashing, but it could hardly make DDL applications saturate link bandwidth [4]. MP-RDMA [20] is the first to address the challenge that RDMA cannot effectively take advantage of rich multiple paths in DCNs [14, 28, 31]. It proposes a hardware-based solution with "built-in" path selection and congestion avoidance mechanisms. The key challenge it focuses on is the limited on-board memory (see also FaRM [7], LITE [36] and INFINISWAP [13] that tackle similar constraints). Unfortunately, hardware-based solution cannot support requirements-oriented path selection [38]. In contrast, kernel-space solutions [9, 30] can hardly interact with RDMA where kernel is bypassed on data path. Similar to our work, Wang *et al* propose a software-based load balancing solution [40] using data partition. However, their solution *assumes* that user space applications can readily choose the port number of UDP packets, which is infeasible using current RoCEv2 design. Avatar [29] is another middleware solution in RDMA that allows multiple applications to share a single RDMA transport by multiplexing WR messages from multiple applications via consistent QPs. Avatar can eliminate lock contention and provide fair data scheduling. However, it does not tackle the same problem as our work, namely, RDMA application load balancing over multiple core DCN paths (with possibly a single host RNIC).

8 CONCLUSION & FUTURE WORK

This paper presents *Maestro*, a purely *software-based modular multi-path* RDMA solution that brings efficiency and flexibility. A novel vNICs based user space path control mechanism and a *middle-ware* transport layer is proposed to achieve effective multi-path transmission in RDMA without introducing extra memory copy. The user space path monitor and load balance are also realized to provide application-aware path selection and resilience for the heterogeneous requirement of DDL applications. Our experiments show that Maestro can effectively utilize multiple paths by aggregating bandwidth and detecting path failure in collective communication. Meanwhile, Maestro introduces negligible CPU overhead in data paths which is feasible for large-scale DDL platforms.

Maestro is presented to bring inspiration for the community to leverage the flexibility of software-defined techniques. We plan to i) extend the idea to state-of-art platforms (Pytorch [25]) to support real-world application by integrating Maestro to large-scale platforms such as Gloo [8] to support both CPU & GPU training. ii) release Maestro as an open-source project to contribute to more RDMA enabled DCNs.

ACKNOWLEDGMENTS

The research was supported in part by NSF under Grants CNS-1618339, CNS-1814322, CNS-1831140, CNS-1836772, CNS-1901103, CNS-2106771 and CCF-2123987.

REFERENCES

[1] IEEE 802.11Qau. 2010. Amendment: 10: Congestion Notification. In *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks*. IEEE. https://1.ieee802.org/dcb/802-1qau/

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1851182.1851192

[4] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. 2013. Per-Packet Load-Balanced, Low-Latency Routing for Clos-Based Data Center Networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (Santa Barbara, California, USA) *(CoNEXT '13)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2535372.2535375

[5] The ctypes Group. [n.d.]. ctypes. https://docs.python.org/3/library/ctypes.html

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[8] Facebok. [n.d.]. Gloo. https://github.com/facebookincubator/gloo.

[9] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824. https://doi.org/10.17487/RFC6824

[10] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 553–560.

[11] Monia Ghobadi, Soheil Hassas Yeganeh, and Yashar Ganjali. 2012. Rethinking End-to-End Congestion Control in Software-Defined Networks. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (Redmond, Washington) *(HotNets-XI)*. Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/2390231.2390242

[12] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

[13] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In

14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[14] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 202–215. https://doi.org/10.1145/2934872.2934908

[15] Anuj Kalia et al. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *USENIX OSDI* (Savannah, GA). 185–201.

[16] Hari Kathi et al. 2006. Data traffic load balancing based on application layer messages. US Patent App. 11/031,184.

[17] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[18] Linux. [n.d.]. qperf. https://linux.die.net/man/1/qperf.

[19] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2019. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. *ACM Transactions on Database Systems (TODS)* 44, 4 (2019), 1–45.

[20] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 357–371. https://www.usenix.org/conference/nsdi18/presentation/lu

[21] Yifei Lu and Shuhong Zhu. 2015. SDN-based TCP congestion control in data center networks. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*. IEEE, Nanjing, China, 1–7. https://doi.org/10.1109/PCCC.2015.7410275

[22] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. *ACM Comput. Surv.* 53, 1, Article 3 (Feb. 2020), 37 pages. https://doi.org/10.1145/3363554

[23] Nvidia. [n.d.]. RoCE. https://docs.mellanox.com/pages/viewpage.action?pageId=12013422

[24] Sonia Panchen, Neil McKee, and Peter Phaal. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176. https://doi.org/10.17487/RFC3176

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., Vancouver, BC, Canada, 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[26] The perftest team. [n.d.]. Perftest. https://github.com/linux-rdma/perftest

[27] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 307–318. https://doi.org/10.1145/2740070.2626309

[28] Jim Pinkerton. 2002. The case for RDMA. *RDMA Consortium, May* 29 (2002), 27.

[29] Haonan Qiu, Xiaoliang Wang, Tianchen Jin, Zhuzhong Qian, Baoliu Ye, Bin Tang, Wenzhong Li, and Sanglu Lu. 2018. Toward Effective and Fair RDMA Resource Sharing. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking* (Beijing, China) *(APNet '18)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3232565.3232636

[30] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/2018436.2018467

[31] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robertazzi. 2013. Design and Performance Evaluation of NUMA-Aware RDMA-Based End-to-End Data Transfer Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 48, 10 pages. https://doi.org/10.1145/2503210.2503260

[32] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[33] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). San Diego, CA, USA. http://arxiv.org/abs/1409.1556

[34] The tcpdump Group. 2010-2021. Tcpdump. https://www.tcpdump.org

[35] Veronika Thost and Jie Chen. 2021. Directed Acyclic Graph Neural Networks. *arXiv preprint arXiv:2101.07965* (2021).

[36] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 306–324. https://doi.org/10.1145/3132747.3132762

[37] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 407–420. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini

[38] Thomas Volkert, Florian Liers, Martin Becke, and Hakim Adhari. 2012. Requirements-oriented path selection for multipath transmission. In *Proceedings of the Joint ITG and Euro-NF Workshop on Visions of Future Generation Networks (EuroView), Würzburg, Bayern/Germany*, Vol. 1.

[39] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. 2011. OpenFlow-Based Server Load Balancing Gone Wild. *Hot-ICE* 11 (2011), 12–12.

[40] Yi Wang, Ya-nan Jiang, Qiufang Ma, Chen Tian, Bo Bai, and Gong Zhang. 2019. RDMA Load Balancing via Data Partition. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, Valencia, Spain, 1–8. https://doi.org/10.1109/ICCCN.2019.8847077

[41] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, Edwin R. Hancock Richard C. Wilson and William A. P. Smith (Eds.). BMVA Press, York, UK, Article 87, 12 pages. https://doi.org/10.5244/C.30.87

[42] Ming Zhang, Junwen Lai, Arvind Krishnamurthy, Larry Peterson, and Randolph Wang. 2004. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. In *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/2004-usenix-annual-technical-conference/transport-layer-approach-improving-end-end

[43] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 523–536. https://doi.org/10.1145/2829988.2787484

[44] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (Irvine, California, USA) *(CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 313–327. https://doi.org/10.1145/2999572.2999593