An Integrated Tutorial on InfiniBand, Verbs, and MPI

Patrick MacArthur, *Student Member, IEEE*, Qian Liu, Robert D. Russell, *Life Member, IEEE*, Fabrice Mizero, Malathi Veeraraghavan, *Senior Member, IEEE*, and John M. Dennis

***Abstract*—This tutorial presents the details of the interconnection network utilized in many high performance computing (HPC) systems today. "InfiniBand" is the hardware interconnect utilized by over 35% of the top 500 supercomputers in the world as of June, 2017. "Verbs" is the term used for both the semantic description of the interface in the InfiniBand architecture specifications, and the name used for the functions defined in the widely used OpenFabrics alliance implementation of the software interface to InfiniBand. "Message passing interface" is the primary software library by which HPC applications portably pass messages between processes across a wide range of interconnects including InfiniBand. Our goal is to explain how these three components are designed and how they interact to provide a powerful, efficient interconnect for HPC applications. We provide a succinct look into the inner technical workings of each component that should be instructive to both novices to HPC applications as well as to those who may be familiar with one component, but not necessarily the others, in the design and functioning of the total interconnect. A supercomputer interconnect is not a monolithic structure, and this tutorial aims to give non-experts a "big-picture" overview of its substructure with an appreciation of how and why features in one component influence those in others. We believe this is one of the first tutorials to discuss these three major components as one integrated whole. In addition, we give detailed examples of practical experience and typical algorithms used within each component in order to give insights into what issues and trade-offs are important.**

***Index Terms*—InfiniBand, verbs, MPI, flow-control, virtual lanes, congestion control, channel semantics, memory semantics, completion handling, point-to-point operations, collective operations.**

I. Introduction

MOST scientific High Performance Computing (HPC) applications today run on supercomputing platforms that provide thousands of cores communicating over high speed interconnects [1]–[3]. InfiniBand [4]–[6] is the network architecture used for inter-processor communication in over 35% of the top 500 supercomputers as of June, 2017 [7]. Applications in commercial clusters typically communicate via a standard sockets interface to the software TCP/IP transport and network layers that run over a hardware Ethernet Network Interface Card (NIC). In InfiniBand clusters a hardware NIC is replaced by a hardware Host Channel Adapter (HCA) that directly provides the InfiniBand transport and network layers (comparable to the TCP and IP software layers) as well as the link and physical layers (comparable to Ethernet hardware layers). Instead of a sockets interface, InfiniBand provides a Verbs interface that directly accesses the HCA, and HPC applications typically access the Verbs interface via the standard Message Passing Interface (MPI) [8], [9], which provides a more convenient higher-level transport-independent interface implemented to allow HPC applications using it to run over many different network architectures. Other applications and protocols that run over or have been ported to InfiniBand (IB) include High Frequency Trading (HFT) [10], key-value systems [11], [12], virtual machine migration [13], having the kernel Virtual Memory (VM) system fetch pages from remote memory via InfiniBand (IB) [14], the Hadoop Distributed File System (HDFS) [15], the Sockets Direct Protocol (SDP) [16], and other storage protocols [17]–[19].

Previous tutorials have focused on a single layer of the HPC stack, such as InfiniBand [6], [20] or MPI [21]–[24], or have covered a larger number of architectures and software stacks in less detail [25]. This tutorial is one of the first to discuss in an integrated fashion the layers under an HPC application as shown in Fig. 1: the HCA, the Verbs Application Programming Interface (API), and the MPI. Our goal is to provide a comprehensive overview of the interactions within and between the various components necessary to communicate in a modern, high performance computing cluster. Our audience is non-experts new to HPC as well as experts in one component who would like to understand the other components and their interactions. Insights are given into which features of InfiniBand are most useful to MPI applications, and how verbs are used by common MPI communications functions. We give examples of some of the algorithms and techniques utilized within each component, and lessons learned from practical experience. The results of some benchmarks for verbs and MPI are also presented as an example of performance considerations important to HPC users.

Manuscript received December 1, 2016; revised May 31, 2017; accepted July 11, 2017. Date of publication August 29, 2017; date of current version November 21, 2017. This work was supported by the National Science Foundation under Grant OCI-1127228, Grant OCI-1127340, and Grant OCI-1127341. *(Corresponding author: Robert D. Russell.)*

P. MacArthur and R. D. Russell are with the Computer Science Department, University of New Hampshire, Durham, NH 03824 USA.

Q. Liu is with the Mathematics and Computer Science Department, Rhode Island College, Providence, RI 02908 USA.

F. Mizero is with Center for Open Science, Charlottesville, VA 22903 USA.

M. Veeraraghavan is with the Department of Electrical and Computer Engineering, University of Virginia, Charlottesville, VA 22911 USA.

J. M. Dennis is with National Center for Atmospheric Research, Boulder, CO 80307 USA.

Digital Object Identifier 10.1109/COMST.2017.2746083

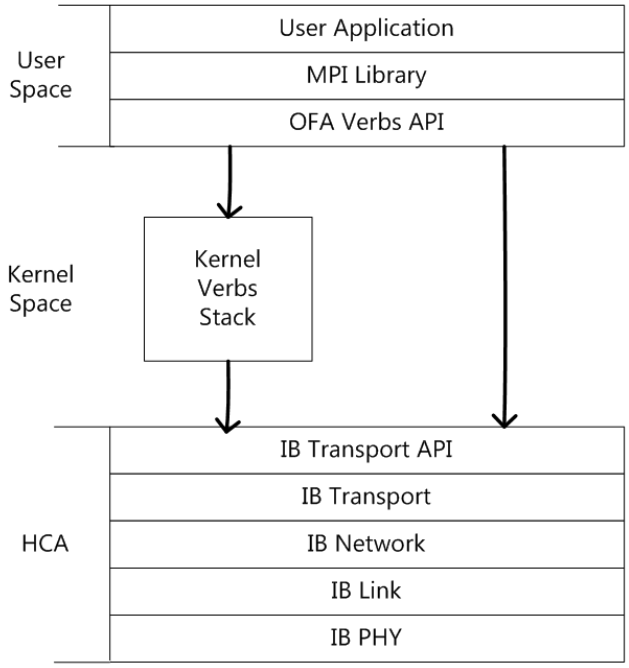


Fig. 1. InfiniBand (IB), Verbs, MPI, and application Layering.

The rest of this tutorial is organized to explain the layers indicated in Fig. 1 from the bottom up. See [26] for a tutorial on data centers that shows their organization and contains a brief discussion of InfiniBand in that context.

Section II discusses the InfiniBand (IB) layers in Fig. 1, giving an overview of its distinguishing features followed by separate subsections that explain: Section II-A its architecture and addressing; Section II-B its physical layer; Section II-C its basic packet format; Section II-D its link layer including link-level flow control, error control, and support for virtual lanes; Section II-E its network layer including subnet packet forwarding, subnet management, commonly used packet routing algorithms, and Software Defined Networking (SDN); and Section II-F its transport layer including the services offered, transport headers, connection management, end-to-end error control, congestion control, congestion detection, packet injection rate adjustment, and other approaches to congestion control.

Section III discusses the OFA Verbs Application Programming Interface (API), an open-source InfiniBand-specific alternative to the traditional sockets interface that exposes the unique features of the InfiniBand architecture to higher-level applications such as the MPI library. This section is subdivided into separate subsections: Section III-A comparison of sockets and verbs; Section III-B introduction to verbs; Section III-C verbs for channel semantics, including sending data and receiving data; Section III-D completion handling, including "busy-polling" and "wait-wakeup" techniques, Section III-E memory registration, a unique InfiniBand requirement unknown to socket users; Section III-F verbs for memory semantics; and Section III-G practical experience using verbs, with attention to performance concerns.

Section IV discusses MPI and how it utilizes the verbs API to implement various kinds of interprocess communication paradigms. This section is subdivided into: Section IV-A point-to-point communication; and Section IV-B collective operations such as MPI_Barrier, MPI_Reduce and MPI_Allreduce.

Section V discusses some experiments with standard benchmarks run on systems organized as shown in Fig. 1. It starts with an overview of the test systems and their components and tools, followed by the subsections: Section V-A wire protocol analysis; Section V-B verbs performance analysis; Section V-C MPI point-to-point performance analysis; and Section V-D conclusions.

Section VI discusses future directions in the continuing evolution of InfiniBand, especially its accommodation of the rapidly evolving Non-Volatile Memory (NVM) technologies.

Section VII concludes with a summary of the topics discussed in this tutorial.

Because of the large number of acronyms commonly used in the computing fields we are covering, an alphabetical list of those referenced in this tutorial follows Section VII.

II. INFINIBAND

InfiniBand (IB) is a packet-switched networking technology defined by the InfiniBand ^{SM} Trade Association (IBTA) [4] as a high bandwidth, low latency interconnect for data center and HPC clusters. Its distinguishing transport level features are:

- "message passing" – the unit of transfer available to IB applications is a "message", there is no concept of a "byte stream" as in TCP, and therefore no need for layers above IB that are designed around messages, such as MPI, to incur the overhead of tracking message boundaries within a stream.
- "zero copy" – direct memory-to-memory transfers across the interconnect with no intermediate buffering or copying by software in an end node's memory. Typical IB switches [27] utilize cut-through designs which have a small amount of buffer space to hold packets only as needed for queueing purposes.
- "kernel bypass" – direct user-level access to the hardware Host Channel Adapter (HCA) with no operating system intervention during data transfers. For example, when sending a large message, the HCA hardware on the sending node segments it into packets sized for transmission as it puts the data directly from the user's memory onto the underlying wire. The receiving node's HCA hardware reassembles these packets into the original message directly in the user's memory.
- "asynchronous operation" – to easily enable overlap of computation with data transfers as well as making it easier to gain better bandwidth utilization by keeping multiple transfers in progress simultaneously.

This section provides an overview of InfiniBand organized into the following subsections: Section II-A architecture and addressing; Section II-B physical layer; Section II-C basic packet format; Section II-D link layer; Section II-E intra-subnet packet forwarding and routing; Section II-F transport layer. Table I gives a concise comparison between some features in InfiniBand and Ethernet.

TABLE I
COMPARISON OF INFINIBAND AND ETHERNET

Feature	Ethernet	InfiniBand
Host adapter	Network Interface Card (NIC)	Host Channel Adapter (HCA)
Programming model	Sockets	Verbs
Layer 3 addressing	Internet Protocol (IP) address	Global Identifier (GID): 64-bit subnet ID assigned by SM plus 64-bit Globally Unique Identifier (GUID) assigned by HCA manufacturer
Forwarding tables	Distributed control; each switch discovers neighbors independently	Centralized control via Subnet Manager (SM)
Layer 2 addressing	Media Access Control (MAC) address statically assigned by NIC manufacturer	Local IDentifier (LID) dynamically assigned by SM
Packet capture	Use standard operating system capture tools, such as Wireshark [28] or tcpdump [29]	Use a vendor-specific tool such as ibdump [30] from Mellanox
Flow control	Pause frames or Priority-based Flow Control	Credit-based link-by-link flow control
Data rate	1, 2.5, 5, 10, 25, 40, 50, 100 Gbps	8, 16, 32, 54, 100 Gbps (for common 4x link width)
Error control	Single 32-bit CRC trailer recalculated by each switch	End-to-end 32-bit Invariant Cyclic Redundancy Code (ICRC) plus 16-bit Variant Cyclic Redundancy Code (VCRC) recalculated by each switch
Offload support	With exception of TCP Offload Engines (TOEs), limited to checksum and large segment offloads	Full offload at all levels including transport-layer
Congestion control	Drop packets, which triggers window size reduction and retransmissions at upper layers	Transport-layer congestion notification flags to avoid switch-to-switch spread of congestion

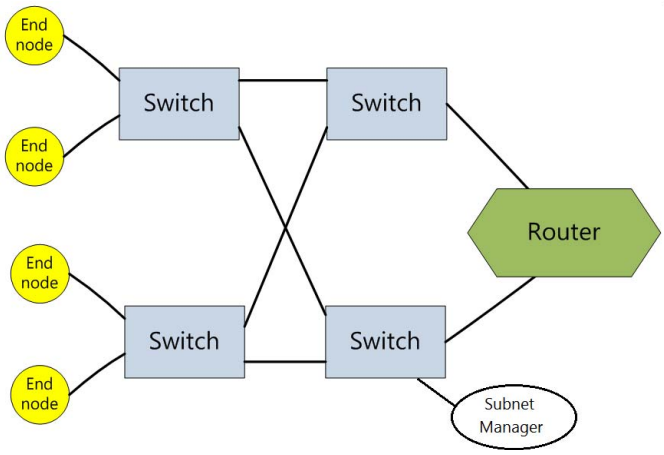


Fig. 2. InfiniBand network architecture [4].

A. Architecture and Addressing

Architecture: Fig. 2 shows the InfiniBand Architecture (IBA) of a single subnet with end nodes (typically computers), switches, a Subnet Manager (SM), and optionally a router. End nodes attach to the subnet via Host Channel Adapters (HCAs), which are comparable to Ethernet Network Interface Cards (NICs). Each HCA supports one or more physical ports.

It is important to note that the IBA Specifications define the internal behavior of the various InfiniBand components, but in many cases not the details of their implementation. As long as a component, such as a switch, HCA, or connecting cable, conforms to the functionality and interoperates with conforming components from other manufacturers as required by the specifications, a manufacturer is free to design and implement it according to its own proprietary design by which it wishes to differentiate itself from other manufacturers. Examples of such distinguishing characteristics would be performance, cost, capacity, energy consumption, heat generation, etc.

Transport layer communication is done between a Queue Pair (QP) in each communicating HCA port. A Queue Pair consists of a Send Queue (SQ) (used to send out-going messages) and a Receive Queue (RQ) (used to receive incoming messages), as explained in detail in Section III-C. User applications create, use, and tear down queue pairs via the verbs API. A "queue" is analogous to a First-In First-Out (FIFO) waiting line – items must leave the queue in the same order that they enter it. Beyond that requirement, the IBA standard does not specify how a queue should be implemented internally in the HCA hardware. Each manufacturer must provide a driver as part of the OFA Verbs API whose input will be function calls and data structures that are defined in great detail by the API. So for example, there are API functions to add structures to the SQ and RQ, but the internal mechanism that performs those functions, and the format of the elements on those queues, are designed by the manufacturer. Of course,

the format and use of the packets sent and received over the wire are precisely defined by the IBA specifications so that HCAs of different manufacturers can interoperate.

Unlike the distributed nature of routing and management in IP/Ethernet networks, InfiniBand specifies a required Subnet Manager (SM) as a centralized entity which can be implemented in either hardware or software, and which must be running on either a switch or an end node somewhere in each subnet in order to perform several functions critical to the operation of that InfiniBand subnet, as explained in the paragraph entitled "**Subnet Management**" in Section II-E. Even back-to-back connections between HCAs must have a SM running on one side or the other in order to function properly. A subnet may have multiple SMs, in which case they will choose among themselves a "master" SM and the others become "standby" SMs, ready to take over should the master SM fail.

All hosts and switches run a Subnet Management Agent (SMA). Communication between the SM and SMAs is performed using a class of Management Datagram (MAD) called the Subnet Management Packet (SMP). The SM configures all HCAs and switches in its subnet to enable connectionless packet forwarding using a simple address called a Local IDentifier (LID). Host applications generate InfiniBand packets that carry source and destination LIDs in their headers. Switches perform packet forwarding based on destination LIDs.

Each link is divided into Virtual Lanes (VLs), each of which functions as a virtual link. Each VL consists of a Send Queue and a Receive Queue, and flow control is performed separately for each VL. Administrators can configure different Service Levels (SLs) to map flows onto different VLs to implement Quality of Service (QoS) [20].

The *Router* component of the architecture shown in Fig. 2 is intended to interconnect locally addressed InfiniBand subnets. But this tutorial will not discuss routers, further because, as stated in [4, Sec. 19.2], "the present IBA Router specification does not cover the routing protocol nor the messages exchanged between routers." Nonetheless, IB has been successfully utilized without routers in a production environment between two clusters over distance [31].

Addressing: There are three types of addresses: Locally unique (within a subnet) Identifiers (LIDs), Globally Unique (within a network) Identifiers (GUIDs), and Global Identifiers (GIDs). A LID is a 16-bit number assigned at startup by the SM to each HCA port and switch (one per switch). A per-port LID Mask Control (LMC) specifies the number of least-significant bits of that port's LID to allow for multipathing. For example, if the LMC assigned to a HCA port is n, then 2^n different, consecutive LIDs can be used to reach that port, and the SM, which configures the LID-based Forwarding Data Bases (FDBs) in switches, can provide multiple paths (disjoint wherever possible) through the fabric to reach that port. Although a LID has 16 bits, LIDs 0x0000 and 0xFFFF are reserved, and LIDs 0xC000 through 0xFFFE inclusive are reserved for multicast groups, leaving only LIDs 0x0001 through 0xBFFF (49151) for assignment to HCA ports and switches in the subnet.

Since LIDs are simple 16-bit integers assigned by the SM to be unique within the subnet, they are used as a "flat" address to uniquely identify each port in the subnet. This is the simplest and fastest type of network addressing, because a "flat" address has no substructure and requires no decoding to be used. Therefore, unicast forwarding within a switch requires just using the LID as an index into a FDB whose maximum possible size of 49,152 entries can easily be handled by today's memory technologies. Of course, LID addressing limits the maximum number of addresses in a subnet, which may in turn prevent scaling to exascale clusters unless IB routers are defined and implemented.

A Globally Unique Identifier (GUID) is a manufacturer-assigned EUI-64, a 64-bit, globally unique identifier (hence it is persistent, unlike LIDs that are dynamically assigned by the SM). In Ethernet, the EUI-64 address is derived from the 6-byte Media Access Control (MAC) address [32], and is used for Stateless Address AutoConfiguration (SLAAC) of IPv6 addresses [33]. Similarly, in IB networks, the GUID is used in the Interface ID field of IPv6 addresses for inter-subnet communications.

A Global Identifier (GID) is defined in the IBA specification [4] to be an IPv6 address. As with IPv6 for Ethernet networks, the first 8 bytes are used for the subnet prefix, which in InfiniBand networks is a unique Subnet ID. The default GID for every port uses the port's GUID in the second 8 bytes. A port can also have additional locally administered GUIDs. Thus, GUIDs and GIDs are required primarily for inter-subnet communications. Because routers are required to connect subnets, the rest of this tutorial only refers to flat LID addressing, as we are focusing on the operations of a single IB subnet typical of current supercomputers and datacenters.

Subnet initialization is performed by the SM, which first executes topology discovery to learn the GUIDs of all HCA and switch ports in the subnet and to assign them unique LIDs. The fact that there are only 49151 LIDs numbers available for assignment to ports is already a constraint on the size of IB subnets, so developing a feasible deadlock-free set of routes (i.e., no cycles or loops in the routes) that minimizes the number of LIDs is a major goal. However, [34] has proven achieving this goal is NP-complete and has also developed a number of heuristics whose performance has been evaluated by simulation.

Next, the SM runs one of the routing algorithms described in Section II-E to determine switch FDBs. It then downloads these tables to all the switches. These SM steps are required at startup before an IB network can be used to carry user data. For large subnets with many thousands of LIDs, this can take considerable time.

B. Physical Layer

We briefly describe the PHYsical Layer (PHY) properties of InfiniBand. The exhaustive definition of the InfiniBand physical layer is given in the Volume 2 standard [35]; we only provide a high-level overview here.

InfiniBand defines several wire speeds. The initial version of the specification supported Single Data Rate (SDR)

TABLE II
STANDARD INFINIBAND SPEEDS, AND THEIR PER-LANE SIGNALING AND THEORETICAL DATA RATES IN GIGABITS PER SECOND (GBPS). ALL PER-LANE SIGNALING RATES ARE MULTIPLES OF 0.15625 GBPS [35]

Name	Signaling Rate	Encoding	Data Rate
Single Data Rate (SDR)	2.5	8b/10b	2
Double Data Rate (DDR)	5	8b/10b	4
Quad Data Rate (QDR)	10	8b/10b	8
Fourteen Data Rate (FDR)	14.0625	64b/66b	13.64
Extended Data Rate (EDR)	25.78125	64b/66b	25

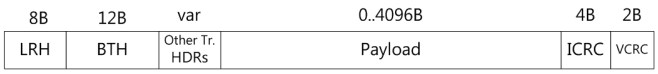


Fig. 3. Local (within a subnet) data packet format [4].

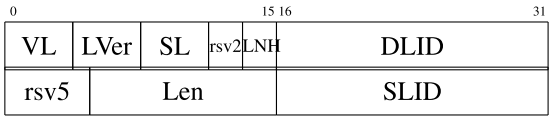


Fig. 4. Local Routing Header (LRH) format [4]. The rsv2 and rsv5 fields are reserved. Each line is 32 bits wide.

that operates at a signaling rate of 2.5 Gigabits per second. More recent revisions of the InfiniBand specification offer faster speeds as displayed in Table II [35]. Note that achieving these speeds requires that both the Peripheral Component Interconnect express (PCIe) bus [36] and the host memory also support them [37]. Each link contains 1, 4, 8, or 12 lanes, with 4 lanes being the most common, leading to a signaling rate of $4 \times 2.5 = 10$ Gbps across an SDR link. Each physical lane operates independently, and each packet, along with surrounding control symbols, is striped across all available physical lanes.

InfiniBand speeds up through Quad Data Rate (QDR) use 8b/10b encoding [35] (which is also used in Gigabit Ethernet [38]). This encoding enables receivers for high-speed serial transmission over distance to function properly. For each of the 256 possible 8-bit data bytes, this encoding scheme defines two 10-bit symbols designed such that when a transmitter is given an 8-bit data byte, it transmits the corresponding 10-bit symbol that keeps the absolute difference between the total number of 0's and 1's sent on the physical wire less than or equal to two. Doing so guarantees frequent transitions between 0 and 1, which allows the receiver to correctly detect each bit in the signal. Additionally, since 488 of the 1024 possible encoded 10-bit symbols are invalid,[1] this allows fast detection of bit errors which produce these invalid symbols. However, this encoding reduces the theoretical maximum data rate seen by applications from 10 Gbps to 8 Gbps for a 4x SDR link. Fourteen Data Rate (FDR) and Extended Data Rate (EDR) use 64b/66b encoding [39] as used in 10 Gigabit Ethernet, which lowers the encoding overhead (symbol bits per data bit) from 25% to 3.125%.

InfiniBand can run over copper or optical cables, depending on the HCAs and switches in use. Copper cables use QSFP+ or CXP connectors. Passive copper cables support lengths from 1 to 5 meters [40]. While the InfiniBand standard allows passive optical cables, modern high-speed optical cables are *active* cables, which contain electronic components to amplify or otherwise manipulate the signal to handle higher speeds or transmission lengths; active optical InfiniBand cables on the market at the time of writing can reach a distance of 100 meters. For this reason, passive optical fiber specifications are only defined for SDR, DDR, and QDR speeds—implementers use active optical cables for higher speeds, and apart from the electrical transceivers at the ends of the cables, the properties of these active cables are not specified in the standard.

InfiniBand link partners use a link training process similar to Ethernet autonegotiation to determine link width, speed, and other physical attributes. Active cables contain EEPROMs which identify parameters to use for link training.

C. Basic Packet Format

As shown in Fig. 3, packets are of variable length and consist of a Local Routing Header (LRH) shown in Fig. 4, a Base Transport Header (BTH) shown in Fig. 8 that is discussed below in the **Transport Headers** paragraph, Other Transport Headers (Other Tr. HDRs), a variable length payload, an Invariant Cyclic Redundancy Code (ICRC) that detects errors in the fields that do not change during transit between source and destination nodes, and a Variant Cyclic Redundancy Code (VCRC) that detects errors in the fields that potentially could change on every hop. The size of the packet payload is limited by the Maximum Transmission Unit (MTU), a link-level parameter that can be configured administratively to a power of 2 in the range 256 to 4096 bytes inclusive. HCA and switch ports are normally configured to an MTU of 2048 or 4096.

The fields of the LRH are shown in Fig. 4. The important fields are the Source LID (SLID), the Destination LID (DLID), the Virtual Lane (VL), and the Service Level (SL). Details of how these fields are used are described in Section II-E.

D. Link Layer

In this subsection, we first discuss the credit-based flow control mechanism used by InfiniBand. We then discuss error control and conclude by briefly discussing the virtual lane mechanism. The link layer is configured by the Subnet Manager (SM) as part of "link initialization", which occurs when an HCA or switch attached to that link is powered on, when a link times out during error recovery, or when a link's LID is changed by the SM due to either a subnet topology change or a failure of the master SM.

Link-Layer Flow Control: Ethernet networks do not use link-by-link flow control. While the Ethernet standard [38] includes a Pause frame, it is rarely used since it stops all traffic on the link. Thus, in Ethernet networks, senders can send

[1]In addition to 512 10-bit symbols defined by the 256 8-bit data bytes, there are 24 10-bit symbols (with no corresponding 8-bit data bytes) defined for low-level link control, giving a total of 536 valid 10-bit symbols.

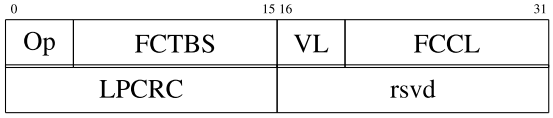


Fig. 5. Flow Control Packet (FCP) format [4].

packets too fast and overflow switch buffers causing packet losses. Losses are recovered via upper-layer protocol logic, for example, through TCP end-to-end error control. Data center networks mitigate congestion using Priority-based Flow Control [41], which allows pausing different traffic flows independently, unlike the standard Ethernet Pause frame which stops all traffic indiscriminately.

In contrast, InfiniBand strictly avoids packet loss by employing link-by-link flow control, which prevents a data packet from being sent from one end of a link if there is insufficient space to receive the packet at the other end of that link. This does not prevent packet loss due to other types of link errors, as discussed below under **Error Control**, but is intended to prevent packets being dropped due to buffer overflow in switches, a principle cause of transport-layer retransmissions that occur in TCP/IP/Ethernet for example. However, although this mechanism prevents packet loss due to congestion, it allows the effect of one congested link to spread backwards in the network, creating victim flows even for traffic that does not traverse the congested link, as discussed in Section II-F.

Flow control credits are measured in 64-byte units called "blocks". For example, in order to send a data packet consisting of 2048 bytes of application data plus headers and CRCs, a physical port on an HCA or switch requires the other end of the physical link to have at least 33 blocks of available space.

Each end of a data link maintains several counters to support flow control, and Flow Control Packets (FCPs), shown in Fig. 5, are used to synchronize these counters between both ends of each data link. FCPs may be sent as often as necessary to enable efficient link utilization. For example, a switch may send an FCP as soon as it has sent a block to its next-hop port. Additionally, the IBA standard [4] requires that a flow control packet be sent at most 65,536 symbol times after the previous flow control packet.

For each VL, a switch or HCA port maintains four 12-bit counters.

Two 12-bit counters control data packet sending:

- ***TBS*** (Total Blocks Sent) indicates the total number of blocks (module 4096) this side has sent out over this link since the last link initialization. This counter is reset to 0 on each link initialization, and each time this side successfully sends a packet out over this link it increments this counter by the number of blocks in that packet.
- ***CL*** (Credit Limit) sets a limit on the TBS value, such that this side cannot send a packet over this link if doing so would cause the TBS counter to be incremented beyond this limit (module 4096). This counter is reset to 0 on each link initialization, and is set to the value of the Flow Control Credit Limit (FCCL) field from each FCP received from the other side of this link.

Two 12-bit counters control data packet receiving:

- ***ABR*** (Adjusted Blocks Received) indicates the total number of blocks (module 4096) successfully received by this side of this link. This counter is reset to 0 on each link initialization, is set to the value of the Flow Control Total Blocks Sent (FCTBS) field from each FCP sent by the other end of the link, and each time this side successfully receives a data packet it increments this counter by the number of blocks (module 4096) in that packet.
- ***FB*** (Free Blocks) indicates the total number of free blocks remaining in this side's input receive buffer. This counter is set to the full input receive buffer size on each link initialization, is decremented by the number of blocks received each time a packet is successfully received into this side's input buffer, and is incremented by one each time a block is successfully delivered from the input buffer by this side. A switch delivers a block by sending it out over the next-hop port. An HCA delivers a block by storing it directly into an application supplied receive region in the local host memory.

Fig. 5 shows the contents of each flow control packet. These fields are populated as follows:

- ***OP*** (4-bit OPeration) 0x1 for a flow control init packet, 0x0 for a normal flow control packet
- ***FCTBS*** (12-bit Flow Control Total Blocks Sent), the sender's TBS counter
- ***VL*** (4-bit Virtual Lane)
- ***FCCL*** (12-bit Flow Control Credit Limit), the sender's ABR counter plus the maximum of its FB counter and 2048
- ***LPCRC*** (16-bit Link Packet CRC), covers the first 4 bytes of the FCP
- ***rsvd*** (16 unused/reserved bits).

In order to send a data packet containing N blocks, the sender must have $0 \leq$ (CL - (TBS + N)) ≤ 2048 (all calculations modulo 4096). If this condition is not true, the sender may not send this N-block data packet until future FCPs are received that will enable this condition to become true. The HCA for each output port maintains a management counter called "PortXmitWait" which counts the number of clock ticks during which data was ready to be transmitted but there were not enough credits available to send it. One use for this counter is in a congestion control scheme discussed in subsection **Other Approaches to Congestion Control** of Section II-F.

FCPs are used to control the flow of a data packet across the wire from *port A* to *port B* as follows:

1) The condition mentioned above is satisfied, so *port A* increments its local TBS by N and sends a data packet containing N blocks to *port B*.
2) The data packet from *port A* arrives in the input buffer of *port B*, which causes *port B* to decrement its local FB counter and increment its local ABR counter by N.
3) *Port A* sends an FCP in which the FCTBS value is set to the value of *port A*'s newly updated local TBS counter, and the FCCL value is set to the sum of *port A*'s local ABR and FB counters.

4) After receiving this FCP, *port B* updates its local ABR and CL values based on the FCTBS and FCCL fields of that FCP, respectively.
5) *Port B* sends an FCP to *port A*. The FCTBS value is set to the value of *port B*'s TBS counter (unchanged from what it was at the start of this exchange), and the FCCL value is set to the sum of *port B*'s local ABR and FB counters (as updated by this exchange).
6) *Port A* receives this FCP from *port B* and updates its local ABR and CL counters based on the FCTBS and FCCL fields in that FCP.

Error Control: Link-layer error detection is supported using the CRCs [42] shown in Fig. 3. The ICRC field covers all headers and payload in a packet with the exception of the VL field in the LRH and several fields of the BTH which can be changed by switches and are therefore included in the VCRC. Thus the VCRC is set and checked on every link, while the ICRC is set by the original source HCA and checked only by the final destination HCA.

For the ICRC, a 32-bit CRC is used, and the CRC polynomial is the same as that used by Ethernet. For the VCRC, a 16-bit CRC is used, and the CRC polynomial is the same as that used in HIPPI-6400 [43]. In both cases the CRC calculation is done using both bit and byte ordering consistent with the CRC calculation done by Ethernet.

At the link-layer, optional forward error correction encoding at the symbol level is defined only at FDR and higher speeds, due to the 64b/66b encoding used at these rates [35]. For lower rates, or where forward error correction fails, errored packets are dropped and error counters are updated for management actions, if necessary.

As discussed below in the **Transport-layer Services** and **End-to-end Error Control** subsections, the HCA takes no recovery action for unreliable services when packets are dropped, because many applications using unreliable services, such as video streaming or High Frequency Trading (HFT), are time-critical and "old" (i.e., retransmitted) data is effectively useless. However, for reliable services the HCA utilizes strict packet ordering rules, end-to-end ACKnowledgement (ACK) packets, and Packet Sequence Numbers (PSNs) and the ACK request bit in the Base Transport Header (BTH) (shown in Fig. 8) on each packet to deal with a dropped packet by retransmitting that packet and all packets already transmitted after it (if any).

Support for Virtual Lanes (VLs): Virtual Lanes (VLs) provide a mechanism for creating multiple virtual links that share a single physical link [4]. There are two types of VLs, data VLs which transmit data packets, and management VLs which transmit only management packets.

Each physical output port may optionally support up to a maximum of 16 VLs as determined by the manufacturer (one required Management VL, which is Virtual Lane 15 (VL15), and up to 15 data VLs). Most Mellanox HCAs support 8 data VLs [44]. The SM can configure the number actually utilized.

To enable consistent policy across a fabric, administrators assign a Service Level (SL) to a flow. The SL in a packet header remains unchanged as the packet traverses through the subnet, while the VL may change at each hop. Each device has a Service Level to Virtual Lane (SL2VL) mapping table which is used to assign each service level to an appropriate virtual lane.

All the VLs at each output port in a device share a common buffer space whose size is determined by the manufacturer and is usually kept small, on the order of 32 Kilobytes per VL [45], due to cost and chip area considerations as well as the desire to keep latency low, since large buffers can introduce delay [46]. For each output port there is a VL arbiter that determines how the output physical link bandwidth is shared between the competing VLs. Each arbiter is controlled by three configurable components: a *high-priority table*, a *low-priority table*, and a *high priority limit*. The arbiter serves as many packets from the high priority table as it can, up to the high priority limit, before serving a single packet from the low priority table. The full algorithm is shown in Fig. 6.

Implications of this algorithm are:

- Packets are always sent as a whole, even though the weights are in units of 64-byte blocks.
- If the *high-priority limit* is configured to 0, only one high-priority packet may be sent before giving the low-priority packets a chance.
- Normally only 1 low-priority packet is sent at a time.
- If neither table contains an entry for a VL, packets from that VL may never be sent.
- Within a VL, packet ordering in the output queue is maintained on the wire.
- VLs may appear in both tables and/or multiple times within a table, enabling packets to be served with both low latency and high throughput.

Each active VL configured on a port, except VL15, maintains a "Watchdog Timer" that monitors the arrival of flow control updates. This timer is reset on link initialization and on each arrival of a flow control update. If this timer expires without receiving an update, a "flow control update error" occurs and the link recovers by initiating PHYsical Layer (PHY) retraining. The time out interval is defined [4] to be about 400,000 symbol times with an accuracy in the range [196,000..412,000].

E. Intra-Subnet Packet Forwarding and Routing

In this subsection, we discuss packet forwarding, the Subnet Manager (SM), and the SM's role in packet forwarding.

Packet Forwarding: An InfiniBand switch implements three types of packet forwarding [4]: directed-route Subnet Management Packet (SMP) forwarding, unicast packet forwarding, and multicast packet forwarding. This paper only reviews the details of the unicast packet forwarding, because OpenMPI [47], discussed in detail in Section IV, only uses reliable connection service and that service uses unicast packet forwarding. However, other approaches to MPI implementation using unreliable services have also been investigated [48], [49], including the use of unreliable multicast for the MPI_Bcast collective operation [50].

Directed-route SMP forwarding is only used before the SM has assigned LIDs or has set up the switch FDBs. Typically

```
loop
    SEND(next queued VL15 packet)
    SEND(next flow control packet)
    next ← HEAD(VL[active.pointer.vl].output_queue)
    if active.weight_counter > 0 ∧ next ∧ has_credits(active.pointer.vl,next.length) then
        weight_counter ← weight_counter − next.length
        SEND(next)
        last_sent ← next
        if active = low_priority_table then
            active ← high_priority_table
        else if high_priority_limit ≠ 255 then
            high_priority_counter ← high_priority_counter − next.length
            if high_priority_counter ≤ 0 then
                high_priority_counter ← 4096 × high_priority_limit
                active ← low_priority_table
            end if
        end if
    else
        active.pointer ← active.pointer.next
        weight_counter ← active.pointer.weight
        if next = last_sent then
            active ← active.other
        end if
    end if
end loop
```

Fig. 6. Virtual lane arbitration algorithm.

the Destination LID (DLID) field in the LRH (LRH:DLID) of a SMP is 0xFFFF. Unicast packet forwarding is used to forward packets with a DLID field within the range [0x0001, 0xBFFF] identifying a single destination port with that LID.

Multicast packet forwarding is used to forward packets with a DLID field within the range [0xC000, 0xFFFE] that indicates all destination ports belonging to that multicast group. Multicasting is only provided for unreliable datagram service, which means that messages can be lost during transit without any indication to either the sender or receiver. Multicasting is designed so that each switch replicates the packet only to its output ports that ultimately lead to one or more unique members of the multicast group as determined by the subnet manager at the time each member joins the multicast group. Similarly, both sending and receiving HCAs are responsible for replicating packets if more than one member of the multicast group is located on that HCA.

A switch uses the fields in the LRH to forward a packet as follows:

1) When an input packet arrives at a port, the DLID in it's LRH is used as an index into the switch's FDB.
2) That FDB entry points to the metadata for the output port on which to forward this packet.
3) The SL in this packet's LRH is used as an index into the SL2VL mapping table that is part of that output port's metadata.
4) That SL2VL table entry determines which output VL to use when forwarding this packet on that output port.

When only Virtual Lane 0 (VL0) is used, packets headed to the same output port from different input ports are served in round-robin mode on a packet-by-packet basis. Switch fabrics are typically non-blocking, i.e., whenever an output port is free, it will be cross-connected through the switch fabric to an input port that has a packet to send to that output port. The link-transmission arbiter at the output port selects the next packet to transmit based on the VL arbitration procedure described in Fig. 6.

Subnet Management: According to the IBA Specifications [4], each IB subnet must have one "master" Subnet Manager (SM) active on any switch or node in the subnet. Among its tasks, the SM must discover the IB fabric, initialize it, and regularly sweep the subnet for changes in the topology. OpenSM [51] is an implementation of an InfiniBand SM. During the initial subnet activation, OpenSM performs three essential steps: subnet topology discovery and LID/GID configuration, path computation, and forwarding table configuration. It also programs the SL2VL mapping tables and VL arbitration tables just discussed, and sets congestion control thresholds discussed below under **Connection Management**.

During the subnet discovery and LID/GID configuration step, OpenSM uses directed route Subnet Management Packets (SMPs) to reach nodes/ports and find their attributes such as node type, number of ports, port number, port connectivity, port state, etc. To reach a node prior to LID configuration, a directed route SMP must contain an output port for each switch along its path; this array of ports is referred to as the initial path vector. A reverse path vector consisting of input ports is created as the SMP traverses each node. After processing node attributes

in the response SMP, OpenSM may assign a unique LID or a range of LIDs (if LMC $>$ 0) to each discovered node. Through iterative SMP queries, unique LIDs are assigned to all ports [52].

The IBA Specifications [4] do not specify a particular algorithm for path computation, so during this step OpenSM can be configured (using the -R option) to use one of six currently supported routing algorithms to compute paths between all pairs of ports in the subnet: MinHop [53], UpDown [54], LAyered SHortest path (LASH) [55], Ftree [56], Dimension Order Routing (DOR), and Torus_2QoS. However, additional routing schemes such as Deadlock-free Fast Single-Source-Shortest-Path (DFSSSP) [57] and DownUp [58] are also supported by patched versions to the official OpenSM. See [59] for a comparison of a number of these algorithms as used by various InfiniBand systems.

Routing Algorithms: Following is an overview of three popular routing schemes: MinHop, UpDown, and LASH. An ideal routing scheme should offer deadlock-free, shortest-path, and load-balanced routing.

MinHop is the default routing algorithm in OpenSM. In the first step, MinHop computes the shortest paths between all pairs of ports in the subnet. In the second step, MinHop computes the FDBs for all the switches in the subnet. The FDB entries are computed from the shortest paths obtained in the first step. In the last step, MinHop uses a link equalization algorithm to balance the number of assigned destinations across all ports of each switch.

UpDown has been established as the de facto routing scheme for irregular topologies. UpDown is a topology-agnostic deadlock-free routing scheme originally developed for Autonet [60]. OpenSM's implementation of UpDown consists of three main steps: (i) auto-detection of root nodes, (ii) ranking process, and (iii) routing table computation.

In step (i), for each switch in the subnet, the hop lengths to all Host Channel Adapters (HCAs) are computed. From this information, for each switch, the number of HCAs reachable at each hop value is determined. The switches with longer paths to a greater number of HCAs are then selected as root nodes. In the example subnet shown in Fig. 7, the computed root node is switch `a`. This step can be bypassed if a set of root nodes is specified in the OpenSM configuration file.

In step (ii), the root nodes are first assigned rank 0. Thereafter, a Breadth First Search (BFS) [62] is executed starting with each root node selected to be a "frontier." Each unvisited node reachable within 1 hop from the frontier is assigned rank $k+1$, where k is the rank of the parent node or frontier. In an iterative manner, each switch or terminal node is assigned a rank number. For each link, the port that is closer to a root node is the `up` end. In the example subnet, switches `b, c, d,` and `e` are each assigned rank 1, while switches `f, g, h,` and `i` are each assigned rank 2. If both ports at the ends of a link are at same hop distance from a root node, the port on the switch with the lower identifier is designated the `up` end. Thus, in the case of the $b-c$ link in Fig. 7, the port at switch `b` is the `up` end.

In step (iii), the shortest paths are computed from each switch to all ports in the subnet in order to populate the FDBs.

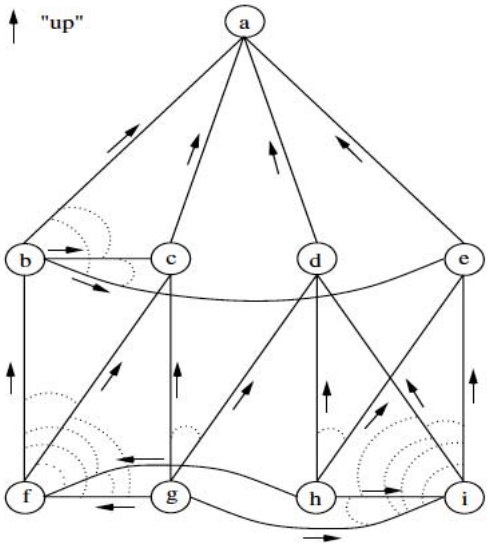


Fig. 7. Example 9-switch subnet with direction labels on links [61] © Springer-Verlag Berlin Heidelberg 2000.

While computing the shortest paths, every route is checked to see whether or not it is a legal route. A legal route is one of the 3 cases: (i) a sequence of up-direction links followed by a sequence of down-direction links, (ii) all up-direction links, or (iii) all down-direction links. For instance, in the example subnet shown in Fig. 7, the path $f-c-g-d$ constitutes an illegal route, where $c-g$ is the forbidden transition. By the end of this step, each switch contains an FDB that maps each destination LID to a corresponding output port.

Despite its popularity and relative simplicity, the original UpDown routing algorithm has been shown to have drawbacks: it provides no guarantee of utilizing the shortest possible paths for all packets, and it tends to concentrate traffic at the root nodes [61], [63]. Techniques to improve the UpDown scheme include Depth-First Search (DFS) Based UpDown [64], Adaptive routing [65], [66], and SMART routing [67].

LASH (LAyered SHortest path [55]) is a deadlock-free topology-agnostic alternative to the UpDown scheme. It relies on the use of multiple layers to prevent deadlocks while providing shortest path and load-balanced routing. The LASH routing scheme has three main steps:

In step (i), LASH computes shortest paths for each $\langle source, destination \rangle$ pair. In step (ii), LASH assigns shortest paths to different layers. A route is added to a layer only if it does not have the potential to cause deadlocks in combination with routes previously added to that layer. This is done by analyzing the channel dependency graph for each layer, since [68] has proven that a necessary and sufficient condition to guarantee deadlock-free routing is the absence of cycles (loops) in this graph. By the end of this step, all paths assigned to a layer are as short as possible and deadlock-free. Finally, in step (iii), to promote load balancing, the number of paths assigned across all layers are balanced wherever possible.

In general, LASH outperforms UpDown, but faces scalability issues. In InfiniBand networks, the concept of layers

can be realized with virtual lanes as described under **Support for Virtual Lanes (VLs)** in Section II-D. In OpenSM, LASH populates the SL2VL mapping tables in each switch. and configures the HCAs to use the appropriate SL for each ⟨*source*, *destination*⟩ pair.

Software Defined Networking: The Software Defined Networking (SDN) paradigm has created significant changes in the designs of switches/routers and networks. Recognizing that the significant cost of switches/routers lies in the control-plane software built into these systems, the SDN paradigm calls for moving this software out of switches/routers into external commodity servers. SDN has been embraced by commercial datacenters that primarily run Ethernet switched networks. However, there is increasing recognition of the potential for the SDN paradigm to have an impact on InfiniBand based High Performance Computing (HPC) systems.

Specifically, Lang and Yuan [69] and Lee *et al.* [70] note that the Subnet Manager (SM) used in InfiniBand subnets is already similar to SDN controllers. However, they propose adding OpenFlow-type per-flow management capabilities to InfiniBand subnet managers. A similar idea is explored to determine the potential of SDN to improve performance of MPI applications using application-aware tagging of packets [71].

Software Defined Optical Networks (SDONs) [72] are being developed for metro- and wide-area optical networks. These advances will soon become necessary in HPC datacenters as more and more optical technologies are integrated into InfiniBand and HPC systems for reasons of power-efficiency [73], [74].

Finally, concepts of using SDN in InfiniBand HPC systems where the compute nodes are virtualized are explored by Cohen [75].

F. Transport Layer

In Ethernet networks, the transport and network layers (TCP/IP) are typically implemented in software, although TCP Offload Engines (TOEs) have become common for high-speed networks. However, InfiniBand specifies native protocols up through the transport layer which are implemented in the hardware Host Channel Adapters (HCAs) at all speeds.

The IB transport layer provides the services directly visible via the Verbs API to the software layers above it, as shown in Fig. 1.

In this subsection, we discuss (1) the types of transport service, (2) the headers used by the InfiniBand transport layer protocol, (3) connection management, (4) transport layer reliability, and (5) congestion control.

Transport-Layer Services: The IB transport-layer defines five types of services: (i) Reliable Connection (RC), (ii) Reliable Datagram (RD), (iii) Unreliable Datagram (UD), (iv) Unreliable Connection (UC), and (v) eXtended Reliable Connection (XRC). Of these, an HCA is required to support RC, UD, and UC, whereas RD and XRC are optional to support. The connection services (RC, UC, and XRC) are based on a client-server model similar to that in TCP, as is discussed in Section III. The reliable services (RC, RD, and XRC) are required to verify that packets are delivered in order, to prevent receiver processing of duplicate and out-of-order packets, and to detect and retransmit erroneous and missing packets. This is accomplished using packet sequence numbers and explicit acknowledgment messages. Upon error detection, the errored or missing packet and all subsequent packets are retransmitted. However, the link-level flow control discussed above eliminates the dropping of packets in congested situations, thereby reducing the need for retransmissions to other types of "hard" errors such as CRC checks, etc. Unreliable services (UD and UC) offer none of the ordering or duplication guarantees, and just silently drop erroneous packets. However, they do generate sequence numbers, which would allow a receiver to detect missing or out-of-order packets, although recovery in these situations is explicitly outside the scope of the IBA standard [4].

Communication is based on Queue Pairs (QPs), where a QP consists of a Send Queue (SQ), a Receive Queue (RQ), and their associated Completion Queues (CQs) which are discussed in Section III-C. The QP concept is comparable to the port number in TCP and UDP, and makes it possible to multiplex many independent flows to the same destination HCA. For example, in connection-oriented services each connection is mapped to a unique QP in the HCA at each end, so that no other traffic can flow between these QPs. In datagram services, one datagram QP can send to any other datagram QP in the network.

Messages in datagram services are limited to a maximum payload length of one MTU at the sender HCA, but connection-oriented services can send messages up to a maximum payload length of 1 Gibibyte ($2^{30} = 1,073,741,824$ bytes). This requires the transport layer to transparently segment messages into packets when they are sent, and to transparently reassemble these packets when they are received. Since the MTU can be different for different links in a connection, the Path-MTU is determined when the connection is established by the Connection Manager as the smallest MTU in use on the path between the source and destination HCAs, and that Path-MTU is used as the size of the individual packets in segmented messages. Note that for datagram services no connection, and therefore no Path-MTU, exists. Thus, if an MTU configured on an intermediate or destination link is smaller than the sender's MTU, that datagram is simply dropped by the link.

The RD service architecture was designed for multi-process applications such as MPI and is discussed in [20]; however, to the best of our knowledge, the optional RD service is not implemented in modern InfiniBand hardware because of its severe limitation that allows only a single outstanding transfer operation per queue pair. The need for it has been replaced by the optional XRC [76] service, which was designed specifically to enable MPI to scale to very large IB clusters without the RD limitation. XRC was introduced in IBA release 1.3 [4] in 2015.

Transport Headers: Since all transport headers appear in packets on the wire, they represent information that must

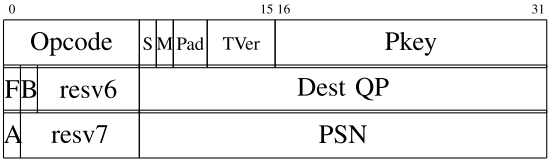


Fig. 8. Base Transport Header (BTH) format [4]. The resv6 and resv7 fields are reserved. Each line is 32 bits wide.

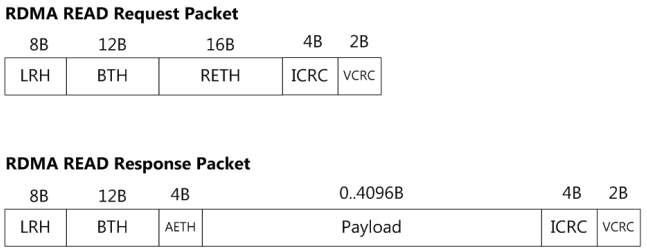


Fig. 9. RDMA READ Request and RDMA READ Response packet formats [4].

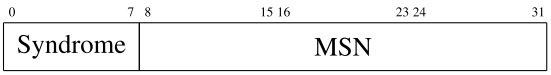


Fig. 10. ACK Extended Transport Header (AETH) format [4]. This line is 32 bits wide.

be interoperable between all switches and HCAs in an IB network, regardless of who manufactures the switch or HCA. Therefore, they are all precisely defined in the IBA Specifications [4], from which the figures that follow in this section are taken.

Fig. 8 illustrates the fields in the 12-byte Base Transport Header (BTH), which is used by all transport services. For purposes of our discussion, the important fields are a 1-byte Operation Code (OpCode), a 3-byte Destination QP number (Dest QP), a 3-byte Packet Sequence Number (PSN), and a 1-bit ACKnowledgement request flag (A).

The OpCode depends upon the type of transport service. For the RC and XRC services, commonly used OpCodes identify SEND, ACKnowledgement, RDMA WRITE, RDMA READ Request, and RDMA READ Response. As an example, complete packet formats are shown for the last two types of packets in Fig. 9. The first, middle and last payloads of a segmented message are distinguished in the OpCode. The Destination QP number identifies the QP which receives this packet in the destination HCA. The PSN is generated by all transport services, but only the reliable services are required to utilize it. The A bit is set only in reliable services in order to request the receiving QP to send back an ACKnowledgement message. The use of the F and B bits is described in our discussion of packet injection rate adjustment below.

In addition to the BTH, there are other transport headers, of which we review two important headers, the 4-byte ACK Extended Transport Header (AETH), shown in Fig. 10, which is used in reliable connection service, and the RDMA Extended Transport Header (RETH), shown in Fig. 11.

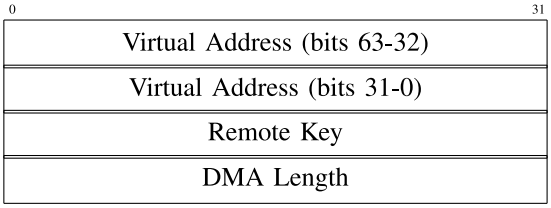


Fig. 11. RDMA Extended Transport Header (RETH) format [4]. Each line is 32 bits wide.

The AETH is included in the ACKnowledgement (ACK) packet and in the Read Response packet shown in Fig. 9. The AETH has two fields, the 1-byte Syndrome and the 3-byte Message Sequence Number (MSN). The Syndrome is encoded to indicate if the packet is an ACK or a Negative ACKnowledgement (NACK). If it is an ACK associated with RC service, the Syndrome also encodes the end-to-end flow-control credits from the receiving HCA for the QP associated with the connection. If it is a NACK, the Syndrome encodes an error code. The MSN is associated with a Send Sequence Number (SSN), which is set by the sender for each work queue element (i.e., message that the application passes via the verbs API for transmission to a remote destination). The first work queue element has an SSN of 0. The destination HCA sets the MSN to 0 for the first message, and subsequently keeps increasing MSN for each received message. It appears that SSN is not transmitted in any header, but MSN is returned in the AETH (which is included in the Acknowledgment packet or in the first and last packets of a RDMA READ Response message). The MSN is used at the sender to complete work queue elements and set events in a Completion Queue (CQ) (see Section III).

The RDMA Extended Transport Header (RETH) (Fig. 11) contains the Virtual address, Remote Key, and DMA Length fields. The RETH is present in the first packet of a RDMA WRITE First, RDMA WRITE Only, and RDMA READ Request packet because the first packet needs the virtual address of the starting point in the memory allocated at the remote end for the WRITE or READ operation. The Remote Key is required to ensure that the remote end has access rights to write or read into its memory location. The DMA Length field carries the total number of data bytes in the message.

Connection Management: Unlike in TCP, where connection management is part of the TCP specification, in InfiniBand, communication management is defined as part of the management system (along with subnet management, performance management, device management, etc.). Each management message is sent in a Management Datagram (MAD), which uses the SEND OpCode on the UD service offered by IB's transport layer.

Each end of an RC service requires a Connection Manager (CM) to exchange MADs in order to create and associate QPs at the two ends. For connection establishment, a sample sequence consists of a REQuest (REQ) from the client to the server, a REPly (REP) from the server to the client (to indicate that the request was accepted), and a Ready To Use (RTU) from the client to the server. This appears to be much

like the three-way TCP SYN, SYN+ACK, ACK sequence in TCP connection establishment. The REQ message carries the initial Packet Sequence Number as in TCP, and the local QP number (at the client) to use for the data communication. The REP message carries the local QP number at the server for the data connection. Since REQ and REP are carried in MAD messages, which run over UD service, the Destination QP carried in the BTH of the MAD messages is that of the remote connection manager. The Datagram Extended Transport Header (DETH), which follows the BTH in UD service, carries the Source QP of the connection manager. Thus, unlike in TCP, where the SYN, SYN + ACK and ACK connection establishment segments are carried in-band, i.e., on the same port numbers as the data connection, here the communication management messages REQ, REP and RTU are carried out-of-band, requiring the specification of the QP numbers to use for the data connection within the payload of the management messages.

A release procedure similarly uses Disconnect REQ and Disconnect REP MADs for orderly release. Stale connections are released with a time-out procedure much in the same way as handled in TCP.

End-to-End Error Control: At the transport-layer, error detection for a reliable service connection is done in the sending Host Channel Adapter (HCA) by setting the Packet Sequence Number (PSN) and ACK Request bit in the BTH of each packet sent. The receiving HCA reflects back the PSN in the AETH of a ACK or NACK message and a RDMA READ Response packet. If the AETH indicates an error detected by the receiving HCA, the Syndrome field encodes a 5-bit timer value that indicates the interval before the sending HCA can resend the missing or erroneous packet on that connection. However, for each connection the HCA maintains an encoded 3-bit retry counter on the sending side, where a value of 7 means an infinite number of retries are possible. If less than 7, this counter is decremented when each retry is sent, and after the counter reaches 0, no further retries are attempted and the sending connection enters the error state, preventing any further use of it.

Every reliable service connection maintains a Transport Timer to time outstanding requests, since these may get lost in transit and therefore not generate a NACK from the other side. But unlike in TCP, it is a static timer, not a per-segment timer. It is static because the value is derived according to a specified formula [4] from a "Local ACK Timeout" value whose minimum is defined by the vendor of each HCA but may be configured to a larger value when the connection is created. It is also not a per-packet timer; instead a connection's derived timer is reset each time it sends a packet with the ACK Request bit set in the BTH, when it receives a NACK, and when it receives an ACK at a time it is awaiting responses for other packets sent with sequence numbers higher than that reflected back in the AETH. Hence a connection's timer is in effect used to detect the absence of responses from the other end of the connection, not necessarily the loss of a particular packet. Therefore, selective retransmission is not supported as the timer measures the smaller of two values: the time since sending a packet with its ACK Request bit set, and the time since the last valid ACK packet arrived. This allows a Transport Timer timeout to detect when a specific packet is lost even though the timer is not per-packet. All packets sent after the missing packet are therefore also retransmitted. Receiving HCAs have intelligence to silently drop duplicate packets.

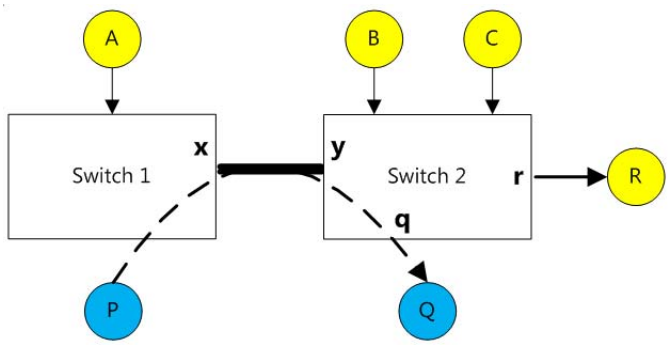


Fig. 12. An example congestion scenario.

Congestion Control: InfiniBand congestion control, as detailed in [4], [77], and [78], consists of optional transport-layer functionality for adjusting the packet injection rate at an HCA port, and optional switch output port support for detecting congestion and marking transport header bits to indicate congestion for processing by the receiving HCA port. This mechanism and its properties have been extensively studied in [79]–[83].

In Ethernet, a congested port at a switch or NIC will simply drop packets. TCP detects the dropped packets and lowers the window size accordingly [84]. But as already described above in Section II-D, InfiniBand link-layer flow-control is designed to prevent packet loss. This causes congestion to spread from switch to switch [85]. Congestion can cause real world variability in application runtime, as has been observed on a proprietary Cray XE6 system [86], and also on InfiniBand networks [77].

The next example, based on Fig. 12, illustrates congestion spreading in a simple InfiniBand network. For simplicity, we assume that all links transmit with equal bandwidth, that all sender HCAs send the same sized packets, and that *receiver HCA R* is always ready to receive those packets at full link bandwidth.

1) If we assume that *HCA B* on *switch 2* is trying to send data to *HCA R* at its full link bandwidth, and that no other HCAs are sending data, then there is no congestion at *port r* and a cut-through switch design could forward packets from *HCA B* through to *HCA R* at the full link rate.
2) If both *HCA B* and *HCA C* on *switch 2* try to send data to *HCA R* at their full link bandwidth, then there is twice as much total traffic heading to *HCA R* as the link from *port r* to *HCA R* can handle. Therefore *port r* is congested, even though *port r* always has credits from *HCA R* to send data at full link bandwidth. If *port r* uses fair arbitration, the bandwidth to *HCA R* will be equally shared by both *HCA B* and *HCA C* (i.e., 50% each).

3) If in addition to *HCA B* and *HCA C* on *switch 2* trying to send data at their full link rate, *HCA A* on *switch 1* tries to send data at $\frac{1}{2}$ of its full link bandwidth via the *x-to-y link* to *HCA R* on *switch 2*, fair arbitration on *port r* will allow each of *HCA B*, *HCA C*, and *port y* to forward data at $\frac{1}{3}$ of the link bandwidth. Initially *port x* will continue to forward data from *HCA A* to *port y* at the $\frac{1}{2}$ rate offered by *HCA A*, but because *port y* can only forward to *port r* at the smaller $\frac{1}{3}$ rate arbitrated by *port r*, the input buffer at *port y* will quickly fill up. This will cause *port y* to slow the rate it issues credits to *port x*. This in turn causes *port x* to reduce the rate it issues credits to *HCA A*, causing the data rate experienced by *HCA A* to be reduced to $\frac{1}{3}$ of link bandwidth, the same as that experienced by *HCA B* and *HCA C*.
4) If both *HCA A* and *HCA P* on *switch 1* each tries sending data at $\frac{1}{2}$ of its full link bandwidth via *port x* on *switch 1* through *port y* on *switch 2*, fair arbitration at *port x* will share the available bandwidth of the *x-to-y link* equally between *HCA A* and *HCA P*, so each will have credits to send at only $\frac{1}{6}$ of the full link bandwidth. *Node P*'s transmission rate will be reduced, even though it is not contributing traffic to the congestion at *port r*, and *port q*, the destination of the flow from *HCA P* to *HCA Q*, is free. This phenomenon is known as Head of Line (HoL) blocking [87]. Note that if the input-side buffers were implemented to use Virtual Output Queues (VOQs) [87] (i.e., to maintain one virtual queue corresponding to each output port), this would eliminate HoL blocking of packets from *HCA P* to *HCA Q* at *port y*, but would only restore the sending rates of both *HCA A* and *HCA P* to $\frac{1}{3}$, because *port y* will be able to return credits to *port x* twice as fast, but since those credits are allocated fairly by *port x* between *HCA A* and *HCA P*, they both continue to share the link equally.

Congestion Detection: In an InfiniBand subnet, congestion is a situation in which the amount of traffic offered on a VL in a switch output port exceeds the rate at which those packets can be forwarded over that VL. It can be triggered when either:

1) multiple active traffic flows from input ports converge onto a single output port VL;
2) the receiving end of the output link stops sending flow control credits back to the switch output port.

When one of these conditions occurs, the number of packets queued for sending on a switch output port VL increases, and if that number exceeds a threshold configured for each port and VL by the fabric administrator [80], that port VL enters the "congested state" [4].

Packet Injection Rate Adjustment: As illustrated in Fig. 13 (the same setup as Fig. 12), congestion control consists of a series of three actions [81]:

① A switch's VL in the congested state can be configured to periodically (as determined by a configured "marking rate") set the Forward Explicit Congestion Notification (FECN) bit in the BTH for packets of the VL experiencing congestion. Normally this marking would be configured for all switch output ports except for a switch output port connected to another switch port when its threshold is exceeded due to condition 2). The reason for this exception is illustrated by *port x* on *switch 1* in Fig. 13. This port stops receiving flow control credits from *port y* on *switch 2* due to the congestion at *port r* on *switch 2* caused by condition 1). Both *HCA A* and *HCA P* are sending traffic through *port x*, but only traffic from *A* contributes to the congestion at *port r*. Therefore, *port x* should not set the FECN bit in any outgoing packets – the packets from *HCA A* will be marked at *port r*, so *A*'s injection rate will be reduced, but packets from *HCA P* will not be marked, so its injection rate will not be reduced.

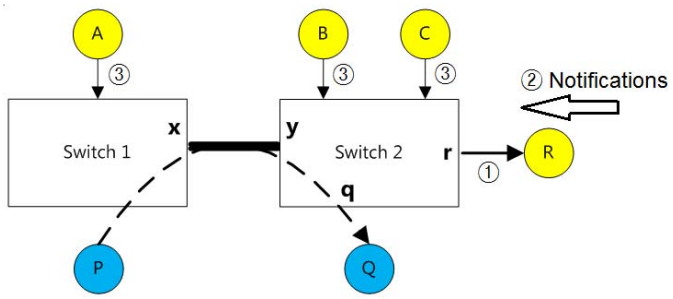


Fig. 13. Congestion notification steps.

② When *HCA R* receives a packet with the FECN bit set, it notifies the sender of the FECN-marked packet either by setting the Backward Explicit Congestion Notification (BECN) bit in the BTH of the next outgoing ACK packet back to that sender, or by sending an explicit Congestion Notification Packet (CNP) with its BECN bit set.

③ When *HCA A, B, or C* receives a packet with the BECN bit set, it must decrease the rate at which it injects packets into the subnet. The mechanism to do this can be configured at each HCA port to apply on either a per-QP or per-SL basis. The per-QP basis manages the sending rate of only the specific source QP that is contributing to the congestion, whereas the per-SL basis could lead to unfairness since the congested QP flow may impact other flows using the same SL.

The components in this HCA mechanism consist of the following: each port configures a common Congestion Control Table (CCT) to hold Injection Rate Delay (IRD) values (ordered so that the smallest IRD value is in the first table entry, the largest value in the last). For each QP/SL flow the port configures a set of 3 items: a Congestion Control Table Index (CCTI) into the port's common CCT (initialized to a configured non-negative value called CCTI_Min), a configured CCTI_Increase value that is the amount the CCTI will be increased, and a CCTI_Timer to control when the CCTI will be decreased (initialized to a configured timeout value).

Each time it receives a packet with the BECN bit set, this HCA port selects the appropriate QP/SL flow structures, and increments CCTI by CCTI_Increase (but not beyond the end of the port's CCT) so it will point at a larger IRD value in the CCTI. When this port's output link arbitrator selects the next packet from this QP/SL flow to send, the port will delay sending it by the IRD value from the CCT indexed by CCTI.

When the number of packets queued up to send on a congested switch output port (*switch 2*'s *output port r* in Fig. 13), drops below a configured threshold, that port leaves the congestion state and stops marking the FECN bit in outgoing packets. Therefore the destination *HCA R* of those packets will no longer see the FECN bit set, so it will no longer set the BECN bit in packets it sends back to the source HCAs of those packets. All this time the periodic CCTI_Timers on each *source HCA A, B, or C* port have been running independently, and each time one of these timers expires, it is reinitialized to its configured timeout value, and if its associated CCTI is greater than CCTI_Min, that CCTI is decremented by 1, thereby reducing the IRD value used by subsequent packets sent on that QP/SL flow.

Other Approaches to Congestion Control: Studies have shown that VLs (discussed previously at the end of Section II-D) could offer an alternative way to improve this situation [88]–[90]. For example, Guay *et al.* [89] propose a solution that uses a VL-based scheme as an alternative to the congestion control approach of having senders lower their injection rates.

Reference [91] proposes a new dynamic congestion management system that could read InfiniBand's "PortXmitWait" counters (which record the number of clock ticks during which a port had insufficient credits to transmit) of switch ports connected to HCAs, and use this information to dynamically move flows (such as *A-to-R* in Fig. 13) that contribute to hot spots to a separate Slow Lane VL, while directing non-contributing flows (such as *P-to-Q*) to a Fast Lane VL.

Several simulation studies [92]–[94] propose some modifications to the existing InfiniBand congestion mechanism that could significantly improve congestion detection and recovery, as well as simplify the task of configuring the current multitude of parameters, by taking a more dynamic approach to detecting and responding to congestion.

G. Summary

In this section, we discussed the general architecture of InfiniBand as well as aspects of its link and transport layers. We discussed the credit-based flow control mechanism, which allows InfiniBand fabrics to be lossless but has implications further up the stack, in particular for transport layer congestion control. We discussed error control at the link and transport layers. We discussed the Subnet Manager, its role in populating forwarding tables, and common routing algorithms used in InfiniBand. We discuss the transport protocol and its headers, including the types of operations which are used in applications via the verbs API.

III. Verbs API

The sockets API has been the traditional high-level interface to networking since it was introduced in 1983 as part of the Berkeley Software Distribution (BSD) of the Unix operating system. It has evolved since then, and forms the basis for the POSIX Sockets API Standard [95].

The verbs API is an interface introduced by the OpenIB Alliance in 2005 as the OpenFabrics Enterprise Distribution (OFED) [96] interface to InfiniBand technology. In 2006 the OpenIB Alliance changed its name to OpenFabrics Alliance (OFA), and generalized OFED to include all standards-based Remote Direct Memory Access (RDMA) technologies: InfiniBand (IB), Internet Wide Area RDMA Protocol (iWARP) [97]–[99], and RDMA over Converged Ethernet (RoCE) [100]. Fig. 1 showed how the OFA Verbs API is layered between IB and MPI. Recent versions of Microsoft Windows provide an alternative API called Network Direct [101] for IB as well as iWARP and RoCE. There are OFED interfaces implemented in both kernel space and user space; however, this tutorial will focus exclusively on the user space verbs API, as this is of most interest to user space MPI libraries and HPC applications which often need to be portable across multiple kernels.

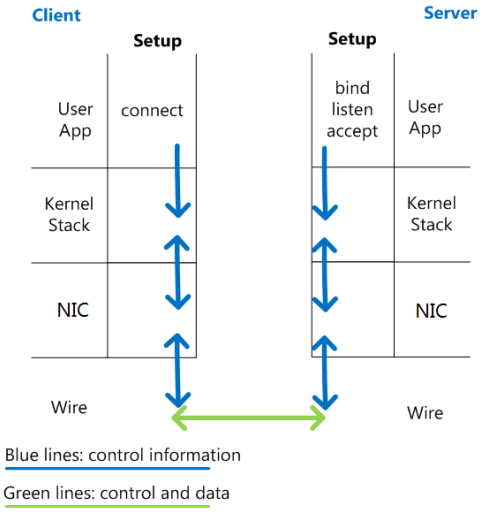


Fig. 14. TCP/IP connection setup.

This section is organized into the following subsections: Section III-A comparison of sockets and verbs; Section III-B introduction to verbs; Section III-C verbs for "channel semantics" (that type of application-level I/O most similar to sockets); Section III-D completion handling (a concept unknown with sockets); Section III-E memory registration (a new concept introduced with verbs); Section III-F verbs for "memory semantics" (a type of application-level I/O unknown with sockets); Section III-G practical experience using verbs.

A. Comparison of Sockets and Verbs

Some of the similarities between Sockets and Verbs:

1) Both are able to utilize IPv4 and IPv6 addresses.
2) Both provide user-level *send* and *recv* functionality.
3) Both provide full-duplex operation.
4) Both provide a reliable and an unreliable transport.
5) Both utilize the client-server model for connection establishment in reliable mode.

We illustrate this similarity by comparing the connection establishment operations for TCP/IP in Fig. 14 with the corresponding operations for verbs in Fig. 15.

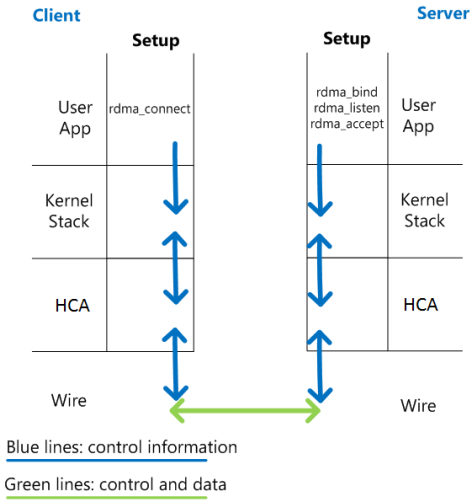


Fig. 15. InfiniBand connection setup.

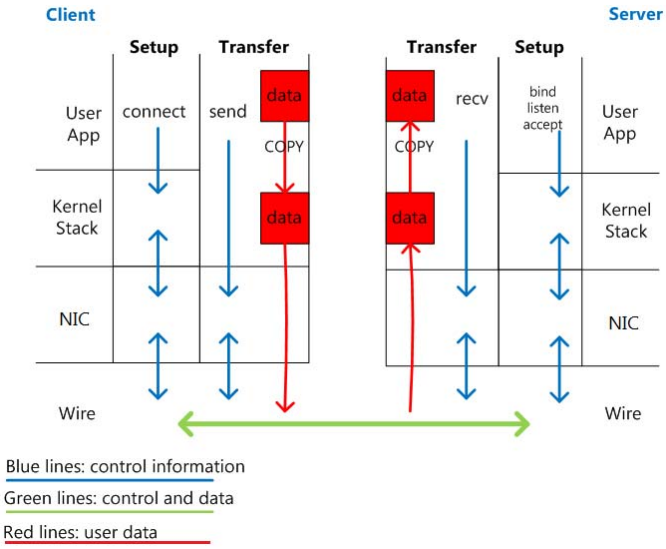


Fig. 16. TCP/IP data transfer.

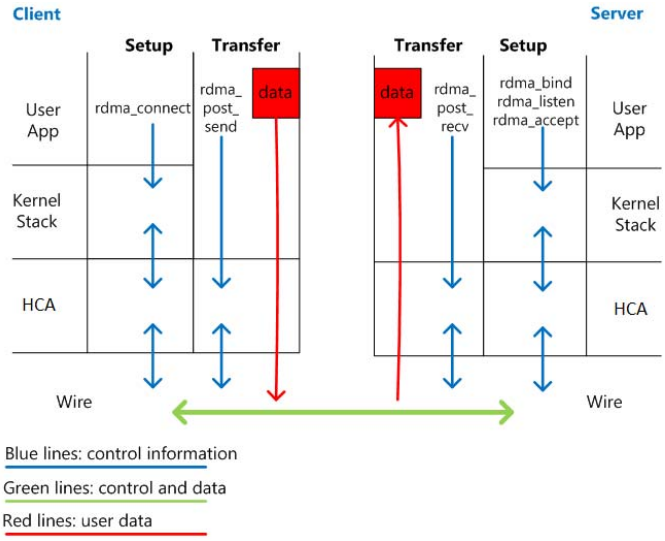


Fig. 17. InfiniBand data transfer.

Some of the differences between Sockets and Verbs:

1) Sockets provide a stream of bytes in reliable connection mode (TCP), and unreliable messages in unreliable datagram mode (UDP). Verbs provide only messages, no streams, in both Reliable Connection (RC) and Unreliable Datagram (UD) modes.
2) Sockets rely upon software buffering in the end nodes. On a *send* operation, the kernel copies data from user-space into a kernel-space buffer from which it is actually transferred over the wire into another kernel-space buffer on the receiving side, where a *recv* operation copies data from that buffer into user-space. Verbs do not use buffering in the end nodes – on a *send* operation the hardware Host Channel Adapter (HCA) transfers data directly from user-space over the wire into the user-space provided by the *recv* operation on the receiving side. This is called "zero copy" [16].
3) Sockets require kernel intervention during data transfers. Verbs do not, as the user program deals directly with the HCA. This is called "kernel bypass", which together with "zero copy" is illustrated by comparing a TCP transfer in Fig. 16 with a corresponding verbs transfer in Fig. 17.
4) Sockets can utilize any user-space memory for data transfers. Verbs can also utilize any user-space memory for data transfers, but they require that the application registers this memory prior to initiating the transfer.
5) Sockets operate synchronously, whereas verbs operate asynchronously. Normally a socket *recv* operation blocks until data appears in the receive-side's kernel buffer and can be copied into the user-space buffer, but if the application puts the socket into non-blocking mode using the O_NONBLOCK flag, the *recv* operation simply fails with an EAGAIN or EWOULDBLOCK error without copying any data, so that the application must subsequently reissue the same *recv* operation. Similarly, a normal socket *send* operation blocks until sufficient space is available in the sending-side's kernel buffer to hold the entire user message, but for a socket in non-blocking mode, the *send* operation simply fails with an EAGAIN or EWOULDBLOCK error without copying any data, so that the application must subsequently reissue the same *send* operation. However, as described more fully in Section III-C below, when using verbs a thread simply enqueues the equivalent of a *send* or a *recv* operation with the local HCA and gets control back immediately. A thread, not necessarily the one enqueuing the operation, must subsequently invoke another verb to determine when an enqueued transfer has completed. This difference enables threads using verbs, but not sockets, to enqueue multiple outstanding operations on the same connection and to proceed in parallel with the HCA data transfers, all without any kernel intervention.

B. Introduction to Verbs

The InfiniBand Architecture Specifications [4] introduce the term *verbs* as follows.

> "IBA (The InfiniBand Architecture) describes a service interface between a Host Channel Adapter

(HCA) and a consumer of the I/O services provided by that HCA. The required behavior exposed to the consumer is described by a set of semantics called *Verbs*. Verbs describe operations that take place between an HCA and the consumer based on a particular queuing model for submitting Work Requests (WRs) to the HCA and returning Completion Status (CS).

The intent of Verbs is not to specify an API, but rather to describe the interface sufficiently to permit an API to be defined that allows a consumer of I/O services to take full advantage of the architecture.

Verbs describe the parameters necessary for configuring and managing the HCA, allocating (creating and destroying) Queue Pairs (QPs), configuring QP operation, posting WRs to the QP, and getting CS from the Completion Queue (CQ)."

Put simply, the IBA standard purposely does not define a syntax for verbs that programmers can use when writing a program, since to do so would require the standard to conform to the syntactic conventions of a particular operating system and programming language. The OpenFabrics Alliance does define a "C" language syntax for these verbs, and it also provides the OpenFabrics Enterprise Distribution (OFED) software implementation of this syntax for the Linux operating system. In the OFED API [96], a syntactic *verb* is synonymous with a syntactic *function* as that term is known to and understood by programmers.

OFED verbs do not operate in a vacuum – the OFED software also defines a large number of *data structures* on which verbs operate. Since verbs interact directly with the HCA, these data structures will closely mirror structures within the HCA. However OFED is careful to make clear that the syntax defined by these data structures is not necessarily identical to the underlying *hardware structures* in the HCA, since different RDMA technologies (IB, iWARP, RoCE) and different vendors within each technology are free to define their own internal hardware structures to best fit with their implementation.

A key point in the quote about IBA given above is that the interface described by verbs is based on a queuing model for both submitting Work Requests (WRs) to the HCA and returning Completion Status (CS) from the HCA. Because queues are an important design feature of InfiniBand, they will also be an important data structure type in the OFED API, and a large number of verbs will deal with creating, destroying, and using queues. We limit our discussion to the queues and their associated verbs dealing with data transfers. Fig. 19 shows the `rdma_post_recv` and `ibv_poll_cq` verbs with the queues utilized in InfiniBand to receive data over a connection, and should be compared with the corresponding TCP *recv* operation in Fig. 18.

We further limit our discussion to verbs relating to data transfer operations for the Reliable Connection (RC) transport service. We begin by introducing in Section III-C those verbs whose functionality most closely resembles that of corresponding functions in TCP. These verbs implement the

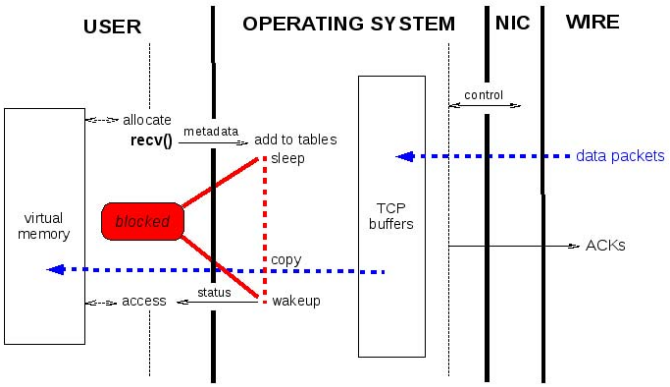


Fig. 18. TCP recv (time increases from top to bottom).

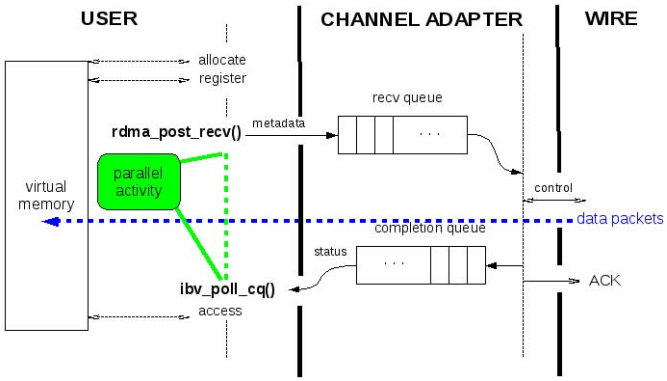


Fig. 19. InfiniBand recv (time increases from top to bottom).

so-called "channel semantics" defined in Section 3.6 of the IBA Specifications [4] as "the communication style used in a classic I/O channel". In Section III-F we discuss those verbs that implement "memory semantics", which have no counterpart in TCP.

C. Verbs for Channel Semantics

The term "work request" (WR) used in the quotation about IBA given above in Section III-B is a generic term describing the data structure which is posted on a Work Queue (WQ) in order to initiate a transfer of a single message. The information contained in a WR data structure includes the requested transfer operation plus "metadata" that describes the message data to be transferred. This includes the virtual address where the message data is located in memory, the number of bytes of data in the message, and the memory registration information for that memory region. Message data itself is not included in the WR, since the Host Channel Adapter (HCA) will transfer the data across the wire directly to or from its location in user virtual memory. Note that due to different options that a user may apply, the WR data structure submitted to a SQ is called a Send Work Request (SWR) and has a somewhat different internal structure than the WR data structure submitted to a RQ, which is called a Receive Work Request (RWR).

Data transfers in InfiniBand are asynchronous, which means the HCA transfers data into or out of user memory simultaneously with the thread that initiated the transfer performing

independent operations on other parts of its memory. This asynchronous mode of operation applies to both the sender and receiver of the data. Therefore, on each side of a data transfer there are two verbs that a user must issue. The first verb needs to *post* the WR onto the end of a WQ in the HCA. When this WR gets to the head of that queue, the HCA will act upon it. The second verb needs to *poll* a *Completion Queue* (CQ) in the HCA in order to obtain from it the *Work Completion* (WC) describing the result of the transfer (i.e., the success or failure status of the transfer, along with any additional information such as the number of bytes actually transferred or the error code). Between the posting of a WR and the successful polling of a completion, the application must consider the transfer to be "in progress", which implies that the application must not change the memory involved in the transfer.

Sending Data: There are 5 phases in sending one message, and we illustrate these 5 phases by looking at how an application actually sends data.

1) An application starts to send data by calling the `rdma_post_send` verb that *posts* a *Send Work Request* (SWR) data structure onto a *Send Queue* (SQ) data structure in the HCA. Once this SWR has been placed at the end of the SQ, the `rdma_post_send` verb is finished and the application continues with the next instruction following the verb invocation. However, the data transfer is not finished – control of the transfer has simply been handed over to the HCA, so the user program must not change the data area involved in the transfer until it has obtained a *Work Completion* (WC) for this transfer from the HCA.
2) The SWR will remain on the SQ until it moves to the front of that queue, at which point the HCA uses it to start transferring data from user memory directly onto the wire.
3) While the HCA moves data out of user memory directly onto the wire, it transparently segments it as necessary into physical Maximum Transmission Unit (MTU) sized packets that contain appropriate SEND OpCode values in the Base Transport Header (BTH) for each packet, as described in Section II-F. Note that during the transfer, the HCA on the receiving end of the wire will be moving that data off the wire directly into the receiving side's user memory as it arrives. All WRs involve only a single user-level message – the sending-side HCA transparently segments that message into physical Maximum Transmission Unit (MTU) sized packets, and the receiving-side HCA reassembles those packets into a single message.
4) When the sending side HCA finishes sending the last packet of a message, it will receive back from the receiving side HCA a transport-level ACKnowledgement (ACK) packet containing status information about the transfer. (Since the connection is full duplex, it can also start to process the next SWR (if any) from the SQ.) The sending side HCA incorporates this information into a *Work Completion* (WC) data structure and enqueues it onto the end of its local *Completion Queue* (CQ) associated with this SQ. This WC contains a user-defined identification of which SWR initiated the transfer and a code to indicate the type of operation that completed. There is also an optional feature that enables the HCA to notify the program asynchronously when this WC has been placed on the CQ.
5) In order to remove the WC from the HCA's CQ, the application invokes the `ibv_poll_cq` verb, which will place the WC into a user-designated area of memory for access by the user program. At this point the user-level data transfer is complete, and the application program is free to change the contents of its memory area involved in the completed transfer.

Receiving Data: When using channel semantics, each user-level `rdma_post_send` verb on one end of the wire must be matched by a corresponding user-level `rdma_post_recv` verb on the other end. As when sending a message, there are also 5 phases in receiving one message.

1) An application starts to receive data by calling the `rdma_post_recv` verb that *posts* a *Receive Work Request* (RWR) data structure onto a *Receive Queue* (RQ) data structure in the HCA, as illustrated in Fig. 19. Once this RWR has been placed at the end of the RQ, the `rdma_post_recv` verb is finished and the application continues with the next instruction following the verb invocation. However, the data transfer is not finished – control of the transfer has simply been handed over to the HCA, so the user program must not reference the data area involved in the transfer until it has obtained a *Work Completion* (WC) for this transfer from the HCA.
2) The RWR will remain on the RQ until it moves to the front of that queue, at which point the HCA uses it to start to receive data packets from a sender at the other end of the wire.
3) As the packets arrive, the HCA moves their data directly off the wire into the user memory described by the RWR at the front of its RQ, removing packet headers, checking trailing CRCs, and reassembling these MTU-sized packets into a single message, as required by the IBA wire protocol.
4) When the receiving side HCA finishes receiving the last packet of a message, it will send back to the sending side HCA a transport-level ACK packet containing status information about the transfer. The receiving side HCA also incorporates this information into a *Work Completion* (WC) data structure and enqueues it onto the end of its local *Completion Queue* (CQ) associated with the RQ. This WC contains the total number of bytes actually received, a user-defined identification of which RWR initiated the transfer, and a code to indicate the type of operation that completed. There is also an optional feature that enables the HCA to notify the program asynchronously when it places this WC on the CQ.
5) In order to remove the WC from the HCA's CQ, the application invokes the `ibv_poll_cq` verb, which will place the WC into a user-designated area of memory for access by the user program. At this point the user-level

data transfer is complete, and the application program is free to reference the contents of its memory area involved in the completed transfer.

So far we have described 3 verbs: `rdma_post_send`, `rdma_post_recv`, and `ibv_poll_cq`, and 3 queue structures: the Send Queue (SQ), the Receive Queue (RQ), and the Completion Queue (CQ). Because every `rdma_post_send` verb on one end of the wire must be paired with an `rdma_post_recv` verb on the other end, the SQ and RQ on the same side are represented by a single Queue Pair (QP) data structure, `struct ibv_qp`. Furthermore, the user can choose to have one CQ shared by both the SQ and the RQ, or two separate CQs, one for each WQ. The data structure for a CQ is `struct ibv_cq`.

There are verbs to dynamically create and destroy these objects: `ibv_create_qp` and `ibv_destroy_qp` for QPs; `ibv_create_cq` and `ibv_destroy_cq` for CQs. The verbs to create a queue enable the user to specify its size.

D. Completion Handling

The `ibv_poll_cq` verb removes Work Completions (WCs) from a Completion Queue (CQ), but if that CQ is empty, `ibv_poll_cq` simply returns a value of 0 to indicate it did not find any WCs to remove. In this situation, the program can perform other work and try again later, but if it's logic requires getting a WC before it can go on to anything else then it has two approaches: "busy-polling" or "wait-wakeup".

Busy-Polling: This is the simplest approach, since it just requires a loop that repeatedly calls `ibv_poll_cq` until it actually finds and returns a WC from the CQ. This approach gives the best response, and hence the lowest latency, because the program will detect a WC as soon as the Host Channel Adapter (HCA) adds it to the CQ. However, it is obviously expensive in terms of CPU cycles, although they are all user-space cycles because the kernel is not involved.

Wait-Wakeup: This approach avoids the high cost in user-space CPU cycles, but it adds kernel-space CPU cycles if the scheduler has to deal with putting a thread to sleep and awakening it again, and this in turn increases the latency. This requires three new verbs:

1) The `ibv_req_notify_cq` verb "arms" (i.e., activates) the notification mechanism, such that when the HCA adds a new WC to the indicated CQ, it will also "wakeup" any thread waiting for this by generating a "cq_event".
2) The `ibv_get_cq_event` verb puts the calling thread to sleep until a "cq_event" has been generated by the HCA.
3) The `ibv_ack_cq_events` verb must be called to acknowledge every "cq_event" returned by `ibv_get_cq_event`.

There are several ways these verbs can be used in conjunction with `ibv_poll_cq`, but it is important to note that none of these verbs are influenced in any way by the current contents of the CQ, nor do any of them change the contents of the CQ. In addition, arming the event notification is a "one-shot" operation: when the HCA generates a cq_event, it also "disarms" (i.e., deactivates) the notification mechanism. This can cause "race conditions" when trying to integrate calls to `ibv_poll_cq` with these wait-wakeup verbs. To deal with these races, the following pseudocode sequence shows a common way to block waiting for the next WC to arrive:

1) Call `ibv_poll_cq` and if it finds a WC then by-pass the following loop
2) loop
3) Call `ibv_req_notify_cq` to arm cq_event generation by the HCA
4) Call `ibv_poll_cq` and exit loop if it finds and returns a WC
5) Call `ibv_get_cq_event` to put the thread to sleep until the HCA generates a cq_event
6) Call `ibv_ack_cq_events`
7) Call `ibv_poll_cq` and exit loop if it finds and returns a WC
8) end loop

Notice that there are 3 separate calls to `ibv_poll_cq` in this sequence.

The call in step 1) catches any WC put into the CQ by the HCA since the previous execution of this sequence. The loop in step 2) is entered only if this call in step 1) finds no WC in the CQ.

The call in step 4) catches any WC put into the CQ by the HCA while in the loop but before the notification mechanism has been armed by the call to `ibv_req_notify_cq` in step 3). This can happen on the first iteration of the loop if that WC was put into the CQ immediately after the call to `ibv_poll_cq` in step 1) failed to find a WC in the CQ, and it can happen on a subsequent repeat of the loop if a "false wakeup" occurs in step 5), as explained below.

The call to `ibv_poll_cq` in step 7) will normally pick up the WC that terminated any wait initiated by `ibv_get_cq_event` in step 5).

Finally, the end loop in step 8) can only be reached if the `ibv_poll_cq` in step 7) fails to find a WC in the CQ, a situation triggered by a "false wakeup" in step 5). A false wakeup can only happen if, in a previous execution of this sequence, the HCA put a WC into the CQ during the time between the call to `ibv_req_notify_cq` in step 3) and the call to `ibv_poll_cq` in step 4), which therefore caused the call in step 4) to dequeue that WC from the CQ and exit that previous sequence without calling `ibv_get_cq_event` in step 5). This can occur only during a small but finite time-window, so normally it will happen only rarely, but since it is possible, the application must handle this case. When it does occur, that WC will have caused the HCA to generate a cq_event and disable the notification mechanism, so that in the subsequent execution of this sequence when execution reaches the call to `ibv_get_cq_event` in step 5), it will not wait but will return immediately due to that cq_event "left over" from the prior sequence.

E. Memory Registration

One of the major differences between sockets and OFED verbs is the need to *register* all memory involved in an

RDMA transfer. There are two OFED verbs that deal with this: `ibv_reg_mr` which creates a new `struct ibv_mr` that contains registration information, including access rights, for a user-defined memory region, and `ibv_dereg_mr` which destroys an existing `struct ibv_mr`. Memory registration is crucially important to the performance of RDMA for the following reasons.

1) `ibv_reg_mr` causes the operating system to "pin" the physical memory pages holding the registered user-space virtual memory into physical memory so they cannot be swapped out [102]. This is necessary because data transfers involving this memory are performed directly by the HCA without operating system involvement and therefore without operating system awareness of when these transfers occur. If the OS were to swap out a page during an RDMA transfer, severe data corruption could obviously result. The "pin" is released by the `ibv_dereg_mr` verb.
2) Registration enables hardware Host Channel Adapters (HCAs) to store copies of the virtual-to-physical mapping for all memory regions so that during subsequent transfers the HCAs on both sides of the transfer can map virtual addresses contained in the packet headers into physical addresses needed by their respective memory buses without kernel intervention [102], [103].
3) Because the HCA stores the virtual-to-physical mappings of all registered memory, it can dynamically check the validity of the virtual addresses contained in the packet headers.
4) Registration enables the HCA to create "keys" in the `struct ibv_mr` that link the registered memory with the process that performs the registration. The user must supply these keys as part of every transfer, which in turn allows the HCA to dynamically find the correct mapping to apply to that transfer and to check that the registered access rights allow the type of transfer being performed.

F. Verbs for Memory Semantics

The `rdma_post_send` and `rdma_post_recv` verbs are analogous to similar send and recv functions in TCP because the user-level programs on both sides in a data transfer must actively participate in the transfer: the sending side user must explicitly send the data, and the receiving side user must explicitly receive it. This method of transferring data, known as "channel semantics" because of its similarity with the well-known channel model of I/O, is illustrated in Fig. 20,

Sockets have only a channel model of I/O, but RDMA has an additional "memory semantics" model in which the user on only one side of a transfer (the "active" side) performs an operation, while the user on the other "passive" side does nothing. Effectively the memory of the passive user becomes an extension of the memory of the active user, since the active user can read and/or write to that memory without the passive user being aware that anything is happening. Of course, the passive side has to grant permission for the active side to access any of its memory prior to any transfers to or from that memory. The passive side does this by transferring to the active side the "metadata" (i.e., starting virtual address, length, and registration key) for the passive side's contiguous memory area that it is allowing the active side to access. Once the active side has this metadata, it includes this in its subsequent Work Requests (WRs) that access the passive side's memory.

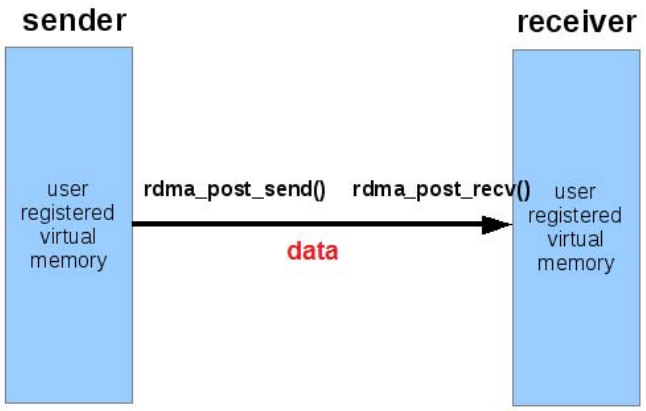


Fig. 20. RDMA "channel semantics".

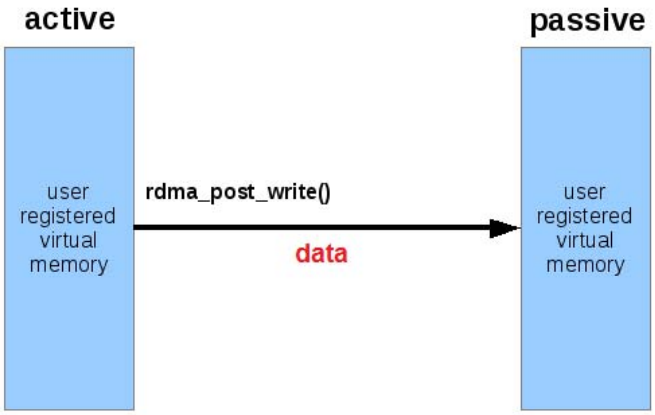


Fig. 21. RDMA WRITE "memory semantics".

There are two verbs that utilize memory semantics: `rdma_post_write` transfers data from the active user's memory into the passive user's memory, as illustrated in Fig. 21; `rdma_post_read` transfers data in the opposite direction, from the passive user's memory into the active user's memory, as illustrated in Fig. 22. These figures, which illustrate memory semantics, should be compared with Fig. 20, in order to clarify their differences with channel semantics. Note that **both** these verbs post a SWR to the SQ – `rdma_post_read` does **not** use the RQ.

As mentioned earlier, "memory semantics" transfers are performed entirely by the active side of the transfer, that is, the side that posts the SWR. This side must therefore know the location of the memory involved on **both** sides of the transfer, and must possess the proper memory registration keys for **both** sides of the transfer in order to include all this metadata information in the SWR. Recall that a "channel semantics" Send Work Request (SWR) or Receive Work Request (RWR) only needed to have the address, length, and memory

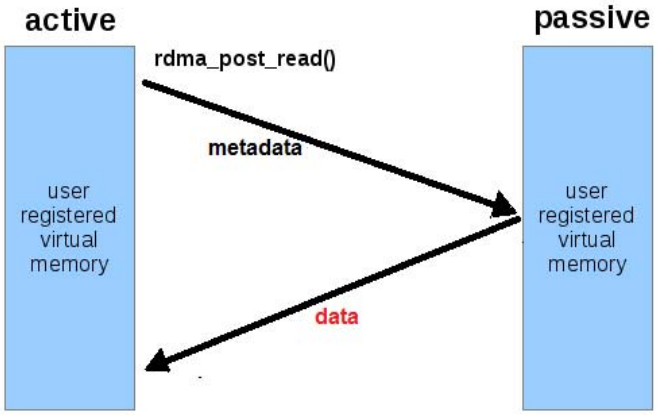


Fig. 22. RDMA READ "memory semantics".

registration key for the local memory involved in the transfer, that is, for the memory on the side posting the WR.

In "memory semantics" transfers, how does the active side obtain this metadata about the remote side of the transfer (i.e., the passive side)? The answer to this question is technically "outside the scope of the standard", which means that the programmer is responsible for doing this, and can do it anyway he/she sees fit. The most common technique is for the passive side to send this metadata to the active side prior to the time when the active side needs to use the passive side's memory. The application may use a "channel semantics" transfer to do this, since this can utilize the same connection as the "memory semantics" transfer. However, a separate connection of any type, including a TCP/IP connection, could be used instead. Once the active side obtains metadata about a remote memory area, it can utilize that same metadata in many different transfers – it does not have to obtain this information before each transfer if the same passive side memory area is reused. Application design is a major factor in deciding how and when to obtain this information.

Note that "memory semantics" as provided by InfiniBand differs from higher-level Remote Memory Access (RMA) abstractions provided by MPI and PGAS languages. In particular, InfiniBand exposes the virtual memory addresses of remote memory regions. Higher-level RMA abstractions, such as MPI-2 RMA [9], use higher-level structures, such as memory windows, which allow memory semantics transfers without exposing the programmer to the remote virtual memory addresses. Additionally, although InfiniBand does not specify any synchronization mechanism for accessing remote memory, MPI-2 memory windows provide explicit synchronization calls to enforce ordering between local access to the memory region by the passive side and remote memory operations by the active side. Finally, MPI-2 RMA allows remote operations on strided and other complex data types, whereas the passive side buffer for an InfiniBand memory semantics transfer operation must be contiguous [104].

G. Practical Experience Using Verbs

Verbs programming is much more complex than traditional sockets programming because it requires programmers to directly reference many structures and functions defined by the API. Consequently, there are numerous ways to utilize verbs in order to increase different performance metrics in certain circumstances, and a number of articles have been published that study the use and performance of verbs in general [105]–[107], as well as those that study their use by specific applications, such as MPI [37], [108], FTP [109], GridFTP [110], and NFS [111]. The choice of features to employ will be constrained by the needs of the application, such as emphasis on latency, throughput, or cpu cycles, and by any host resource limitations, such as memory available for registration, or options provided by the Host Channel Adapter (HCA).

One of the least anticipated results is that aligning the starting virtual address of each message on a multiple of 64-bytes can improve throughput over unaligned messages [107]. The IBA standard allows messages to be aligned on arbitrary boundaries, but its requirement that data be transferred in full 64-byte "flow control blocks" (padded if necessary) has the unstated side effect of forcing some implementations to fetch 2 64-byte blocks from memory for each 64-byte block placed on the wire, thus slowing down the effective data rate.

A more obvious method of improving throughput is to keep the wire busy, either by using larger messages or by sending multiple simultaneous messages or both, as shown in [107]. Those results indicate that taking advantage of the asynchronous nature of verbs permits the programmer to keep the wire busy by posting multiple and/or large messages, which can be particularly beneficial for throughput intensive applications such as file transfer.

When an application's emphasis is on latency, messages are usually much smaller than when the emphasis is on throughput, and as mentioned above in Section III-D, busy-polling can produce a significant improvement over wait-wakeup. In addition, an optional `IBV_SEND_INLINE` flag is available on the `rdma_post_send` and `rdma_post_write` verbs for use when sending "small" messages, where the HCA defines the limit on "small" messages, which may be 0. When this flag is present on a send operation, the HCA copies the message data into its internal buffers at the time the SWR is posted (without this flag no data is copied). This has several benefits for the application: the message data area does not need to be registered; the message data area becomes immediately available for reuse by the application without waiting for a Work Completion (WC); and the message latency may be reduced because when the SWR advances to the head of the HCA's Send Queue (SQ), the HCA can place the message data directly onto the wire without involving the PCIe bus or the host memory system.

A final option that can have a significant influence on performance is the ability of the programmer to post "unsignaled" Send Work Requests to the SQ. This option can be set at the time a queue pair is created, and is normally off, so that each SWR posted to the SQ will generate a WC to its associated CQ. When this option is set, a SWR posted to the SQ will not generate a WC unless the `IBV_SEND_SIGNALED` flag is provided in the call to `rdma_post_send` or `rdma_post_write`. This means that the application can

avoid the overhead required to process that WC, which may improve performance. For example, if a program wants to post N SWRs in succession to the same SQ, it may want to post the first $N-1$ unsignaled and supply the `IBV_SEND_SIGNALED` flag only on the last SWR in order to know when the entire sequence has completed. This works because the IBA standard requires that all requests posted to the same queue must complete in the same order as they were posted, which implies that when the signaled Nth SWR completes, the $N-1$ previous unsignaled SWRs must have also completed. One caveat: N is limited to the size of the SQ, which itself is a parameter that can be configured by the application when the SQ is created. This limitation exists because HCAs may keep unsignaled SWRs on the SQ until a WC is processed for a subsequent signaled SWR, and therefore an unlimited sequence of unsignaled SWRs would eventually overflow the SQ. Also note that all Receive Work Requests (RWRs) posted to the Receive Queue (RQ) always generate a WC and are therefore unaffected by this option.

IV. MPI

The Message Passing Interface (MPI) [9] is a *de facto* standard specification for a portable suite of functions that enable components of a parallel program to communicate at a high-level of abstraction from the underlying communication mechanisms provided by a host computer cluster. The MPI Forum [112], a group of vendors, developers, researchers, and users of MPI, released the first approved MPI-1.0 Specification in May, 1994. The most recent release, MPI-3.1 [8], was in June, 2015.

The MPI specification is independent of its underlying implementations for various computer and networking hardware. In other words, a library implementation of MPI to use InfiniBand will deal with details of utilizing the Verbs API, whereas a different library implementation of MPI to use Ethernet will utilize the sockets API, but in both cases the user MPI application will not know or care about the specific details of the underlying technology or its API. This has enabled parallel programs written for MPI to be portable between generations of disparate systems, and as such it has become the most widely used communications mechanism for High Performance Computing (HPC) applications.

An example of a complex application built on MPI is the Community Earth System Model (CESM) [113], a large scientific application comprised of approximately 1.5 million lines of code distributed in a number of large component models [113], [114]. Each component model simulates a different piece of the Earth's physical system. For example, there exist ocean, atmosphere, land, and sea-ice components which simulate each of the Earth's major physical systems. Each component model, developed by different groups of Scientists and Software Engineers, has evolved into its current form over the course of decades.

CESM uses MPI for a wide variety of tasks, including job launching, disk I/O, and passing information between distributed memory spaces. CESM has been ported to a large number of compute platforms, and utilizes a broad diversity of MPI versions, including but not limited to OpenMPI [47], MPI/Pro [115], MPICH [116], [117], MPICH versions and derivatives, including MPICH2 [118], MVAPICH [119], [120], IBM MPI [121], and Intel MPI [122]. Due to the complexity and size of both CESM and MPI, it is impossible to completely describe CESM's use of MPI here. However, there are several basic types of MPI calls that represent the majority of MPI usage within CESM. We first examine two common types of MPI operations: point-to-point communication and collective operators. We next describe in greater detail the most common MPI operators used in CESM. These operations are discussed in more detail in MPI tutorials [21], [22]. More recent tutorials [23], [24] cover additional operation types which are outside the scope of this tutorial, including file I/O and one-sided operations.

TABLE III
MPI OPERATIONS DISCUSSED IN THIS SECTION

	Blocking	Non-blocking
Point to point	MPI_Send()	MPI_Isend()
	MPI_Ssend()	MPI_Issend()
	MPI_Recv()	MPI_Irecv()
Collective	MPI_Barrier()	
	MPI_Reduce()	
	MPI_Allreduce()	

Communication operators in MPI use the concept of a *communicator*, which is an object containing a *group* of MPI processes. Each given communicator will never receive messages sent using a different communicator. An MPI *group* is an ordered (sub)set of unique processes within an MPI job. For bookkeeping purposes, the position (starting at 0) of a process within a group is called its *rank* in that group. Note that a unique MPI process can only appear once within a single group, but one process can belong to many different groups, and hence to many different communicators based on those groups. Since groups are ordered, a process's rank within each of those groups can be different, so its rank is always relative to its membership in a group (or in a communicator containing that group).

This section contains the following subsections: Section IV-A point-to-point communication; Section IV-B collective operations

A. Point-to-Point Communication

Point-to-point communication involves passing messages between two ranks within a communicator. In CESM, point-to-point communication operators are typically used to update neighboring ranks in a bulk-synchronous fashion, which refers to an organizational structure that alternates between execution phases that are strictly local computations followed by communication phases. These communication phases are often referred to as boundary exchanges and are typically composed of point-to-point communication operators. The vast majority of all data moved throughout the network by a CESM job uses point-to-point operations.

Addressing in point-to-point operations is done by specifying parameters for both the communicator and the rank within that communicator of the destination (when sending) or source (when receiving) process. The application also supplies a *tag* value which MPI uses to match send and receive operations. A send operation will only complete a receive operation with a matching tag, although a send or receive operation may specify a wildcard tag which will match a corresponding operation with any tag value. MPI implementations can perform tag matching with specialized tag matching hardware (when available), via a rendezvous protocol that uses IB to transfer the data, or in software by copying data into the appropriate buffers.

There are several different flavors of point-to-point operations within MPI, and these operators may be either blocking or non-blocking, as well as synchronous or asynchronous. Table III categorizes the most common MPI send and receive operations.

The MPI_Recv(), MPI_Send(), and MPI_Ssend() calls are blocking, meaning that they will not return to the user until MPI considers the operation to be complete. For MPI_Recv(), the operation completes when the received data is placed into the user's buffer. MPI_Send() is an asynchronous MPI operation, meaning that the operation completes when the caller may reuse the buffer he supplied to the call. This does not guarantee that MPI has sent the data, and is similar to sockets semantics. MPI_Ssend() is a synchronous send, which has stricter and more complicated semantics. When MPI_Ssend() returns control to the user, the caller is able to re-use its buffer as in MPI_Send(), but additionally, the sender knows that the receiver has issued a matching receive operation and will receive the data in the absence of errors. Again, however, there is no guarantee that the underlying network has sent the data when MPI_Ssend() returns to the caller.

The MPI_Irecv(), MPI_Isend(), and MPI_Issend() calls are non-blocking. These calls behave like their blocking counterparts, but return control immediately to the user after starting the operation. Because it is not safe to modify or use the application buffer as soon as a non-blocking call returns control to its caller, these calls return a handle that the user can use in one of the MPI_Wait* operations to determine when the operation completes and the buffer is free for reuse. Non-blocking operations have two advantages over their blocking counterparts. First, non-blocking communication allows the order that sends complete in the network to be different than the order in which the application issues the sends. Second, it is possible to use non-blocking communications to overlap computations with communications. The performance gains from using this technique depend on the numerical algorithms being used. For example, in CESM, this does not provide a significant improvement for the Spectral Element method but provides a 8% performance improvement for the Discontinuous Galerkin method [123]. This is because the communication in the Spectral Element method has a much stronger dependency on the data transmitted during the boundary exchange. In both cases, the packing and unpacking of messages can also be overlapped with the communication for a minor performance gain.

There is no visible wire protocol difference between the blocking and non-blocking calls. While all variants are used within CESM, the MPI_Isend() variant dominates in terms of both number of calls and total data transferred. For the non-blocking operators, MPI provides functions to verify that the transfer has completed. An application may wait for completion of a specific message using MPI_Wait(), a number of messages using MPI_Waitall(), or at least one of a number of messages using MPI_Waitany(). The MPI_Wait() functions generally map onto the `ibv_poll_cq` verb and the completion channel associated with the completion queue.

MPI supports both an eager and a rendezvous protocol. The eager protocol maps directly onto the `rdma_post_send` and `rdma_post_recv` transfer verbs (see Section III-C). At the start of MPI communication, MPI posts many receive work requests with calls to `rdma_post_recv` using "anonymous" buffers controlled by the MPI implementation. When a message is received, MPI copies the data from the anonymous buffer that received it into the buffer specified by the MPI_Recv operation. For larger messages, the rendezvous protocol is used instead. In this case, the MPI_Send operation maps onto a "memory semantics" (see Section III-F) `rdma_post_write` or `rdma_post_read` verb in order to transfer the data [124]. However, before such a "memory semantics" transfer work request can be posted, the "metadata" about the remote MPI buffer (i.e., its virtual address, length, and memory registration key) must be communicated to the initiator of the operation via an IB "channel semantics" (see Section III-C) exchange as shown in Fig. 20. For small messages, where the overhead introduced by copying is smaller than the extra delay introduced by the rendezvous protocol, the MPI implementation uses a single-message eager protocol instead. More details are provided below in the Experimentation Section V-A.

The eager and rendezvous protocols have performance tradeoffs. The eager protocol requires pre-allocated buffers internal to the MPI implementation, which consume resources that are then unavailable to the MPI application for computation. Additionally, an application which issues many eager operations in a short amount of time can exhaust the eager buffer pool. This will cause a delay while the sender waits for these buffers to become available again. On the other hand, the rendezvous protocol requires MPI to process incoming rendezvous control messages asynchronously to the user application. Common MPI implementations are single-threaded and thus cannot service asynchronous IB events in order to progress communication while that thread is performing application computations. For example, if an MPI operation requires a rendezvous exchange of remote virtual addresses, lengths, and keys before the data transfer itself can start, the MPI library initiates the "channel semantics" exchange and then returns control to the application. If the application never returns control to MPI, MPI will not be aware that this "channel semantics" exchange completed and will thus never post the subsequent "memory semantics" verbs that actually transfer the data. Therefore, when using non-blocking communication, the application programmer must be careful to call one of the MPI_Wait* functions

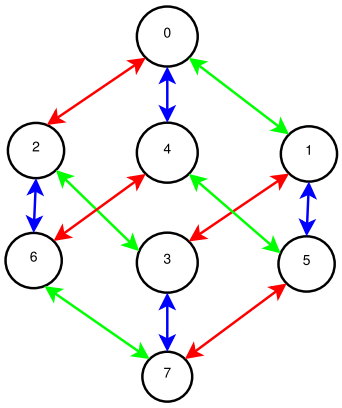


Fig. 23. MPI_Barrier communication pattern for 8 ranks. Note that all communications are symmetric. Green represents the first round of communication, red the second, and blue the third.

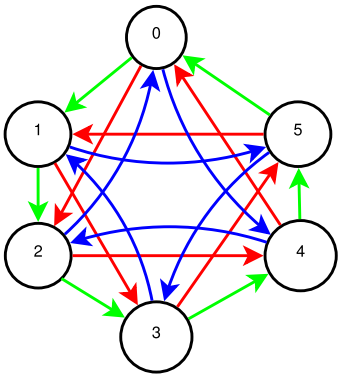


Fig. 24. MPI_Barrier communication pattern for 6 ranks, showing asymmetric communication when the number of ranks is not a power of 2. Green represents the first round of communication, red the second, and blue the third.

at regular intervals to ensure that progress is made on large transfers.

B. Collective Operations

MPI collective operators provide a higher level of abstraction to the user by providing commonly need functionality that involves participation of all processes belonging to a single communicator. Collective operations typically use highly optimized algorithms or hardware to accelerate these common operations. CESM also utilizes collective operators, including Barriers, Allreduce, and Gather operators. The Barrier operation enforces synchronization across all participating ranks, while the Allreduce operator will perform a particular arithmetic operation, such as sum, across all participating ranks. Like the point-to-point operators, the Gather operators allow the communication of data between ranks in a general manner. Note that messages sent as part of collective operations generally use the same underlying network mechanisms as point-to-point operations. We next describe in detail two representative collective operators: Barriers and Allreduce. Other operations, including Gather, are discussed in other MPI tutorials [21], [22].

MPI_Barrier: This call blocks all ranks within a given communicator until they all reach the barrier [125]. It is usually used to synchronize processes between sequences of computation or message exchanges.

Because state-of-the-art MPI implementations are architected to support rank counts greater than 100,000, complex distributed control algorithms are typically used versus simpler non-scalable centralized algorithms. We examined wire traces of the Ohio State University (OSU) MPI_Barrier benchmark [126] when running 2, 4, 6, or 8 ranks within a single communicator on a single InfiniBand node, using the OpenMPI implementation [47], [127], [128]. Using this method, we have determined the algorithms that OpenMPI uses to implement the barrier calls. In particular, if the number of ranks n is a power of 2, OpenMPI will use the communication strategy specified by Brooks' barrier algorithm [129], [130]. See the pictorial example in Fig. 23 for $n = 8$. Brook's algorithm arranges the communication into $\log_2 n$ rounds, numbered 0 through $((\log_2 n) - 1)$, in each of which n messages are exchanged. In each round, each rank exchanges messages with its "neighbor," which is assigned per round such that in round i, rank x's neighbor will be rank $(x \oplus 2^i)$ (i.e., exclusive or). A rank will enter the next round and send its message for that round when it receives the barrier message from its neighbor for the current round.

Brook's algorithm requires that n be a power of 2. For cases where n is not a power of 2, the total number of ranks could be padded to the next-higher power of 2 by having some of the ranks in the barrier simulate the missing ranks. This, however, increases the message complexity, and the cost increases as the number of ranks participating in the barrier increases. Let n' be the next-highest power of 2 greater than n. If there are $n = 4096$ ranks, then $n \log_2 n = 49152$ messages are exchanged, but with 4097 ranks, $n' = 8192$ and there would be $n' \log_2 n' = 106496$ messages exchanged. Thus, OpenMPI instead uses the dissemination algorithm described by Han and Finkel [131] when n is not a power of 2. See Fig. 24 for $n = 6$. Like Brook's algorithm, the communication is arranged into $\lceil \log_2 n \rceil$ rounds of n messages each. The most significant difference between this algorithm and Brook's algorithm is that instead of exchanging messages in pairs, the ranks send messages in a ring (i.e., in round i, rank x sends to rank $((x + 2^i) \mod n)$). This eliminates the symmetry present in Brook's algorithm, but reduces the message complexity from $n' \log_2 n'$ to $n \log_2 n'$. Thus, for $n = 4097$ ranks, only 53261 messages need to be exchanged, about half the 106496 messages exchanged by Brook's algorithm.

A few key points can be observed. First, all barrier calls will result in the same communication pattern during the same run of the application. That is, messages will be exchanged between the same senders and receivers for each MPI_Barrier call (although not necessarily in the same order). Second, when n is a power of 2, all communication is symmetric — every communication from A to B will have a corresponding communication from B to A. This is because the communication in each round of Brooks' algorithm is symmetric. However, this is not always the case when n is not a power of 2. This is because in the dissemination algorithm, each rank x sends to

rank $((2^i + x) \mod n)$ in round i, so ranks do not necessarily receive messages from the ranks they send to, but if they do, it will never be in the same round as the send.

MPI_Reduce and MPI_Allreduce: The MPI_Reduce operation collects values from each rank in the communicator, computes a single value using a reduction function, and stores that result at a user-specified destination rank. The MPI_Allreduce operation is similar, but broadcasts the resulting value to all ranks in the communicator. In order to determine the exact messages exchanged by OpenMPI to implement MPI_Reduce and MPI_Allreduce, we examined wire traces of a simple MPI program that utilized the MPI_PROD (product) operation to multiply distinct prime numbers (one per rank) with varying numbers of ranks. In this way, we could tell where each individual rank's value was being transmitted by factoring the values that we observed on the wire.

For MPI_Reduce with fewer than 8 ranks, each rank sends a message with its value to the destination. However, for an MPI_Reduce operation with 8 or more ranks, OpenMPI uses a simple spanning tree-like communication structure instead. Intermediate values are passed up the communication tree, as shown in Fig. 25. We describe this algorithm assuming that the "root" rank that receives the final result is rank 0.[2] If the number of ranks $n \geq 1$ is an exact power of 2 ($n = 2^k, k \geq 0$), then the communication occurs in k rounds. In each round i starting at $i = 0$, rank x ($0 \leq x < n = 2^k$) will send a message with its accumulated value to rank $y = (x - 2^i)$ if and only if the binary representation of x has a 1 at position i, where $i = 0$ represents the least significant bit. At the end of k rounds, the "root" rank has the total accumulated value. When n is not an exact power of 2, then let $n' = 2^k$ be the next higher power of 2 greater than n (so that $2^{(k-1)} < n < n' = 2^k$) and substitute n' for n in the above formulas. Note that this algorithm means that these communication trees are not balanced; rather, for all n, these trees will have at least one leaf of height 1, one leaf of height 2, and so on up through height k.

We have determined that the implementation of the MPI_Allreduce operation in OpenMPI uses the communication graph given by Brook's algorithm [129], [130] as it does for MPI_Barrier when the number of ranks n is a power of 2. In each round, the pairs of ranks exchange the value that has been computed so far by the sending rank. At the end of the algorithm, each rank has received enough information to compute the final overall reduced value. In each step, each rank sends the reduced value that it has obtained thus far to its neighbor rank.

However, when the total number of ranks n is not a power of two, the ranks are split between the highest power of two less than n "core" ranks and the remaining "extra" ranks. Each of the "extra" ranks x sends its value to a corresponding "core" rank y as an extra step before Brook's algorithm starts, which reduces the problem to one that is solved efficiently by Brook's algorithm. The "core" ranks do the normal MPI_Allreduce algorithm, and then each "extra" rank x receives the final result from its corresponding "core" rank y. This effectively reduces the message complexity from $n' \log_2 n'$ (where n' is the next-highest power of 2) that would be required by a naïve implementation of Brook's algorithm to $n \log_2 n + 2(n' - n)$, in which the second term is insignificant if the difference between n and n' is small. Unlike MPI_Barrier, this works for MPI_Allreduce because we are not concerned about efficiently returning at approximately the same time for all ranks; rather, we are concerned with communicating a result to each rank with the least possible number of messages.

[2]If the "root" rank for MPI_Reduce is *not* 0, then for the purposes of the MPI_Reduce algorithm, OpenMPI will treat each rank x as though it were rank $((x + n - r) \mod n)$, where r is the root rank and n is the total number of ranks in the communicator.

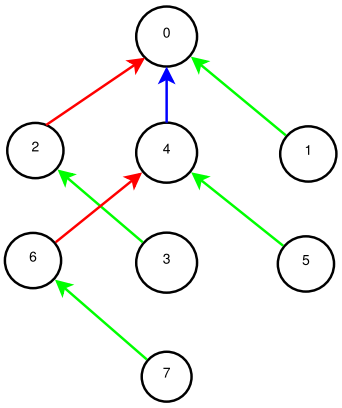


Fig. 25. MPI_Reduce communication pattern for 8 ranks. Note that this uses a tree-like structure. Green represents the first round of communication; red the second; blue the third.

V. Experimentation

We ran several small performance studies to illustrate performance tradeoffs when running MPI on an InfiniBand cluster. We ran our tests at the UNH InterOperability Laboratory [132]. For these experiments, two nodes with Mellanox ConnectX-3 4X Fourteen Data Rate (FDR) InfiniBand Host Channel Adapters (HCAs) were connected to a Mellanox SX6036 switch. The cluster nodes were using OpenFabrics Enterprise Distribution (OFED) 3.18-2 on Scientific Linux 7.2. MPI tests used a single MPI communicator containing two ranks, one on each node.

Although Community Earth System Model (CESM) applications running on supercomputers typically use a vendor-proprietary implementation of MPI, for this study, we used OpenMPI 2.0.0 [47], since it is a readily available open-source implementation of MPI and allows our work to be more reproducible. All of our analysis applies to this version, as MPI [8] does not mandate a wire protocol.

The ibdump command [30] is a tool provided by Mellanox to allow an administrator to capture traffic on an InfiniBand link. It is analogous to tcpdump for Ethernet networks, and produces traces in the "Packet CAPture (PCAP)" file format. Wireshark [28], a widely used open-source packet analyzer, can analyze these traces. We used version 2.0.0-8 of ibdump.

Extrae and paraver are performance analysis tools produced by the Barcelona Supercomputing Center [133], [134]. Extrae is an agent that can be linked to a program and will collect performance information into a trace file. For our experiments,

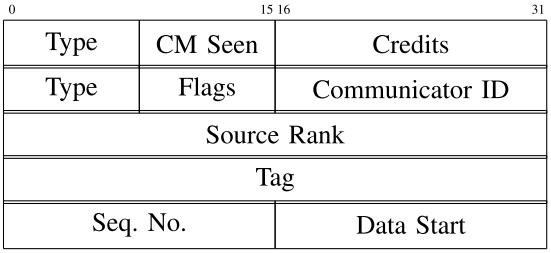


Fig. 26. OpenMPI wire protocol format.

we only enabled MPI tracing, since the other tracing offered by extrae consumes excessive CPU cycles, which interferes with the performance of the MPI operations we are measuring. We used extrae 3.1.0 to collect the traces.

Paraver is a graphical performance analysis tool. We used it primarily to display a graph of MPI calls that our program made over time, although it is also capable of some basic statistical analysis on trace data. We did not use paraver's analysis tools, however, since they did not provide sufficient flexibility for our needs. Instead, to analyze these traces, we wrote a program to calculate the time taken in each call so that we could perform our own statistical analysis on the durations.

We tuned our Linux systems for low latency. We configured the "tuned" service on each system to use the "network-latency" profile, which enables a number of optimizations to reduce wakeups [135]. These include disabling transparent huge pages, disabling automatic Non Uniform Memory Access (NUMA) balancing, and disabling deep CPU sleep states. Additionally, we configured the irqbalance service to respect CPU affinity hints provided by the kernel (by default it ignores them). Finally, we configured our systems to omit scheduling ticks on all CPUs involved in network transfers.

This section is subdivided into the following subsections: Section V-A wire protocol analysis; Section V-B verbs performance analysis; Section V-C MPI point-to-point performance analysis; Section V-D conclusions.

A. Wire Protocol Analysis

Fig. 26 shows the OpenMPI wire protocol format for control and small data messages. When using InfiniBand verbs and small messages, OpenMPI usually uses the verbs for channel semantics transfers. Each MPI message contains two message type fields that have identical contents. The source rank field of the message contains the rank of the process that sent it. The tag field contains the tag value of the message. If this is less than or equal to the **MPI_TAG_UB** attribute of the communicator, then this is a user-supplied tag. Otherwise, it is a system tag used by some other MPI message, which is typically a collective operation. The final header field, the sequence number field, is that of the logical MPI operation from the perspective of the sender, which the MPI library applies internally. Note that the sequence number at each rank is only advanced if the rank actually sends a message for the given MPI operation, so it is possible for messages from different source ranks that belong to a single MPI operation to have different sequence numbers. Additionally, the MPI sequence number has no relation in any way to the InfiniBand protocol sequence number. This is because one MPI operation may actually map into multiple InfiniBand messages on the wire.

Note that, as shown in Fig. 26, OpenMPI messages contain the source rank but not the destination rank. This presents a major hurdle for analyzing the ibdump traces for the MPI collective operations. At the start of the experiments involving collective operations, we added an "initialization" phase: we used an MPI_Send call to send a message containing the destination node's rank from each node to each other node. We then ran the ibdump trace through a program which would parse the trace and build a table from the InfiniBand destination queue pair to the destination MPI rank. Then, for every MPI wire protocol message it would print the source and destination ranks and message type. This allows easy analysis of MPI message flows.

OpenMPI uses a rendezvous protocol instead of the eager protocol in two cases: if the message size is larger than the eager limit (**openib_eager_limit**), or if a process has received many messages from a given peer (the limit is specified in **openib_eager_rdma_threshold**). OpenMPI offers two verbs-based versions of the rendezvous protocol [124], one based on `rdma_post_read` (Fig. 22) and one based on `rdma_post_write` (Fig. 21). We will refer to these protocols based on the RDMA verb used to perform the data transfer. Which RDMA verb MPI uses depends on the setting of the MPI **openib_flags** parameter. If the GET flag is set (the default), then the `rdma_post_read` protocol is used. Otherwise, if the PUT flag is set, then the `rdma_post_write` protocol is used. An administrator or user may set all of these parameters in a configuration file or the **mpirun** command line.

The rendezvous protocol based on `rdma_post_write` requires four IB transfers in order to perform the single MPI message transfer:

1) an `rdma_post_send` by the sender requests "metadata" (i.e., the virtual address, length, and memory registration key) about the receive-side buffers from the receiver,
2) an `rdma_post_send` by the receiver delivers the "metadata" about the user's receive buffer to the sender,
3) the sender uses this metadata in an `rdma_post_write` that "pushes" the data into the receiver's memory,
4) an `rdma_post_send` by the sender notifies the receiver that the MPI data transfer has completed.

However, the rendezvous protocol based on `rdma_post_read` only requires three IB transfers:

1) the sender starts the rendezvous by sending an `rdma_post_send` containing metadata about the send data,
2) the receiver uses this metadata in an `rdma_post_read` that "pulls" the data from the sender's memory,
3) an `rdma_post_send` by the receiver notifies the sender that the MPI data transfer has completed.

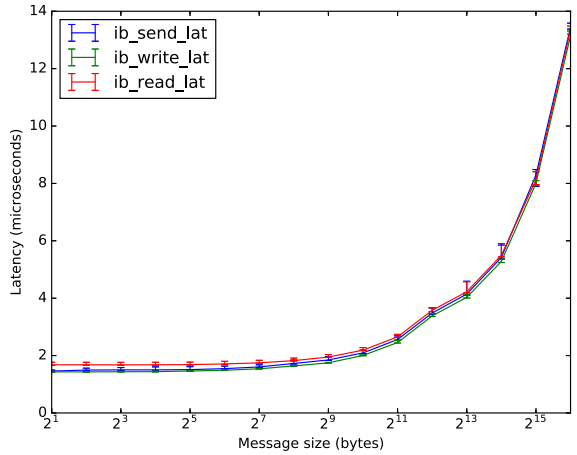


Fig. 27. Latency vs. message size for the basic perftest utilities corresponding to the three basic RDMA transfer operations. Lower is better.

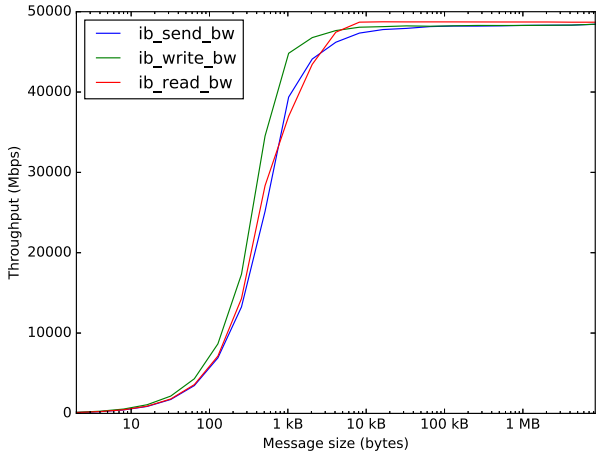


Fig. 28. Throughput vs. message size for the basic perftest utilities corresponding to the three basic RDMA transfer operations. Higher is better.

While the rendezvous protocol based on `rdma_post_read` requires fewer IB transfers, the `rdma_post_read` verb has higher (worse) latency than `rdma_post_write` since it moves data in the opposite direction of the request and thus requires a round trip as well as consuming resources on the remote channel adapter [111].

Synchronous MPI send operations (MPI_Ssend()) require a partial rendezvous to confirm that the receiver has initiated an MPI_Recv() or MPI_Irecv() operation with a matching tag before the MPI library at the sender notifies the sending application that the operation is complete. For eager messages in OpenMPI, the MPI library at the sender sends a special rendezvous message with the eager data attached. Once the receiver matches this send with a receive, it will send an MPI-level acknowledgment message to the sender which completes the operation and allows the MPI_Ssend() call to return.

B. Verbs Performance Analysis

The perftest suite [136] provides a set of standard performance benchmarks for Host Channel Adapters (HCAs). The suite provides latency and bandwidth tests for the three major InfiniBand transfer operations: `rdma_post_send`/`rdma_post_recv`, `rdma_post_write`, and `rdma_post_read`. These results are useful as a baseline to compare against when writing applications.

Fig. 27 and 28 present perftest latency and throughput results from our test environment, respectively. The latency measurements for each of the basic RDMA operations are similar, as expected, with `rdma_post_write` having the lowest (best) latency since the application receiving side is not involved in the transfers, and `rdma_post_read` having the highest latency due to requiring a round trip before any actual data is transferred. Looking at throughput, however, although `rdma_post_write` has the highest throughput for messages of 4096 bytes or fewer, `rdma_post_read` exceeds the `rdma_post_write` throughput for larger messages. This is because an HCA may send multiple `rdma_post_read` requests simultaneously, and the receiving channel adapter can pipeline efficiently by starting to sending the data for the second `rdma_post_read` immediately after the first completes. On the other hand, for `rdma_post_write`, the sending HCA must wait for the first operation to be acknowledged before starting the second operation.

C. MPI Point-to-Point Performance Analysis

We used OpenMPI to examine the performance of the basic variants of the MPI_Send operation in terms of the amount of time spent inside each function call. To do this, we modified the osu_latency tool from the OSU Micro-Benchmarks [126] to allow the user to select which MPI_Send variant to use for the test run. For each variant we sent 1.5 million 8192-byte messages in a ping-pong format, measuring the duration of each function call via tracing with Extrae. We chose these numbers to provide a representative message size that would exercise the InfiniBand fabric, and produce a test runtime of above 10 seconds, which is long enough to reach a steady state and thus have confidence in the results. For the non-blocking send operations, we show the time spent in the MPI_Isend call and the time spent from the end of the MPI_Isend call until the subsequent MPI_Wait call terminates.

We first examine the performance of the eager protocol, as shown in Table IV. Here, we can see that in most cases MPI_Isend and MPI_Send perform similarly. This is because the eager protocol copies into an intermediate buffer and sends a single message, and although MPI_Send is a blocking operation, this implementation allows the sender to complete small MPI_Send operations immediately without blocking — the send must only block if there is insufficient HCA queue space or eager buffer space remaining. Thus, for these two calls, 90% of the total durations (including completion duration with MPI_Isend) are less than 1 microsecond. However, if we add an MPI_Isend call duration with its corresponding completion duration, the result is always larger than its corresponding MPI_Send duration. This amounts to overhead due to the asynchronous nature of MPI_Isend. In particular, MPI_Isend must set up a request handle for the user to pass back to one of the MPI_Wait functions, adding a slight amount of overhead.

TABLE IV
DISTRIBUTION OF MPI_ISEND, MPI_SEND, AND MPI_SSEND DURATIONS IN NANOSECONDS (OUT OF 1500000 SAMPLES) WHEN USING THE **EAGER PROTOCOL**. DUE TO THE HIGHLY SKEWED NATURE OF THESE DISTRIBUTIONS, WE PRESENT QUANTILES RATHER THAN THE MEAN AND STANDARD DEVIATION. THE COMPLETION COLUMN SHOWS THE TIME FROM THE END OF THE MPI_ISEND CALL UNTIL THE TIME OF THE END OF THE CORRESPONDING MPI_WAIT

Percentile	MPI_Isend	Completion	MPI_Send	MPI_Ssend
Minimum	564	160	626	4380
25	628	215	673	4548
50	657	221	683	4618
75	718	231	764	5196
90	740	253	788	5346
99	1979	1551	2012	6469
99.9	2927	2771	2965	7202
99.99	3653	3462	3705	8418
99.999	4041	3908	3901	16159
Maximum	8105	14305	14237	30909
Skewness	5.4318	4.4060	5.7091	2.5418

TABLE V
DISTRIBUTION OF MPI_ISEND, MPI_SEND, AND MPI_SSEND DURATIONS IN NANOSECONDS (OUT OF 1500000 SAMPLES) WHEN USING THE **RENDEZVOUS PROTOCOL** BASED ON `RDMA_POST_READ`. DUE TO THE HIGHLY SKEWED NATURE OF THESE DISTRIBUTIONS, WE PRESENT QUANTILES RATHER THAN THE MEAN AND STANDARD DEVIATION. THE COMPLETION COLUMN SHOWS THE TIME FROM THE END OF THE MPI_ISEND CALL UNTIL THE TIME OF THE END OF THE CORRESPONDING MPI_WAIT

Percentile	MPI_Isend	Completion	MPI_Send	MPI_Ssend
Minimum	354	4741	5232	5203
25	389	5461	6171	5640
50	400	5744	6303	6260
75	415	6073	6387	6345
90	434	6208	6567	6650
99	1704	6868	7907	7855
99.9	2722	7385	9258	8752
99.99	3736	8397	10135	10062
99.999	4211	17576	16402	17118
Maximum	48030	67589	124374	120310
Skewness	10.311	4.4039	11.569	8.5386

In practice, however, an application can absorb this overhead by queuing many such operations simultaneously and using MPI_Waitall, thus reducing the amount of time spent waiting for completions.

On the other hand, MPI_Ssend must wait for an MPI-level acknowledgment before returning to the user, to ensure that it has matched an MPI_Recv request at the receiver. This means that the MPI_Ssend calls must actually block in all cases, so the calls take almost an order of magnitude longer than calls to MPI_Send under the eager protocol (the minimum duration for MPI_Ssend is over 4 microseconds compared to a minimum duration of less than a microsecond for MPI_Send).

Table V shows the same operations, but using the rendezvous protocol (here, we study the OpenMPI default rendezvous protocol based on `rdma_post_read`). The overall duration per operation is larger than for the eager protocol, since the rendezvous protocol must do more work. We also see some interesting patterns. The MPI_Isend calls have a short duration—90% of the observed durations were under a microsecond—since the call simply posts the first rendezvous message and returns. The MPI_Isend completion durations, although larger, are less skewed than for the eager protocol, since 99.99% of the durations are less than 9 microseconds.

When we examine the MPI_Send durations for the rendezvous protocol, we find that at the minimum through the 50th percentile, the MPI_Send duration is approximately the sum of the MPI_Isend and MPI_Isend completion durations. We also observe that the times for MPI_Send and MPI_Ssend have an almost identical overall distribution. This makes sense since once MPI returns control to the user, the application is free to modify the send buffer, which means that MPI must ensure that the data has been transferred before returning control to the user. Thus, when using the rendezvous protocol, MPI_Send and MPI_Ssend must do the same amount of work before returning control to the user.

D. Conclusion

1) The one-sided `rdma_post_write` and `rdma_post_read` operations can lower the CPU overhead of large message transfers by making one side completely passive. The `rdma_post_write` operation has the best latency. However, in some

situations `rdma_post_read` can offer marginally higher throughput, since multiple such requests can be queued simultaneously at the responder.

2) Using the MPI eager protocol for small MPI_Isend and MPI_Send operations can offer very fast data transfers (in hundreds of nanoseconds), but requires intermediate buffers which consume memory that would otherwise be available to the application. Additionally, if the eager buffers fill (which would be more likely in an application that transferred data only in a single direction) then the eager transfers will stall until the buffers can be drained.
3) The rendezvous protocol takes more time per message and requires more wire messages, but eliminates the need for large data copies by taking advantage of the zero-copy semantics of verbs.
4) Even with performance tuning, the duration of MPI operations have a significant amount of variation above the 90^{th} percentile.

VI. Future Directions

The InfiniBand architecture continues to evolve to support future use cases.

A future High Data Rate (HDR) speed has been defined at approximately 200 Gbps, and although products have been announced supporting this speed, the speed has not yet been specified in the latest version of the IB standard [35] at the time of this writing.

The Virtualization Annex [137] defines a Virtual Host Channel Adapter (VHCA) via which an HCA can expose transport endpoints to multiple virtual machines on a system, preserving the virtual machine isolation. Each VHCA is assigned a unique GID by the Subnet Manager (SM).

InfiniBand must adapt to emerging low-latency Non-Volatile Memory (NVM) technologies. This includes the ability to register *peer memory*—memory from other devices on the PCIe bus [138], including Graphics Processing Unit (GPU) memory and Non-Volatile Memory (NVM) devices. Additionally, writes to NVM must explicitly be made durable using a flush operation [139], or the written data may not survive a power loss. Applications which use `rdma_post_write` to write to remote NVM devices would benefit from an addition to verbs to allow an application to explicitly perform a flush as part of, or following, `rdma_post_write` operations. HCAs may also require an application to indicate that a memory region is on a NVM device at registration time, to prepare the HCA to allow a flush operation on the memory region.

The MPI specifications also must evolve to meet the needs of the HPC community [140]. One future work area defines a new MPI object called a session, which allows easier use of MPI from within a library. Additionally, work is being done on allowing multiple MPI endpoints per process, which may improve thread-safety and make it easier to combine MPI with other HPC middleware communication frameworks.

VII. Summary

This tutorial has provided an in-depth discussion of the three major components of the software/hardware interconnect that work together to enable a High Performance Computing (HPC) application to utilize a modern large-scale computing cluster.

Section II presented the InfiniBand Architecture (IBA) and its unique features designed for high-performance: zero-copy direct memory-to-memory transfers to avoid extra copying and buffering in the end nodes; kernel bypass so that user-level applications can access the Host Channel Adapter (HCA) directly in order to transfer data without operating system intervention; asynchronous operation so that network transfers can overlap with computation; and message orientation to eliminate any application overhead to track message boundaries in a byte stream.

We described InfiniBand's major architectural components, its methods of network addressing, the format of its packets on the wire, its link-level flow control scheme that eliminates packet drops caused by lack of space on the receiving end of a link, support for Virtual Lanes (VLs), packet forwarding in switches, and transport-level reliable connection services, including connection management, end-to-end error control, and congestion control. Our explanations provided instructive similarities and differences between InfiniBand and the better known TCP/IP/Ethernet.

Section III presented the OpenFabrics Alliance (OFA) Verbs Application Programming Interface (API), a software library that provides functions and data structures that allow user-space programs to transparently setup and utilize RDMA technologies, which include InfiniBand as well as Internet Wide Area RDMA Protocol (iWARP) and RDMA over Converged Ethernet (RoCE). Again we demonstrated useful similarities and differences between the verbs interface to InfiniBand and the sockets interface to TCP/IP/Ethernet, especially the notions of memory registration and memory semantics operations unique to InfiniBand. We gave examples of how common verbs are utilized to send and receive data and to handle completions, and a number of lessons learned from practical experience with verbs on how to increase application performance.

Section IV presented a snapshot of how OpenMPI provides an HPC application, such as the Community Earth System Model (CESM), with efficient access to InfiniBand via the Verbs API. Since MPI has been the interface of choice for HPC applications for several decades, this section gave HPC application programmers and users a "peek under the covers" to reveal exactly how common MPI point-to-point and collective communication operations transparently and efficiently utilize the features of the latest high-performance interconnects such as InfiniBand.

Section V presented some simple experiments we ran to capture wire traces in order to analyze the performance of OpenMPI's mappings of various MPI transfer operations onto verbs and their observed latency and throughput.

Section VI presented some future developments being undertaken to achieve even faster InfiniBand speeds, to expand

the types of memory that can be accessed directly by InfiniBand, and to enhance MPI specifications to meet the requirements of the HPC user community.

ACRONYMS

ABR	Adjusted Blocks Received
ACK	ACKnowledgement.
AETH	ACK Extended Transport Header.
API	Application Programming Interface.
BECN	Backward Explicit Congestion Notification.
BFS	Breadth First Search.
BSD	Berkeley Software Distribution.
BTH	Base Transport Header.
CCT	Congestion Control Table.
CCTI	Congestion Control Table Index.
CESM	Community Earth System Model.
CL	Credit Limit.
CM	Connection Manager.
CNP	Congestion Notification Packet.
CPU	Central Processing Unit.
CQ	Completion Queue.
CRC	Cyclic Redundancy Code.
CS	Completion Status.
DDR	Double Data Rate.
DETH	Datagram Extended Transport Header.
DFS	Depth-First Search.
DLID	Destination LID.
EDR	Extended Data Rate.
EUI	Extended Unique Identifier.
FB	Free Blocks.
FCCL	Flow Control Credit Limit.
FCP	Flow Control Packet.
FCTBS	Flow Control Total Blocks Sent.
FDB	Forwarding Data Base.
FDR	Fourteen Data Rate.
FECN	Forward Explicit Congestion Notification.
FIFO	First-In First-Out.
Gbps	Gigabits per second.
GID	Global Identifier.
GPU	Graphics Processing Unit.
GUID	Globally Unique Identifier.
HCA	Host Channel Adapter.
HDFS	Hadoop Distributed File System.
HDR	High Data Rate.
HFT	High Frequency Trading.
HoL	Head of Line.
HPC	High Performance Computing.
IB	InfiniBand.
IBA	InfiniBand Architecture.
IBTA	InfiniBand SM Trade Association.
ICRC	Invariant Cyclic Redundancy Code.
IP	Internet Protocol.
IRD	Injection Rate Delay.
iWARP	Internet Wide Area RDMA Protocol.
LASH	LAyered SHortest path.
LID	Local IDentifier.
LMC	LID Mask Control.
LPCRC	Link Packet CRC.
LRH	Local Routing Header.
MAC	Media Access Control.
MAD	Management Datagram.
MPI	Message Passing Interface.
MPICH	MPI CHameleon.
MSN	Message Sequence Number.
MTU	Maximum Transmission Unit.
NACK	Negative ACKnowledgement.
NIC	Network Interface Card.
NUMA	Non Uniform Memory Access.
NVM	Non-Volatile Memory.
OFA	OpenFabrics Alliance.
OFED	OpenFabrics Enterprise Distribution.
OP	OPeration.
OpCode	Operation Code.
OSU	Ohio State University.
PCAP	Packet CAPture.
PCIe	Peripheral Component Interconnect express.
PGAS	Parallel Global Address Space.
PHY	PHYsical Layer.
PSN	Packet Sequence Number.
QDR	Quad Data Rate.
QoS	Quality of Service.
QP	Queue Pair.
RC	Reliable Connection.
RD	Reliable Datagram.
RDMA	Remote Direct Memory Access.
REP	REPly.
REQ	REQuest.
RETH	RDMA Extended Transport Header.
RMA	Remote Memory Access.
RoCE	RDMA over Converged Ethernet.
RQ	Receive Queue.
RTU	Ready To Use.
RWR	Receive Work Request.
SDN	Software Defined Networking.
SDON	Software Defined Optical Network.
SDP	Sockets Direct Protocol.
SDR	Single Data Rate.
SL	Service Level.
SL2VL	Service Level to Virtual Lane.
SLAAC	Stateless Address AutoConfiguration.
SLID	Source LID.
SM	Subnet Manager.
SMA	Subnet Management Agent.
SMP	Subnet Management Packet.
SQ	Send Queue.
SSN	Send Sequence Number.
SWR	Send Work Request.
TBS	Total Blocks Sent.
TOE	TCP Offload Engine.
UC	Unreliable Connection.
UD	Unreliable Datagram.
VCRC	Variant Cyclic Redundancy Code.
VHCA	Virtual Host Channel Adapter.
VL	Virtual Lane.
VL0	Virtual Lane 0.

VL15	Virtual Lane 15.
VM	Virtual Memory.
VOQ	Virtual Output Queue.
WC	Work Completion.
WQ	Work Queue.
WR	Work Request.
XRC	eXtended Reliable Connection.

REFERENCES

[1] R. Loft *et al.*, "Yellowstone: A dedicated resource for earth system science," in *Contemporary High Performance Computing: From Petascale Toward Exascale*, vol. 2. Boca Raton, FL, USA: CRC Press, 2014.

[2] *Yellowstone*. Accessed: Aug. 2014. [Online]. Available: https://www2.cisl.ucar.edu/resources/ yellowstone

[3] S. J. Martin and M. Kappel, "Cray XC30 power monitoring and management," in *Proc. Cray User Group (CUG)*, May 2014, pp. 1–11. [Online]. Available: https://cug.org/proceedings/cug2014_proceedings/includes/files/pap130.pdf

[4] *InfiniBand Architecture Specification Volume 1, Release 1.3*, InfiniBand Trade Assoc. Stand., Beaverton, OR, USA, Mar. 2015. [Online]. Available: http://infinibandta.org

[5] T. Shanley and J. Winkles, *InfiniBand Network Architecture*. Boston, MA, USA: Addison-Wesley, 2003.

[6] P. Grun, *Introduction to InfiniBand for End Users*, InfiniBand Trade Assoc., Beaverton, OR, USA, 2010. [Online]. Available: cw.infinibandta.org/document/dl/7268

[7] *Top 500 the List*. Accessed: Aug. 2017. [Online]. Available: https://www.top500.org

[8] Message Passing Interface Forum. (Jun. 2015). *MPI: A Message-Passing Interface Standard, Version 3.1*. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[9] *The Message Passing Interface (MPI) Standard*. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/

[10] B. K. Park, W.-W. Jung, and J. Jang, "Integrated financial trading system based on distributed in-memory database," in *Proc. Conf. Res. Adapt. Convergent Syst. (RACS)*, Towson, MD, USA, Oct. 2014, pp. 86–87.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. Conf. SIGCOMM (SIGCOMM)*, Chicago, IL, USA, Aug. 2014, pp. 295–306.

[12] J. Jang, Y. Cho, J. Jung, and S. Yoon, "Concurrency control scheme for key-value stores based on InfiniBand," in *Proc. Conf. Res. Adapt. Convergent Syst. (RACS)*, Towson, MD, USA, Oct. 2014, pp. 356–358.

[13] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *Proc. Int. Conf. Cluster Comput. (Cluster)*, Austin, TX, USA, Sep. 2007, pp. 11–20.

[14] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over InfiniBand: An approach using a high performance network block device," in *Proc. Int. Conf. Cluster Comput.*, Burlington, MA, USA, Sep. 2005, pp. 1–10.

[15] N. S. Islam *et al.*, "High performance RDMA-based design of HDFS over InfiniBand," in *Proc. Int. Conf. High Perform. Comput. Netw. Stor. Anal. (SC)*, Salt Lake City, UT, USA, Nov. 2012, pp. 1–12.

[16] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin, "Zero copy sockets direct protocol over InfiniBand-preliminary implementation and performance analysis," in *Proc. 13th Annu. Symp. High Perform. Interconnects (HOTI)*, Stanford, CA, USA, 2005, pp. 128–137.

[17] D. Goldenberg, *InfiniBand Technology Overview*, Stor. Netw. Ind. Assoc., Colorado Springs, CO, USA, 2007. [Online]. Available: www.snia.org/sites/default/education/tutorials/2007/fall/networking/DrorGoldenberg_InfiniBand_Technology_Overview.pdf

[18] R. Noronha, X. Ouyang, and D. K. Panda, "Designing a high-performance clustered NAS: A case study with pNFS over RDMA on InfiniBand," in *Proc. 15th Int. Conf. High Perform. Comput. (HiPC)*, Bengaluru, India, Dec. 2008, pp. 465–477.

[19] J. W. Choi, D. I. Shin, Y. J. Yu, H. EOM, and H. Y. Yeom, "Towards high-performance san with fast storage devices," *ACM Trans. Stor.*, vol. 10, no. 2, Mar. 2014, Art. no. 5.

[20] G. F. Pfister, "An introduction to the InfiniBand architecture," in *High Performance Mass Storage Parallel I/O: Technologies and Applications*, H. Jin, T. Cortes, and R. Buyya, Eds. New York, NY, USA: Wiley, 2001, ch. 42, pp. 617–632. [Online]. Available: gridbus.csse.unimelb.edu.au/∼raj/superstorage/chap42.pdf

[21] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message-Passing Interface*, vol. 1, 2nd ed. Cambridge, MA, USA: MIT Press, 1999.

[22] B. Barney. *Message Passing Interface (MPI) Tutorial, UCRL-MI-133316*. Accessed: Aug. 2014. [Online]. Available: https://computing.llnl.gov/tutorials/mpi/

[23] W. Gropp, R. Thakur, and E. Lusk, *Using MPI-2: Advanced Features of the Message-Passing Interface*, 2nd ed. Cambridge, MA, USA: MIT Press, 1999.

[24] W. Gropp, R. Lusk, R. Thakur, and R. Ross, "Advanced MPI: I/O and one-sided communication," in *Proc. ACM/IEEE Conf. Supercomput.*, Tampa, FL, USA, 2006, p. 202.

[25] O. B. Fredj *et al.*, "Survey on architectures and communication libraries dedicated for high speed networks," *J. Ubiquitous Syst. Pervasive Netw.*, vol. 3, no. 2, pp. 79–86, 2011.

[26] K. Kant, "Data center evolution: A tutorial on state of the art, issues, and challenges," *Comput. Netw.*, vol. 53, no. 17, pp. 2939–2965, Dec. 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128609003090

[27] O. Torudbakken and A. V. Krishnamoorthy, "A 50Tbps optically-cabled InfiniBand datacenter switch," in *Proc. Opt. Fiber Commun. Conf. Expo. Nat. Fiber Opt. Eng. Conf. (OFC/NFOEC)*, Anaheim, CA, USA, Mar. 2013, pp. 1–3.

[28] *Wireshark*. Accessed: Nov. 2016. [Online]. Available: http://www.wireshark.org/

[29] *TCPDUMP*. Accessed: Sep. 2017. [Online]. Available: http://www.tcpdump.org/

[30] *IBDUMP—Monitoring and Debug Tool*. Accessed: Aug. 2014. [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=110&mtag=monitoring_debug

[31] S. Richling, H. Kredel, S. Hau, and H.-G. Kruse, "A long-distance InfiniBand interconnection between two clusters in production use," in *Proc. Int. Conf. High Perform. Comput. Netw. Stor. Anal. State Pract. Rep. Art. (SC)*, Seatle, WA, USA, Nov. 2011, pp. 1–8.

[32] R. Hinden and S. Deering, "IP version 6 addressing architecture," Internet Eng. Task Force, Fremont, CA, USA, RFC 4291, Feb. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4291.txt

[33] S. Thomson, T. Narten, and T. Jinmei, "IPv6 stateless address autoconfiguration," Internet Eng. Task Force, Fremont, CA, USA, RFC 4862, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4862.txt

[34] W. Nienaber, X. Yuan, and Z. Duan, "LID assignment in InfiniBand networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 4, pp. 484–497, Apr. 2009.

[35] *InfiniBand Architecture Specification Volume 2, Release 1.3*, InfiniBand Trade Assoc. Stand., Beaverton, OR, USA, Nov. 2012. [Online]. Available: http://infinibandta.org

[36] *PCI Express Base Specification Revision 3.1a*, Peripheral Compon. Interconnect Special Interest Group, Beaverton, OR, USA, Dec. 2015. [Online]. Available: http://www.pcisig.com

[37] M. J. Koop, W. Huang, K. Gopalakrishan, and D. K. Panda, "Performance analysis and evaluation of PCIe 2.0 and quad-data rate InfiniBand," in *Proc. 16th IEEE Symp. High Perform. Interconnects*, Stanford, CA, USA, 2008, pp. 85–92.

[38] *IEEE Standard for Ethernet*, IEEE Standard 802.3-2015, Mar. 2016.

[39] R. Walker and R. Dugan, *64b/66b Low-Overhead Coding Proposal for Serial Links (Update)*, IEEE Standard 802.3, Jan. 2000. [Online]. Available: http://grouper.ieee.org/groups/802/3/10G_study/public/jan00/walker_1_0100.pdf

[40] *Plugfest #31—April 2017 Combined Cable and Device Integrators' List*, InfiniBand Trade Assoc., Beaverton, OR, USA, Apr. 2017. [Online]. Available: https://cw.infinibandta.org/document/dl/8200

[41] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks*, IEEE Standard 802.1Q-2014, Dec. 2014.

[42] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proc. IRE*, vol. 49, no. 1, pp. 228–235, Jan. 1961.

[43] *Information Technology—High Performance Parallel Interface—6400 Mbit/s Physical Switch Control*, ANSI NCITS Standard 324-1999, 1999.

[44] D. Crupnicoff, S. Das, and E. Zahavi. *Deploying Quality of Service and Congestion Control in InfiniBand-based Data Center Networks*. Accessed: Nov. 2016. [Online]. Available: http://www.mellanox.com/pdf/whitepapers/deploying_qos_wp_10_19_2005.pdf

[45] Q. Liu and R. D. Russell, "Analyzing InfiniBand packets," Annu. OpenFabrics Softw. User Group Workshop, Monterey, CA, USA, Mar 2015. [Online]. Available: http://www.openfabrics.org/images/eventpresos/workshops2015/UGWorkshop/Thursday/thursday_09.pdf

[46] H. Yi, S. Park, M. Kim, and K. Jeon, "An efficient buffer allocation technique for virtual lanes in InfiniBand networks," in *Proc. 2nd Int. Conf. Human Soc. Internet (HSI)*, Seoul, South Korea, 2003, pp. 272–281. [Online]. Available: http://dl.acm.org/citation.cfm?id=1758796.1758834

[47] (2009). *Open MPI: Open Source High Performance Computing*. [Online]. Available: http://www.open-mpi.org

[48] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda, "High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters," in *Proc. 21st Annu. Int. Conf. Supercomput. (ICS)*, Seattle, WA, USA, Jun. 2007, pp. 180–189.

[49] M. J. Koop, R. Kumar, and D. K. Panda, "Can software reliability outperform hardware reliability on high performance interconnects?: A case study with MPI over InfiniBand," in *Proc. 22nd Annu. Int. Conf. Supercomput. (ICS)*, Jun. 2008, pp. 145–154.

[50] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time MPI broadcast algorithm for large-scale InfiniBand clusters with multicast," in *Proc. 21st Int. Parallel Distrib. Process. Symp. (IPDPS)*, Rome, Italy, 2007, pp. 1–8.

[51] *OpenSM Project*. Accessed: Nov. 2016. [Online]. Available: https://www.openhub.net/p/opensm

[52] A. Bermudez, R. Casado, F. Quiles, T. Pinkston, and J. Duato, "On the InfiniBand subnet discovery process," in *Proc. IEEE Int. Conf. Cluster Comput.*, Hong Kong, Dec. 2003, pp. 512–517.

[53] J. J. Garcia-Luna-Aceves, "A minimum-hop routing algorithm based on distributed information," *Comput. Netw. ISDN Syst.*, vol. 16, no. 5, pp. 367–382, May 1989. [Online]. Available: www.sciencedirect.com/science/article/pii/01697552800111

[54] F. J. Alfaro *et al.*, "On the performance of up*/down* routing," in *Network-Based Parallel Computing. Communication, Architecture, and Applications* (LNCS 1979) B. Falsafi and M. Lauria, Eds. Berlin, Germany: Springer, 2000, pp. 61–72. [Online]. Available: http://dx.doi.org/10.1007/107201115_5

[55] T. Skeie, O. Lysne, and I. Theiss, "Layered shortest path (LASH) routing in irregular system area networks," in *Proc. 16th Int. Parallel Distrib. Process. Symp. Abstracts CD ROM (IPDPS)*, Fort Lauderdale, FL, USA, Apr. 2002, p. 8.

[56] X.-Y. Lin, Y.-C. Chung, and T.-Y. Huang, "A multiple LID routing scheme for fat-tree-based InfiniBand networks," in *Proc. 18th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Santa Fe, NM, USA, Apr. 2004, p. 11.

[57] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *Proc. 25th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Anchorage, AK, USA, May 2011, pp. 616–627.

[58] Y.-M. Sun, C.-H. Yang, Y.-C. Chung, and T.-Y. Huang, "An efficient deadlock-free tree-based routing algorithm for irregular wormhole-routed networks based on the turn model," in *Proc. 33rd Int. Conf. Parallel Process. (ICPP)*, vol. 1. Montreal, QC, Canada, Aug. 2004, pp. 343–352.

[59] T. Hoefler, T. Schneider, and A. Lumsdaine, "Optimized routing for large-scale InfiniBand networks," in *Proc. 17th Annu. Symp. High Perform. Interconnects (HOTI)*, New York, NY, USA, Aug. 2009, pp. 103–111.

[60] M. D. Schroeder *et al.*, "Autonet: A high-speed, self-configuring local area network using point-to-point links," *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, pp. 1318–1335, Oct. 1991.

[61] J. C. Sancho and A. Robles, "Improving the up*/down* routing scheme for networks of workstations," in *Euro-Par 2000 Parallel Process* (LNCS 1900), A. Bode, T. Ludwig, W. Karl, and R. Wismller, Eds. Berlin, Germany: Springer, 2000, pp. 882–889. [Online]. Available: http://dx.doi.org/10.1007/3-540-44520-X_123

[62] R. Zhou and E. A. Hansen, "Breadth-first heuristic search," *Artif. Intell.*, vol. 170, nos. 4–5, pp. 385–408, 2006.

[63] J. Flich, P. López, M. P. Malumbres, J. Duato, and T. Rokicki, "Combining in-transit buffers with optimized routing schemes to boost the performance of networks with source routing," in *High Performance Computing* (LNCS 1940), M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, Eds. Berlin, Germany: Springer, 2000, pp. 300–309. [Online]. Available: http://dx.doi.org/10.1007/3-540-39999-2_28

[64] J. C. Sancho, A. Robles, and J. Duato, "A new methodology to compute deadlock-free routing tables for irregular networks," in *Network-Based Parallel Computing. Communication, Architecture, Applications* (LNCS 1797), B. Falsafi and M. Lauria, Eds. Berlin, Germany: Springer, 2000, pp. 45–60. [Online]. Available: http://dx.doi.org/10.1007/107201115_4

[65] W. Qiao and L. M. Ni, "Adaptive routing in irregular networks using cut-through switches," in *Proc. Int. Conf. Parallel Process. Vol. 3. Softw.*, vol. 1. Ithaca, NY, USA, Aug. 1996, pp. 52–60.

[66] P. Geoffray and T. Hoefler, "Adaptive routing strategies for modern high performance networks," in *Proc. 16th IEEE Symp. High Perform. Interconnects (HOTI)*, Stanford, CA, USA, 2008, pp. 165–172.

[67] L. Cherkasova, V. Kotov, and T. Rokicki, "Fibre channel fabrics: Evaluation and design," in *Proc. 29th Hawaii Int. Conf. Syst. Sci.*, vol. 1. Jan. 1996, pp. 53–62.

[68] W. J. Dally and C. L. Seitz, "Deadlock free message routing in multiprocessor interconnection networks," California Inst. Technol., Pasadena, CA, USA, Tech. Rep. 5206:TR:86, 1986. [Online]. Available: http://resolver.caltech.edu/CaltechCSTR:1986.5206-tr-86

[69] M. Lang and X. Yuan, "Software defined networking for HPC interconnect and its extension across domains," Los Alamos Nat. Lab., Los Alamos, NM, USA, Tech. Rep. LA-UR 16–21010, 2016. [Online]. Available: http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-16-21010

[70] J. Lee, Z. Tong, K. Achalkar, X. Yuan, and M. Lang, "Enhancing InfiniBand with Openflow-style SDN capability," in *Proc. Int. Conf. High Perform. Comput. Netw. Stor. Anal. (SC)*, Salt Lake City, UT, USA, 2016, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014953

[71] K. Takahashi *et al.*, "Concept and design of SDN-enhanced MPI framework," in *Proc. 4th Eur. Workshop Softw. Defined Netw.*, Bilbao, Spain, Sep. 2015, pp. 109–110.

[72] A. S. Thyagaturu, A. Mercian, M. P. McGarry, M. Reisslein, and W. Kellerer, "Software defined optical networks (SDONs): A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 4, pp. 2738–2786, 4th Quart., 2016.

[73] L. Schares, D. M. Kuchta, and A. F. Benner, "Optics in future data center networks," in *Proc. 18th IEEE Symp. High Perform. Interconnects*, Mountain View, CA, USA, Aug. 2010, pp. 104–108.

[74] A. Benner, "Optical interconnect opportunities in supercomputers and high end computing," in *Proc. Opt. Fiber Commun. Conf.*, Los Angeles, CA, USA, 2012, pp. 1–60. [Online]. Available: http://www.osapublishing.org/abstract.cfm?URI=OFC-2012-OTu2B.4

[75] A. Cohen. *Software-Defined Networking on InfiniBand Networks*. Accessed: May 2017. [Online]. Available: https://www.openfabrics.org/images/eventpresos/2016presentations/302SWDefinedonIB.pdf

[76] M. J. Koop, J. K. Sridhar, and D. K. Panda, "Scalable MPI design over InfiniBand using eXtended reliable connection," in *Proc. Int. Conf. Cluster Comput.*, Tsukuba, Japan, 2008, pp. 203–212.

[77] J. R. Santos, Y. Turner, and G. Janakiraman, "End-to-end congestion control for InfiniBand," in *Proc. 22nd Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 2. San Francisco, CA, USA, Mar. 2003, pp. 1123–1133.

[78] M. Gusat *et al.*, "Congestion control in InfiniBand networks," in *Proc. 13th Annu. Symp. High Perform. Interconnects (HOTI)*, Stanford, CA, USA, Aug. 2005, pp. 158–159.

[79] E. G. Gran *et al.*, "First experiences with congestion control in InfiniBand hardware," in *Proc. 24th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Atlanta, GA, USA, Apr. 2010, pp. 1–12.

[80] E. G. Gran and S.-A. Reinemo, "InfiniBand congestion control: Modelling and validation," in *Proc. 4th Int. ICST Conf. Simulat. Tools Tech. (SIMUTools)*, Barcelona, Spain, 2011, pp. 390–397. [Online]. Available: http://dl.acm.org/citation.cfm?id=2151054.2151122

[81] E. G. Gran *et al.*, "On the relation between congestion control, switch arbitration and fairness," in *Proc. 11th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput. (CCGrid)*, Newport Beach, CA, USA, May 2011, pp. 342–351.

[82] E. Gran *et al.*, "Exploring the scope of the InfiniBand congestion control mechanism," in *Proc. 26th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Shanghai, China, May 2012, pp. 1131–1143.

[83] Q. Liu *et al.*, "The dynamic nature of congestion in InfiniBand," in *Proc. 6th Int. Conf. Workshop Comput. Commun. (IEMCON)*, Vancouver, BC, Canada, Oct. 2015, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jps?tp=&arnumber=7344436

[84] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," Internet Eng. Task Force, Fremont, CA, USA, RFC 5681, Sep. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5681.txt

[85] P. Thaler. *Congestion Spreading: The Dark Side of Link-Based Congestion Control*. Accessed: Nov. 2016. [Online]. Available: www.ieee802.org/3/cm_study/public/september04/thaler_3_0904.pdf

[86] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proc. Int. Conf. High Perform. Comput. Netw. Stor. Anal. (SC)*, Denver, CO, USA, 2013, pp. 1–12.

[87] T. Nachiondo, J. Flich, and J. Duato, "Buffer management strategies to reduce HoL blocking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 6, pp. 739–753, Jun. 2010.

[88] H. Subramoni, P. Lai, S. Sur, and D. K. Panda, "Improving application performance and predictability using multiple virtual lanes in modern multi-core InfiniBand clusters," in *Proc. 39th Int. Conf. Parallel Process. (ICPP)*, San Diego, CA, USA, Sep. 2010, pp. 462–471.

[89] W. L. Guay, S.-A. Reinemo, O. Lysne, and T. Skeie, "dFtree: A fat-tree routing algorithm using dynamic allocation of virtual lanes to alleviate congestion in InfiniBand networks," in *Proc. 1st Int. Workshop Netw. Aware Data Manag. (NDM)*, Seattle, WA, USA, 2011, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2110217.2110219

[90] J. Escudero-Sahuquillo *et al.*, "A new proposal to deal with congestion in InfiniBand-based fat-trees," *J. Parallel Distrib. Comput.*, vol. 74, no. 1, pp. 1802–1819, Jan. 2014.

[91] F. Mizero, M. Veeraraghavan, Q. Liu, R. D. Russell, and J. M. Dennis, "A dynamic congestion management system for InfiniBand networks," *Supercomput. Front. Innov.*, vol. 3, no. 2, pp. 5–20, 2016. [Online]. Available: http://superfri.org/superfri/article/view/91/76

[92] Y. Birk and V. Zdornov, "Improving communication-phase completion times in HPC clusters through congestion mitigation," in *Proc. SYSTOR Israeli Exp. Syst. Conf.*, Haifa, Israel, May 2009, Art. no. 16.

[93] Q. Liu and R. D. Russell, "RGBCC: A new congestion control mechanism for InfiniBand," in *Proc. 24th Euromicro Int. Conf. Parallel Distrib. Netw. Based Process. (PDP)*, Heraklion, Greece, Feb. 2016, pp. 91–100.

[94] Q. Liu, R. D. Russell, and E. G. Gran, "Improvements to the InfiniBand congestion control mechanism," in *Proc. 24th Annu. Symp. High Perform. Interconnects (HOTI)*, Santa Clara, CA, USA, Aug. 2016, pp. 27–36. [Online]. Available: http://www.hoti.org/hoti24/program/

[95] *The Open Group Base Specifications Issue 7*, IEEE Standard 1003.1, 2013. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html

[96] OpenFabrics Alliance. *OpenFabrics Enterprise Distribution (OFED) Overview*. Accessed: Aug. 2014. [Online]. Available: https://www.openfabrics.org/ index.php/openfabrics-software.html

[97] *Understanding iWARP: Eliminating Overhead and Latency in Multi-GB Ethernet Networks*. Neteffect, Las Vegas, NV, USA, Sep. 2007. [Online]. Available: http://download.intel.com/support/network/adapter/pro100/sb/understanding_iwarp.pdf

[98] S. Jang, "High-speed remote direct memory access (RDMA) networking for HPC: Comparative review of 10GbE iWARP and InfiniBand," Margalla Commun., Woodside, CA, USA, Feb. 2010. [Online]. Available: https://www.hpcwire.com/2010/02/24/remote_direct_memory_access_networking_for_hpc_comparative_review_of_10gbe_iwarp_and_infiniband/

[99] B. Metzler, F. Neeser, and P. Frey. (Mar. 2009). *A Software iWARP Driver for OpenFabrics*. [Online]. Available: http://www.openfabrics.org/archives/sonoma2009/monday/softiwarp.pdf

[100] D. Cohen, "Remote direct memory access over the converged enhanced Ethernet fabric: Evaluating the options," in *Proc. IEEE HOT Interconnects*, New York, NY, USA, 2009, pp. 123–130. [Online]. Available: http://www.hoti.org/hoti17/program/slides/Panel/Talpey_HotI_RoCEE.pdf

[101] *Network Direct Service Provider Interface (SPI)*, Microsoft, Redmond, WA, USA, Jul. 2010. [Online]. Available: https://msdn.microsoft.com/en-us/library/cc904397(v=vs.85)aspx

[102] F. Mietke *et al.*, "Analysis of the memory registration process in the mellanox InfiniBand software stack," in *Proc. Eur. Conf. Parallel Process.*, Dresden, Germany, 2006, pp. 124–133.

[103] B. Hausauer and R. Sharp, "RDMA enabled I/O adapter performing efficient memory management," U.S. Patent 11/357 446, Feb. 17, 2006.

[104] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and O. K. Panda, "Host-assisted zero-copy remote memory access communication on InfiniBand," in *Proc. 18th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Santa Fe, NM, USA, Apr. 2004, p. 31.

[105] J. Vienne *et al.*, "Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud computing systems," in *Proc. 20th Annu. Symp. High Perform. Interconnects (HOTI)*, Santa Clara, CA, USA, Aug. 2012, pp. 48–55. [Online]. Available: ieeexplore.ieee.org/document/6299072/

[106] P. MacArthur and R. D. Russell, "A performance study to guide RDMA programming decisions," in *Proc. 14th Int. Conf. High Perform. Comput. Commun. 9th Int. Conf. Embedded Softw. Syst. (HPCC ICESS)*, Liverpool, U.K., Jun. 2012, pp. 778–785. [Online]. Available: ieeexplore.ieee.org/document/6332248

[107] Q. Liu and R. D. Russell, "A performance study of InfiniBand fourteen data rate (FDR)," in *Proc. 22nd High Perform. Comput. Symp. (HPC)*, Tampa, FL, USA, Apr. 2014, p. 16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663526

[108] M. J. Koop, T. Jones, and D. K. Panda, "Reducing connection memory requirements of MPI for InfiniBand clusters: A message coalescing approach," in *Proc. 7th Int. Symp. Cluster Comput. Grid*, Rio de Janeiro, Brazil, 2007, pp. 495–504.

[109] P. Lai, H. Subramoni, S. Narravula, A. Mamidala, and D. K. Panda, "Designing efficient FTP mechanisms for high performance data-transfer over InfiniBand," in *Proc. 38th Int. Conf. Parallel Process. (ICPP)*, Vienna, Austria, 2009, pp. 156–163.

[110] H. Subramoni, P. Lai, R. Kettimuthu, and D. K. Panda, "High performance data transfer in grid environments using GridFTP over InfiniBand," in *Proc. 10th IEEE/ACM Int. Conf. Cluster Cloud Grid Comput. (CCGrid)*, Melbourne, VIC, Australia, 2010, pp. 557–564.

[111] B. Li, P. Zhang, Z. Huo, and D. Meng, "Early experiences with write-write design of NFS over RDMA," in *Proc. Int. Conf. Netw. Archit. Stor.*, Hunan, China, 2009, pp. 303–308.

[112] *Message Passing Interface Forum*. Accessed: Oct. 2014. [Online]. Available: https://mpi-forum.org

[113] *CESM: Community Earth System Model*. Accessed: Oct. 2014. [Online]. Available: http://www2.cesm.ucar.edu/

[114] J. W. Hurrell *et al.*, "The community earth system model: A framework for collaborative research," *Bull. Amer. Meteorol. Soc.*, vol. 94, no. 9, pp. 1339–1360, 2013, doi: 10.1175/BAMS-D-12-00121.1.

[115] R. Dimitrov and A. Skjellum, *Software Architecture and Performance Comparison of MPI/Pro and MPICH* (LNCS 2659). Berlin, Germany: Springer, Mar. 2004. [Online]. Available: https://www.researchgate.net/publication/2933523_Software_architecture_and_performance_comparison_of_MPIPro_and_MPICH

[116] *MPICH*. Accessed: Oct. 2014. [Online]. Available: http://www.mpich.org

[117] W. Gropp and B. Smith, "Chameleon parallel programming tools users manual," Argonne Nat. Lab., Argonne, IL, USA, Tech. Rep. ANL-93/23, Jun. 1993. [Online]. Available: digital.library.unt.edu/ark:/67531/metadc283144

[118] J. Liu *et al.*, "Design and implementation of MPICH2 over InfiniBand with RDMA support," in *Proc. 18th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Santa Fe, NM, USA, Apr. 2004, p. 16.

[119] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," in *Proc. 17th Annu. Int. Conf. Supercomput. (ICS)*, Jun. 2003, pp. 295–304.

[120] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:IJPP.0000029272.69895.c1

[121] *Free: IBM Platform MPI Community Edition*, Int. Bus. Mach. Corporat., Armonk, NY, USA. [Online]. Available: https://www.ibm.com/developerworks/downloads/im/mpi/index.html

[122] *Intel MPI Library*. Accessed: Oct. 2014. [Online]. Available: http://software.intel.com/en-us/intel-mpi-library

[123] B. F. Jamroz, "Asynchronous communication in spectral-element and discontinuous Galerkin methods for atmospheric dynamics—A case study using the high-order methods modeling environment (HOMME-homme_dg_branch)," *Geosci. Model Develop.*, vol. 9, no. 8, pp. 2881–2892, 2016.

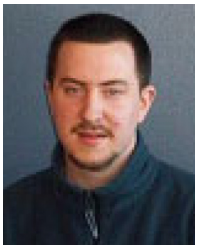
[124] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand: Design alternatives and benefits," in *Proc. 11th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, New York, NY, USA, 2006, pp. 32–39.

[125] *MPI Performance Topics*. Accessed: Aug. 2014. [Online]. Available: https://computing.llnl.gov/tutorials/mpi_performance/

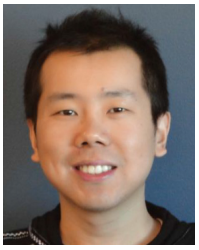
[126] *MVAPICH: MPI Over InfiniBand, Omni-Path, Ethernet/iWARP and RoCE*, Ohio State Univ. Netw. Based Comput. Lab., Columbus, OH, USA, accessed: Sep. 2017. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[127] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges, "InfiniBand scalability in open MPI," in *Proc. 20th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Apr. 2006, p. 100.

[128] R. L. Graham *et al.*, "Open MPI: A high-performance, heterogeneous MPI," in *Proc. Int. Conf. Cluster Comput.*, Barcelona, Spain, 2006, pp. 1–9.
[129] E. D. Brooks, III, "The butterfly barrier," *Int. J. Parallel Program.*, vol. 15, no. 4, pp. 295–307, 1986.
[130] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 1–17, 1988.
[131] Y. Han and R. A. Finkel, "An optimal scheme for disseminating information," in *Proc. Int. Conf. Parallel Process. (ICPP)*, 1988, pp. 198–203.
[132] *InterOperability Laboratory*. Accessed: Aug. 2014. [Online]. Available: http://www.iol.unh.edu
[133] *Extrae*. Accessed: Sep. 2017. [Online]. Available: https://tools.bsc.es/extrae
[134] *Paraver: A Flexible Performance Analysis Tool*. Accessed: Sep. 2017. [Online]. Available: https://tools.bsc.es/paraver
[135] L. Bailey and C. Boyle, *Red Hat Enterprise Linux Performance Tuning Guide*, Red Hat Inc., Raleigh, NC, USA, 2016. [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/
[136] *Perftest*. Accessed: Nov. 2016. [Online]. Available: https://openfabrics.org/downloads/perftest
[137] *Supplement to InfiniBand Architecture Specification Volume 1, Release 1.3: Annex A18: Virtualization*, InfiniBand Trade Assoc. Stand., Beaverton, OR, USA, Sep. 2014. [Online]. Available: http://infinibandta.org
[138] B. Magro, "Recent topics in the IBTA, and a look ahead," presented at the 13th Annu. OpenFabrics Alliance Workshop, Mar. 2017, pp. 1–18. [Online]. Available: https://www.openfabrics.org/images/eventpresos/2017presentations/111_IBTATWG_BMagro.pdf
[139] *Non-Volatile Memory Programming Model, Version 1.1*, Tech. Position Stor. Netw. Ind. Assoc., New Delhi, India, Mar. 2015.
[140] M. Schulz, "The message passing interface: On the road to MPI 4.0 & beyond," presented at the ISC high perform., Jun. 2016, pp. 19–35, accessed: Sep. 2017. [Online]. Available: https://github.com/mpi-forum/mpi-forum.github.io/blob/master/slides/2016/06/2016-06-iscbof.pdf



Patrick MacArthur (S'14) received the B.S. degree in computer science from the University of New Hampshire in 2012, where he is currently pursuing the Ph.D. degree. He has been with the InterOperability Laboratory, University of New Hampshire since 2008. His research interests include high-performance computing and network access to storage class memory.

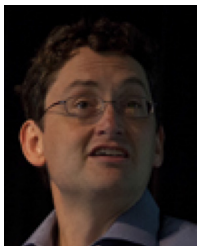


Qian Liu received B.S. and M.S. degrees in computer science from Southwest Jiaotong University in 2003 and 2007, respectively, and the Ph.D. degree in computer science from the University of New Hampshire in 2016. He is currently an Assistant Professor with the Mathematics and Computer Science Department, Rhode Island College.

Robert D. Russell (M'75–LM'15) received the B.A. degree in mathematics and physics from Yale University in 1965 and the Ph.D. degree in computer science from Stanford University in 1972. He was with CERN in Geneva, Switzerland, from 1971 to 1974. In 1975, he joined the faculty with the University of New Hampshire, where he is currently an Emeritus Associate Professor with the Computer Science Department. Since 1999, he has also been associated with the InterOperability Laboratory, University of New Hampshire. His research interests include high-speed interconnects and network-based storage. He is a member of ACM.

Fabrice Mizero received the B.Sc. degree in computer science from Philander Smith College in 2014 and the Master of Engineering degree in computer engineering from the University of Virginia in 2016. He is currently a Software Engineer with the Center for Open Science, Charlottesville, VA, USA. He was with the University of Virginia, where he conducted research in InfiniBand Congestion Control under the supervision of Prof. M. Veeraraghavan.

Malathi Veeraraghavan (M'88–SM'97) is a Professor with the Charles L. Brown Department of Electrical and Computer Engineering, University of Virginia (UVa). After a ten-year career with Bell Laboratories, she served on the faculty with Polytechnic University, Brooklyn, NY, USA, from 1999 to 2002. She served as the Director of the Computer Engineering Program with UVa from 2003 to 2006. She holds 29 patents and has over 90 publications. She was a recipient of six best-paper awards. She served as the Technical Program Committee Co-Chair for the NGN Symposium at IEEE ICC 2013.



John M. Dennis received the M.S. degree in computer science from the University of Colorado Boulder in 1993 and the Ph.D. degree in computer science in 2005. He is a Scientist III with the Computer Information and Systems Laboratory, National Center for Atmospheric Research. He leads a research group that focuses on improving the ability of large scale geoscience applications to utilize current and future computing platforms. His research interests include parallel algorithm and compiler optimization, graph partitioning, and data-intensive computing.