A Measurement Study of Congestion in an InfiniBand Network

Fatma Alali*, Fabrice Mizero*, Malathi Veeraraghavan*, John M. Dennis†
*University of Virginia, Charlottesville, VA, 22904, USA, {fha6np, fm9ab, mv5g}@virginia.edu
‡National Center for Atmospheric Research, Boulder, CO, 80301, USA, dennis@ucar.edu

***Abstract*—This paper presents a measurement study of congestion on a production, highly utilized, 72K-core InfiniBand cluster called Yellowstone. The measurement study consists of a 23-day data collection phase in which port counters of the Yellowstone switches were read multiple times every hour to check for stalls during which the port is unable to send data due to a lack of flow-control credits. A total of 30M data records were obtained and analyzed. Results showed that a significant number of the 100-ms intervals over which a port counter was observed, there were transmission stalls. For example, out of 6M observations of Top-of-Rack (ToR) switch uplink ports, we found that the port was forced to wait for credits in 60% of these 100-ms intervals. Such transmission stalls could increase application execution time, and also decrease cluster utilization. The latter will occur when Message Passing Interface (MPI) Barrier calls are issued for synchronization and communication delays cause one or more MPI ranks to be slower than others.**

***Index Terms*—InfiniBand; Fat-tree; congestion**

I. Introduction

In July 2016, InfiniBand was reported as the cluster interconnect of choice in 70% of the High Performance Computing (HPC) systems deployed in academic, research and government institutions, and as being used in 41% of the Top-500 list of supercomputers [1]. InfiniBand is a switched networking technology that is designed for lossless, low-latency communications. For highly parallelized Message Passing Interface (MPI) applications that are executed on HPC systems, communication delays can become the key determinant of application execution time when computing cores are not limited.

The InfiniBand protocol includes a link-by-link flow control, which is effective in preventing losses in switch buffers, but has an insidious side effect of causing congestion to spread in the network. When an output port P1 of a switch becomes congested, the input-side buffer of another port P2 of the same switch could fill up. This will cause the port of the upstream switch connected to port P2 to be denied flow-control credits, thus effectively reducing the rate of the upstream port. Such a port is referred to as a "victim port" and flows passing through victim ports become "victim flows" (bulk-data flows suffer reduced throughput, and short-messages suffer increased delays).

Studies have shown that network effects, e.g., congestion, can increase variability of the total execution time for large core-count applications [2]. Therefore, many research papers, [3]–[12], have modeled, analyzed, and proposed solutions for InfiniBand congestion control, and evaluated these solutions using simulations or experiments on small testbeds. For example, in prior work [12], we studied the conditions under which congestion occurs by creating various combinations of traffic on a small experimental testbed, and then we developed and evaluated a new Dynamic Congestion Management System (DCMS) solution on this testbed.

In contrast to this prior work, in this paper, we report on a measurement study of congestion in a production HPC cluster. To the best of our knowledge, no such measurement study has been reported in prior work. It is challenging to characterize congestion events because supercomputing centers typically disable congestion control in their InfiniBand clusters. The reason cited for disabling congestion control is that there is no proven study that provides guidance on how to set congestion-control parameters [3]. Therefore, a measurement study of congestion in a production network is not easy.

Specifically, our measurement-based study of congestion was carried out on a highly utilized 72K-core machine called Yellowstone [13]. The Yellowstone network is a fat-tree topology, which consists of Top-of-Rack (ToR) switches, leaf switches, and spine switches. In addition to the main cluster of compute nodes, there is a disk I/O subsystem, and a data analysis and visualization subsystem. A methodology based on observing a port counter called `PortXmitWait` is proposed. For data analysis, a new metric called Forced Idle Time Fraction (FITF) is defined.

Our key contributions are as follows. (i) Our methodology and software offers a means for network administrators to obtain a conservative gauge of the level of congestion in a production network on which congestion control is disabled. (ii) We expected congestion to be predominantly in the disk I/O system, but found that ports in the compute-node cluster also suffered from congestion. In about 60% of the 100-ms intervals in which ToR-switch ports were observed, the ports were stalled waiting for flow-control credits. While in most 100-ms intervals, a port was denied flow-control credits for less than 10% of the interval, there were some instances in which a port was denied credit for significant portions (in the range 60-80%) of the 100-ms interval, with several intervals reaching 100%. Such long stalls in data transmission can impact the completion time of one or more MPI ranks adversely, which can increase execution time of highly parallelized communication-intensive applications that have `MPI_Barrier` calls for synchronizing MPI ranks.

Section II provides background information on InfiniBand

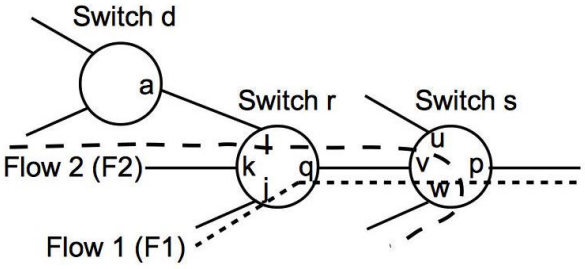


Fig. 1: Illustrative InfiniBand Network [12]

and Yellowstone. Section III describes our measurement study (methodology, data analysis, and metrics). The numerical results are presented in Section IV, and the impact of our findings is discussed in section V. Section VI reviews related work. The paper is concluded in Section VII, and our plan for future work is presented in Section VIII.

II. Background

This section provides background information on InfiniBand networks, and describes the Yellowstone system.

A. InfiniBand

InfiniBand is a packet-switched networking technology designed for high-speed low-latency operation. Packet headers carry 16-bit source and destination Local Identifiers (LIDs), and packets are forwarded by switches with a destination-LID-based table lookup. A centralized subnet manager computes and downloads forwarding tables to the switches. To avoid packet loss, a link-by-link flow-control scheme is used. A transport-layer congestion-control mechanism includes actions at switches, receiving hosts, and sending hosts.

In the link-layer protocol, a transmitter (switch port or host port) is not allowed to send out packets unless the corresponding receiving port has sufficient buffer space. The receiving port sends a Flow Control Packet (FCP) indicating how much space is left in its buffer. If the receiving buffer is full, the transmitter has to wait until it receives an FCP from the receiver.

Congestion control in InfiniBand is based on Explicit Congestion Notification (ECN). The mechanism used to detect congestion at a switch port is not defined in the InfiniBand specification, but rather, it is left up to the vendors. When congestion is detected on a port P, packets transmitted out on port P are marked by setting the Forward ECN (FECN) bit in the transport-layer header. The rate at which the packets are marked is controlled by a configurable parameter called the `Marking_Rate`. When a receiving host receives marked packets for a flow, it sets a Backward ECN (BECN) bit in the acknowledgment (ACK) or other packets that are being sent in the opposite direction. When the sending host receives a BECN-marked packet for a particular flow, the sender reduces its sending rate according to a mechanism that dynamically adjusts inter-packet injection delay.

To understand how the link-by-link flow control mechanism causes the effects of a port's congestion to spread even to ports that have no shared flows with the congested port, consider the example shown in Fig. 1 [12]. Assume that port p of switch s becomes congested because the aggregate incoming rate of packets destined to port p exceeds the port capacity. This can happen when multiple high-throughput transfers (e.g., disk read/write and checkpointing) destined to the same switch port occur concurrently, as demonstrated in experiments in our prior work [12]. Now, assume flow F1 traverses ports j and q of switch r and ports v and p of switch s. When port p gets congested, the buffer on the incoming side of port v of switch s will start to fill up with flow-F1 packets. When this buffer fills up, the rate at which FCPs are generated by port v of switch s to port q of switch r, to offer the latter credits for packet transmission, will decrease. Effectively, the rate of port q of switch r is lowered. Now, consider flow F2, which traverses ports k and q of switch r and ports v and w of switch s. The rate of F2 will be reduced even though this flow does not traverse the congested port p. Flow F2 is a victim flow, and port q of switch r is a victim port. This example illustrates how the presence of the link-by-link flow control algorithm causes the effects of a congested port to spread to other parts of the network.

This type of spreading of the effects of a congested port does not occur in IP/Ethernet networks. This is because in IP/Ethernet networks, when a switch buffer is full, packets are simply dropped. There is no credit-based flow control mechanism, and therefore a link transmitter can freely send packets to a link receiver causing a buffer to overflow. This approach has the advantage of not creating victim ports. Consider the example of flows F1 and F2 shown in Fig. 1. If the switches were Ethernet-based or IP routers, when port p of switch s becomes congested, its buffer will overflow, which causes F1 packets to be dropped. Flow F2, which is not destined to the congested port p, will not experience any dropped packets and hence will be unaffected, unlike in the InfiniBand network. However, if flow F2 also used port p, it would suffer packet losses. Therefore, there are victim flows in TCP/IP networks but not victim ports; the latter is worse as flows that do not even traverse the congested port become victimized.

Therefore, the work presented here is only relevant to InfiniBand-based HPC systems, which as noted in Section I, is deployed widely in academic, research and government institutions. Commercial datacenters typically use Ethernet-based interconnects. The reason for this difference is that parallel programs written for scientific research typically use MPI and require low-latency communications, while commercial datacenters typically support applications that use TCP sockets for communications.

B. Yellowstone

Yellowstone is a high-performance IBM iDataPlex cluster, where the network interconnect is a Mellanox InfiniBand full

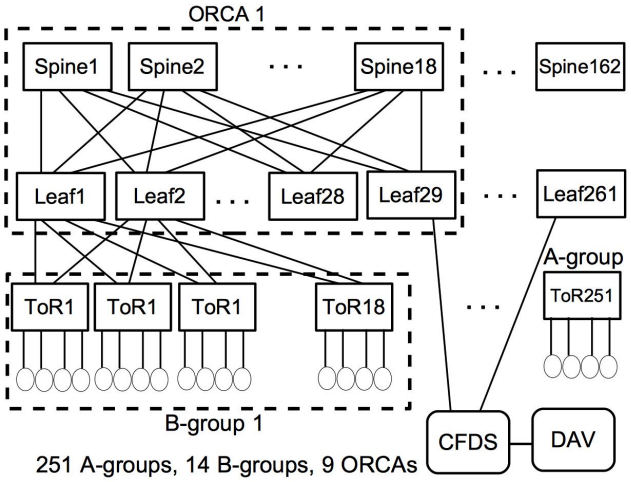


Fig. 2: Yellowstone topology

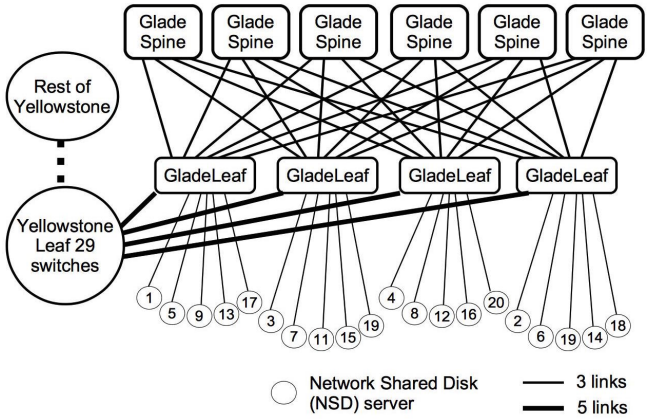


Fig. 3: Glade subsystem topology

fat-tree. All links carry 4-lane Fourteen Data Rate (FDR) (56 Gbps) signals.

Fig. 2 shows the Yellowstone topology. Each ToR switch has 36 ports, 18 of which are connected to compute hosts, and the remaining 18 are connected to leaf switches in ORCA[1] racks. Each ToR switch with its 18 compute hosts is called an A-group. The term B-group is used to represent a group of 18 A-groups. All ToR switches in a B-group are connected to the same two leaf switches of an ORCA rack, which consists of 29 leaf switches and 18 spine switches. For example, ToR1 to ToR18 of B-group 1 are connected to Leaf 1 and Leaf 2 of ORCA 1, Leaf 30 and Leaf 31 of ORCA 2, etc. In other words, all ToR switches of B-group 1 are connected to the first two leaf switches of each of the 9 ORCAs. Similarly, ToR19 to ToR36 of B-group 2 are connected to Leaf 3 and Leaf 4 of ORCA 1, Leaf 32 and Leaf 33 of ORCA 2, etc. Each leaf switch also has 36 ports, 18 of which are connected via downlinks to ToR switches, and the remaining 18 are connected to spine switches of the same ORCA rack to which the leaf switch belongs.

There is a disk I/O subsystem shown as Centralized Filesystems and Data Storage (CFDS) in Fig. 2. This system is named "Glade." The Glade leaf switches are connected to the 29^{th} leaf switch in each ORCA as shown in Fig. 3. There is also a Data Analysis and Visualization (DAV) subsystem, which is connected to the CFDS subsystem, as shown in Fig. 2.

Fig. 3 shows the details of the Glade disk I/O subsystem (marked as CFDS in Fig. 2). The Glade subsystem has 6 Glade spine and 4 Glade leaf switches, which are connected to GPFS Network Shared Disk (NSD) servers.

III. Measurement study of congestion

This section describes the basis for our methodology, the data collection process executed on Yellowstone, our data analysis method, and the definition of a new metric used as a proxy measure for congestion.

[1]ORCA does not appear to be an acronym.

A. Basis for methodology

InfiniBand switches implement two types of port counters among others, namely, `PortXmitCongTime`, and `PortXmitWait`. `PortXmitCongTime` is the amount of time a port has spent in a congested state, while `PortXmitWait` indicates the amount of time a port has data to send but lacks flow-control credits.

If an administrator disables congestion control, as is the case on the Yellowstone system, `PortXmitCongTime` counters will not register any values, and hence cannot be used to measure congestion. However, we contend that `PortXmitWait` counters can be used as a proxy indicator for congestion, and offer the following justification.

First, how does a switch decide that a port is congested? As noted in Section II, the specific mechanism used to detect congestion is left up to vendor implementation. An approach proposed by Gran and Reinemo [7] for congestion detection is as follows: when the fill-ratio of an input-port Virtual Output Queue (VOQ)[2] holding packets destined to a particular output port exceeds a certain predefined level, the switch will consider the output port to be congested. The predefined level for fill ratio is related to the InfiniBand standard congestion-control parameter called `Threshold`. This parameter controls how quickly a switch reacts to congestion, with a value 15 indicating the fastest reaction to congestion onset, and a value 0 for disabled congestion control. Therefore, whether a switch declares a port to be in a congested state or not (i.e., whether or not packets sent on that port should be marked with FECNs) is dependent on these parameters. The `PortXmitCongTime` counter increases for ports declared to be in a congested state.

Second, if the `PortXmitWait` counter of an upstream port increases, does it necessarily mean that there is congestion on some corresponding downstream switch port? The answer is yes. The `PortXmitWait` counter of an upstream port (e.g., port *q* of switch *r* in Fig. 1) increases only when the transmitter has no flow-control credits from the receiver to send packets. This can only happen if the whole input-side buffer of the

[2]VOQs are used to avoid the Head-of-Line (HOL) blocking problem.

corresponding receiving port (e.g., port *v* of switch *s*) is full. But if the input-side buffer is full, it means that irrespective of the `Threshold` parameter setting, the fill ratio of at least one VOQ in this input-side buffer is guaranteed to have crossed the threshold by the time the downstream switch denies flow-control credits to the upstream switch. In other words, some downstream-switch port would have been declared as being congested before a corresponding `PortXmitWait` counter of an upstream switch port starts to increase.

Finally, consider the flipped question: does a downstream congested port necessarily cause the `PortXmitWait` counter of an upstream port to increase? The answer is no because the predefined level for the fill ratio of input-side VOQs of a downstream port is typically less than 100%, which means congestion would be declared even before the input-side buffer fills up completely, while `PortXmitWait` counter of the upstream port will not increase until the corresponding receiving port has no space in its input-side buffer. The implication is that an increasing `PortXmitWait` counter is a conservative monitor for congestion, not an overly optimistic one. In other words, there were likely more congestion events than reported here by a reading of the `PortXmitWait` counters. Therefore, applying our methodology, if large and/or frequent increases in `PortXmitWait` counters are observed, network administrators can be sure that there is network congestion.

In general, since InfiniBand switch buffer sizes are small, e.g., on the order of 64 KB, which is sufficient to hold just 32 frames [14], it is likely that soon after congestion is declared for a downstream-switch port, the `PortXmitWait` counter on at least one corresponding upstream port will increase. Our previous paper [12] presented graphs that show the simultaneous increase in `PortXmitCongTime` counter of a downstream port and the `PortXmitWait` counter of an upstream port. A similar observation was made in an experimental study by Subramoni et al. [15], in which the `PortXmitWait` counter was seen to register increases when congestion was caused by an All-to-All Remote Direct Memory Access (RDMA) communications event.

B. Script implementation and execution for data collection

A Linux command called `perfquery` is available to read InfiniBand port counters. This command was used to read the `PortXmitWait` counter of switch ports at periodic intervals. Since the number of ports in Yellowstone is very large, and we had only limited CPU time to run this script, we decided to observe a set of randomly selected ports for short durations every hour. We wrote a shell script to first randomly select a switch port from across the whole Yellowstone topology, including the Glade and DAV subsystems, and then to issue the `perfquery` command to read the `PortXmitWait` counter of the selected port multiple times with a specified inter-query interval, before selecting the next port for observation.

The approximately 700 switches were divided into 6 sets, to allow for concurrent monitoring. Six instances of the script were executed, with each script running on a different host. The sets of switches are disjoint, and hence no script can randomly draw a port outside its domain.

The steps executed by the script are as follows: (i) Parse the script input arguments to obtain the lower and upper bounds of the assigned set of (approximately 115) switch LIDs. (ii) Select a switch LID at random (using uniform distribution) from the set of assigned LIDs, and select one of the switch ports also at random (using uniform distribution). (iii) Execute a command to check if the selected port is in an operational state. If not, select another switch and port. (iv) Reset the `PortXmitWait` counter for the selected port using the `perfquery` command with the reset (`-R`) flag. (v) Submit a sequence of 100 `perfquery` calls to read the `PortXmitWait` counter of the randomly selected port, with an inter-call time spacing of 100 ms (i.e., the process sleeps for 100 ms after each command completes). Each sequence of 100 calls is referred to as a *round*. In other words, each port was observed for approximately 10 sec. (vi) Append the results of each query into an open comma separated values (CSV) file. (vii) After the 100th query, select another switch and port at random, and repeat steps (iii)-(v).

A `cron` job was used to run the 6 instances of the script (one corresponding to each set of approximately 115 switch LIDs) every hour. After 20 minutes of execution, the scripts terminate. To prevent excessive disk I/O, the results of each `perfquery` are appended to a file that is stored on the local filesystem of the compute node on which the script is run. When the total size of the file exceeds a threshold, the file is copied to permanent storage.

This data collection process was executed for a period of 23 days in March 2016. The aggregate size of the CSV files was 2 GB, and each file had approximately 5M records[3].

C. Data analysis

Each row in the CSV file, which we refer to as a *record*, stores the parameters as well as results of one `perfquery` call. Record i in round r is defined to have the following fields:

$$\{t_r, s_r, p_r, t^q_{r,i}, t^p_{r,i}, W_{r,i}\} \tag{1}$$

where the first three parameters are common for a round r: t_r is the time instant when the 100-call `perfquery` round r was initiated, s_r is the switch LID (unique across the Yellowstone network), p_r is the port number on switch s_r, where $p_r \in \mathbf{P}_{s_r}$ and $\mathbf{P}_{s_r}$ is the set of all ports on switch s_r, and the remaining parameters are specific to a query i within round r: $t^q_{r,i}$ is the time recorded just before the i^{th} `perfquery` call of round r was issued, $t^p_{r,i}$ is the turnaround time of the i^{th} query of round r (i.e., the difference between the time recorded when the `perfquery` call returns and $t^q_{r,i}$), and $W_{r,i}$ is the `PortXmitWait` counter value of port p_r on switch s_r at time t, where t is estimated to be $(t^q_{r,i} + t^p_{r,i}/2)$ since the exact time when the switch s_r received the message to read the `PortXmitWait` counter is not directly measurable.

[3]The data is publicly available at this web site: http://pages.shanti.virginia.edu/HSN/tma17-data/

All records were merged into one CSV file, and a switch-port-classification script was executed to classify the switch LID into one of 7 categories (ToR, Leaf, Spine, GladeLeaf, GladeSpine, DAVSpine, and DAVLeaf) based on the role of the switch in the Yellowstone topology. The port number p_r in each record was used to determine whether the port was connected to an `up` or `down` link in the fat-tree topology of Yellowstone. For example, the `up` link of a ToR switch connects the ToR switch to a leaf switch, while the `down` link of a ToR switch connects the ToR switch to a host. This switch-port-classification script parsed the output of the `ibnetdiscover` tool to map LIDs and ports into the 7 switch categories and `up/down` classifications, respectively, and then added two columns to each record in the merged CSV file indicating the switch category and `up/down` port classification.

The augmented record, denoted $\mathbb{R}_{r,i}$, in the merged CSV file has the following fields:

$$\mathbb{R}_{r,i} \triangleq \{\sigma_r, \lambda_r, t_r, s_r, p_r, t^q_{r,i}, t^p_{r,i}, W_{r,i}\} \tag{2}$$

where the first two fields are new relative to the fields in the original CSV files, as specified in (1). The first field σ is the switch category and has one of these 7 values: ToR, Leaf, Spine, GladeLeaf, GladeSpine, DAVSpine, and DAVLeaf, and the second field λ represents the port type as `up` or `down`.

D. Metrics

We define a new term *Forced Idle Time Fraction (FITF)*, represented by $F_r[i]$, as the fraction of time when a transmitter is made to wait for flow-control credits from the receiver between the i^{th} and $(i+1)^{th}$ `perfquery` calls within querying round r of a switch port. FITF vector $\mathbf{F}_r$ is defined as a vector with 99 entries corresponding to the 100 `perfquery` calls issued to a switch port in one *round*.

Consider two consecutive records $\mathbb{R}_{r,i+1}$ and $\mathbb{R}_{r,i}$ in the final CSV file that belong to the same round r. The i^{th} element of the vector $\mathbf{F}_r$ is given by:

$$F_r[i] = \frac{\tau\,(W_{r,i+1} - W_{r,i})}{(t^q_{i+1} + \frac{t^p_{i+1}}{2}) - (t^q_i + \frac{t^p_i}{2})}, 1 \le i \le 99 \tag{3}$$

where τ corresponds a system tick, which is roughly 22 ns for an FDR switch port.

The $\mathbf{F}_r$ vectors from the various querying rounds of different switch ports are combined based on their σ switch category and λ port type to create a matrix $\mathbf{A}_{\sigma,\lambda}$ shown below:

$$\mathbf{A}_{\sigma,\lambda} = \begin{pmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \\ \vdots \\ \mathbf{F}_{k_{\sigma,\lambda}} \end{pmatrix} \tag{4}$$

where the switches in all $k_{\sigma,\lambda}$ rounds belong to category σ, and the queried ports belong to the category λ. There are 11 (σ, λ) combinations **C**={(Spine, down), (Leaf, up), (Leaf, down), (ToR, up), (ToR, down), (DAVSpine, down), (DAVLeaf, up), (DAVLeaf, down), (GladeSpine, down), (GladeLeaf, up), (GladeLeaf, down)}. There are no (Spine, up), (DAVSpine, up), or (GladeSpine, up) possibilities since Spine is the top-level of the topology.

TABLE I: Total number of querying rounds for each switch category and port type

Port type	Switch category						
	Spine	Leaf	ToR	DAV-Spine	DAV-Leaf	Glade-Spine	Glade-Leaf
down	63129	63527	60217	1422	1192	1455	1165
up	NA	63679	60131	NA	1133	NA	1172

IV. Numerical results

Section IV-A shows examples of how the `PortXmitWait` counter grows within a 10-sec interval for two ports. Section IV-B presents a comparison of FITF across the 11 switch-port categories. Section IV-C presents an analysis of the rounds in which FITF was non-zero (without considering outliers), and Section IV-D discusses the outlier FITF values.

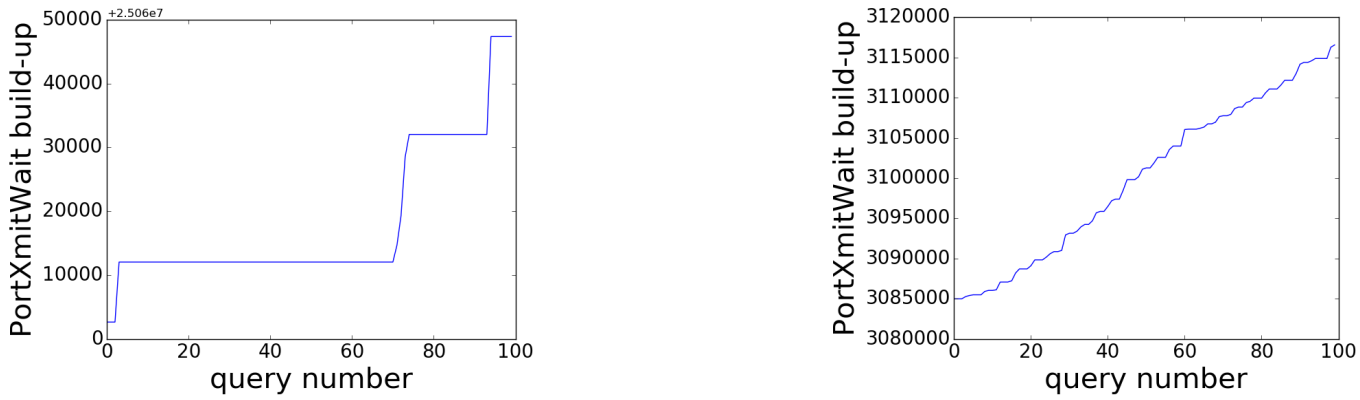
A. Examples illustrating `PortXmitWait` growth

Fig. 4 shows two examples of how `PortXmitWait` counter value increases within one querying round. Even though the counter was reset at the beginning of each round, the `PortXmitWait` value returned from the first `perfquery` call was not zero because there was a time interval between the resetting action and when the call to read the counter was issued. In Fig. 4a, for example, the counter stays unchanged at 25072028 until the 71^{st} query, and then increases to 25092026, where it stays unchanged until the 94^{th} query (the values shown on the y-axis of Fig. 4a should be added to 2.506e7, as shown at the top of the axis, to obtain the actual `PortXmitWait` counter readings). On the other hand, Fig. 4b shows that the counter value increases almost continually within each 100-ms interval.

B. Zero vs. non-zero FITF values

Table I shows the number of querying rounds $k_{\sigma,\lambda}$ for each combination of switch category σ and port type λ. Since the number of ToR switches is roughly equal to the number of leaf switches, the number of querying rounds for these two types of switches were roughly the same. There are much fewer DAV and Glade switches, and hence there were fewer querying rounds for these switches.

Fig. 5 shows a stacked bar-plot for the percentage of zero and non-zero FITF values for each switch category and port type. In the ToR switches, the number of non-zero FITF values is greater than the number of zero FITF values for both up and down port types. For example, a ToR switch down port was queried in 60217 rounds, which means the `PortXmitWait` counter value growth over 100 ms was observed 60217×99 times, which is roughly 6M. In these 6M observations, we found that the port was forced to wait for credits (which is an indication of congestion) in 60% of these 100-ms intervals. The implication of a ToR switch downlink port being held up waiting for credits is that the corresponding Host Channel Adapter (HCA) buffer was full.



(a) Switch category: ToR, port type: down

(b) Switch category: Spine, port type: down

Fig. 4: `PortXmitWait` build-up

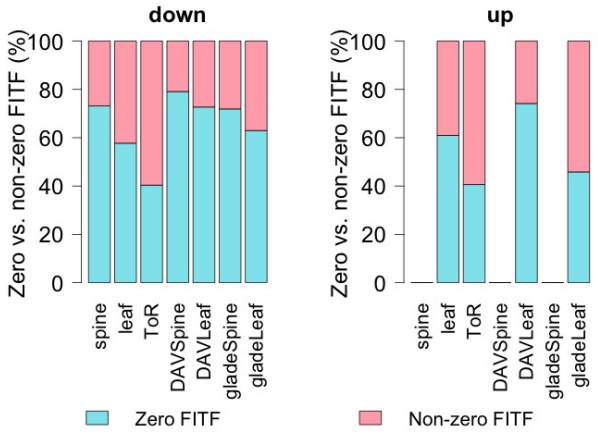


Fig. 5: The percentage of zero and non-zero FITF values for each switch category and link (port) type

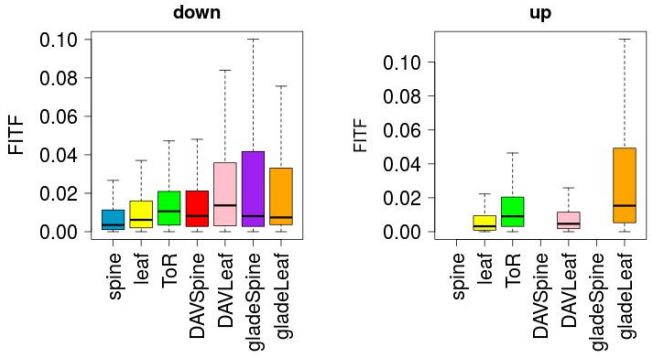


Fig. 6: The maximum FITF value per non-zero round for each switch category and link type (outliers removed)

Fig. 5 shows that for GladeLeaf switches, there is more congestion on the uplink ports than on the downlink ports. A possible explanation for congestion on the uplink ports is as follows. The Glade disk I/O subsystem shown in Fig. 3 offers users multiple filesystems such as `scratch` and `home`. The `scratch` filesystem is used for temporary data storage. Users move data from the `scratch` filesystem to the `home` filesystem for permanent storage. These data transfers could cause congestion on the uplink GladeLeaf ports, as the two filesystems are likely to be connected to nodes served by different GladeLeaf switches, which would necessitate transfers through GladeSpine switches.

Fig. 5 shows that on DAVLeaf switches, congestion occurs at equal rates on downlinks and uplinks. One DAVLeaf switch is connected to one GladeLeaf switch, via 6 links, and these links are classified as downlinks on both switches. Uplinks connect DAVLeaf switches to DAVSpine switches. Since users move data for analysis from the Glade disk I/O nodes into the DAV nodes, and store analysis results back into the Glade disk I/O nodes, traffic flows both ways on the DAV-to-Glade links.

We expected congestion to be predominantly in the Glade disk I/O subsystem switches, but the results in Fig 5 show that ToR switch ports experience congestion even though they primarily serve compute nodes. This could be the effect of cascading rate reductions caused by the link-by-link flow control procedure. Victim ports could occur anywhere in the network far from a congested port.

C. Non-zero FITF values

To gain some insights into the fraction of each 100-ms interval that a port was denied credits, we undertook a study of just those rounds in which the FITF value was non-zero for at least one 100-ms interval. We refer to these rounds as *non-zero rounds*.

Fig. 6 shows a boxplot of the maximum non-zero FITF values across all non-zero rounds of all switch ports belonging to a particular switch category and port type. Using (4), a maximum FITF was computed for each row of the matrix $\mathbf{A}_{\sigma,\lambda}$, i.e., the maximum value across the elements of vector $\mathbf{F}_r$, $1 \leq r \leq k_{\sigma,\lambda}$. From Fig. 6, in which outliers were removed, we conclude that in most of the 10-sec observation

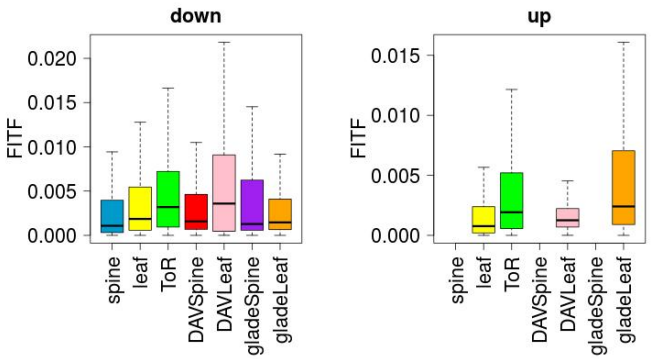


Fig. 7: The average FITF value per non-zero round for each switch category and link type (outliers removed)

rounds, a port was denied credits in less than 10% of a 100-ms interval.

The plot in Fig. 6 shows that the Glade subsystem switch ports had the highest variability in maximum FITF. This is because bulk data transfers do not occur continuously, but, when they do occur, these transfers can cause congestion, and increase the forced idle time significantly. For example, the maximum FITF across all querying intervals of GladeLeaf downlink ports was 85%. The DAV leaf switches also experienced a high variability due to the bulk data transfers that occur occasionally. The ToR switches have more variability when compared to the compute subsystem leaf and spine switches.

Fig. 7 shows a boxplot of the average FITF across non-zero rounds for all switch ports belonging to a particular switch category and port type. The averaging was done across only the non-zero FITF values. The ToR switches experience more variability in average FITF when compared to the spine and leaf switches. This is an interesting result as it is the opposite of our expectation that leaf and spine switch links would have more congestion due to oversubscription at the higher levels of the fat tree.

D. Outlier FITF values

The key finding presented in this section is that there were some 100-ms intervals in which the port was completely stalled, i.e., the transmitter was prevented from sending data due to a lack of flow-control credits for the whole 100-ms interval. Occurrences of these extreme cases (outliers) were not included in the boxplots of Fig. 6 and 7 to allow for a better visualization of the differences between the switch-port categories. Nevertheless, these outliers are important because highly parallelized communication-intensive applications with MPI synchronization calls, e.g., climate-science applications that use 1000 cores or more, will experience significant increases in their execution delays if such long stalls occur. An *outlier* is defined as a data point that lies 1.5 $\times$ IQR above the 75% number or below the 25% number, where IQR is Inter-Quartile range (difference between the 25% and 75% values).

Table II shows the outliers statistics for the different switch ports categories. For example, there were 44807 querying rounds of ToR switch uplink ports that had a non-zero maximum FITF, and out of the 44807 querying rounds, there was a total of 2561 outliers. The `max` rows in Table II for both downlink and uplink ports show FITF values that are greater than 1, which represents 100%. For an explanation of how this is possible, we provide the details of one example query.

Consider the example query shown in Table III. The rows correspond to the values of two consecutive records $\mathbb{R}_{r,1}$ and $\mathbb{R}_{r,2}$ per (2). To find FITF, the difference between the `PortXmitWait` counter readings W in the two consecutive records is computed, and the difference is divided by the time difference between the *estimated* times at which the counter was read. As shown in (3), the exact time at which the port counter is read cannot be determined accurately. Instead, the time is estimated by halving the query turnaround times $t^p_{r,1}$, and $t^p_{r,2}$. If the first query took a shorter time reaching the switch, and the second query took a longer time reaching the switch than estimated by the halving operation, then the time between the two consecutive readings of the port counter could be longer than the estimated time difference. Such an occurrence would result in an estimate of FITF that is greater than 1. Effectively, such FITF values should be interpreted as the port being stalled, waiting for flow-control credits, during the entire 100-ms interval.

V. Discussion: Impact of findings

In this section, we describe the potential value of our findings for InfiniBand network operations and for scientific applications. Specifically, the results could be interesting both to InfiniBand network administrators, and to scientific researchers who run applications on InfiniBand HPC systems.

Network operations: As stated in Section I, most InfiniBand HPC administrators do not enable congestion control due to a perceived lack of proven methods for setting parameters. But if congestion control is disabled, network administrators are blind as to whether or not congestion is occurring, because the `PortXmitCongTime` counters will not register any values. Our proposed method for using `PortXmitWait` counters as a proxy, and our software for collecting the data and analyzing the collected data to derive FITF values, can thus be used by administrators to gain insights into the state of their networks. Further, administrators could correlate FITF values with application execution time. Such a correlation study would show whether or not high variability in application execution can be attributed to network congestion. If a high correlation is observed, network administrators could then enable congestion control.

Administrators can test the various solutions (for setting congestion-control parameters) that have been proposed in research literature (reviewed in Section VI). For example, we designed, implemented and evaluated a scheme called Dynamic Congestion Management System (DCMS) [12]. Recall from Section II that the switch has a parameter called

TABLE II: Details about outliers left out of the boxplot in Fig. 6 and Fig. 7

Port type		Switch category						
		Spine	Leaf	ToR	DAV-Spine	DAV-Leaf	glade-Spine	glade-Leaf
down	min	0.027	0.037	0.047	0.049	0.086	0.102	0.078
	max	0.865	1.179	0.879	0.432	0.409	0.807	0.852
	Number of querying rounds with non-zero maximum FITF	2765	40258	51814	565	550	806	627
	Number of outliers	670	2771	3185	56	43	119	81
	Number of FITF values ≥ 1	none	37	none	none	none	none	none
up	min	NA	0.022	0.046	NA	0.026	NA	0.116
	max	NA	1.163	1.158	NA	0.424	NA	0.959
	Number of querying rounds with non-zero maximum FITF	NA	47167	44807	NA	959	NA	1132
	Number of outliers	NA	3751	2561	NA	111	NA	128
	Number of FITF values ≥ 1	none	23	6	none	none	none	none

TABLE III: Example of two consecutive records belong to the same querying round r

Switch category (σ)	Port type (λ)	i	t^q	t^p	W	W_{i+1}-W_i	$(t^q_{i+1} + \frac{t^p_{i+1}}{2}) - (t^q_i + \frac{t^p_i}{2})$	FITF
Leaf	down	1	1.45640989347e+18	18350267	2426695474	6656811	124199424	1.179
Leaf	down	2	1.4564098936e+18	24841529	2433352285			

`Marking_Rate`. The key idea of DCMS is to dynamically adjust this parameter based on whether-or-not victim flows are being created by a congestion event. This solution was found to be effective through an experimental evaluation. For example, the evaluation results showed that a victim flow's throughput decreased from 8 Gbps to 2.67 Gbps because of congestion at a switch port that the victim flow did not even traverse. However, when DCMS detected the congestion and took action by adjusting the switch `Marking_Rate`, the congestion-causing flows dropped their sending rates, and the victim flow throughput rebounded back to 8 Gbps.

Applications: We offer three examples of the impact of our FITF-related findings on scientific-research applications. *First*, Petrini et al. [16] presented evidence that even frequent, short-duration congestion events can have a detrimental impact on applications that have fine-grained communications (frequent short message exchanges). In fact, our results show that in Yellowstone, while FITF values were of short duration, i.e., less than 10% (see Fig. 6), the network link stalls occurred quite frequently (Fig. 5 showed that the `PortXmitWait` counter registered increases in a significant percentage of the observed 100-ms intervals).

Second, we experimented with the climate-science High-Order Method Modeling Environment (HOMME) application, and found that it issues a synchronization MPI call 140K times in one 90-core run (which is a small configuration; in actual runs by scientists, 1000s of cores are used for each run). If messages sent by one source MPI rank took longer to reach their destination MPI ranks (e.g., these messages were affected by one of the outlier congestion events reported in Table II), then all MPI ranks will be delayed when an `MPI_Barrier` or `MPI_Waitall` call is encountered. This could lead to a significant increase in the total execution time of the application. Furthermore, since synchronization calls cause MPI ranks to stop processing and wait until the slowest MPI rank completes, communication delays can result in under-utilization of compute nodes.

Third, Subramoni et al. [15] stated that their proposed technique for reducing "the amount of network contention observed during the All-to-All/FFT operations" resulted in a "9% improvement in the communication time of P3DFFT at 512 processes."

In *summary*, our methodology, new metric, data collection and analysis software, and specific numerical findings for Yellowstone, are all useful contributions to the InfiniBand provider and user community.

VI. Related work

As noted in Section I, there have been many studies of InfiniBand congestion control [3]–[12], but none of these papers measured congestion on a production InfiniBand network as we have done. Some papers [8], [15], [17] used `PortXmitWait` as an indicator of congestion, which is also the basis of our FITF metric.

Papers that suggested mechanisms for seting congestion-control parameters include work by Pfister et al. [3], Gusat et al. [4], and Gran et al. [6]. All three studies used simulations to characterize the effects of various parameter settings on application metrics.

Several papers, such as VOQsw [18], dFtree [19], vFtree [20], BBQ [21], Flow2SL [10], and pFtree [22], proposed using InfiniBand virtual lanes to combat congestion.

Other papers proposed new/enhanced congestion control mechanisms, unconstrained by the InfiniBand standard. Yan et al. [5] proposed a Power Increase and Power Decrease (PIPD) function for controlling sending rate. Michelogiannakis et al. [23] proposed a Channel Reservation Protocol (CRP) to prevent congestion. Russell et al. [24] developed Red and Green light-Based Congestion Control (RGBCC), which is less sensitive to small changes in parameters when compared to the standard InfiniBand congestion control mechanism. Liu et al. [25] proposed improvements to the InfiniBand standard by adding a Link Bandwidth Availability Report (LABR).

Finally, modifications were proposed to job schedulers, such as SLURM, to take into account network conditions [17] when assigning MPI ranks to nodes. The work by Bhatele et al. [26], which demonstrated the impact of neighborhood jobs on application performance in Cray networks offers a good model for a similar study in InfiniBand networks.

VII. Summary and Conclusions

This paper presented a 23-day measurement study of congestion on a production, highly utilized, 72K-core InfiniBand cluster called Yellowstone. We proposed a methodology based on reading the `PortXmitWait` counter of ports, and a new metric called Forced Idle Time Fraction (FITF). While congestion is likely caused by bulk data flows in the disk I/O and Data Analysis and Visualization subsystems of Yellowstone, our findings were that ports of even ToR switches that serve compute nodes suffered from such flow-control related stalls. In about 60% of the 100-ms intervals in which ToR-switch ports were observed, the ports had to wait for flow-control credits, but most of these transmission stalls were shorter than 10 ms. Prior work showed that short but frequent congestion events are detrimental to applications with fine-grained communications. Also, some congestion events lasted for significant portions (in the range 60-80%) of the 100-ms intervals, with several events even reaching 100%. Such events can significantly increase execution time of applications that have MPI synchronization calls.

VIII. Future Work

The focus of this paper was to determine whether congestion occurs in production networks. While preliminary answers were offered based on prior work, we propose to undertake a thorough simulation study to answer the following questions in a more rigorous manner: (i) Under what conditions (combinations of network traffic) does congestion occur? (ii) How does `PortXmitWait` growth relate to congestion levels at which the impact on applications becomes important? (iii) What is the relationship between FITF (as a measure of congestion) and application performance?

Acknowledgment

This work was supported by NSF ACI-1340910, CNS-1405171, and CNS-1531065.

References

[1] "Life in the Fast Lane: InfiniBand Continues to Reign as HPC Interconnect of Choice." http://blog.infinibandta.org/tag/top500/, July 8, 2016.

[2] T. Hoefler, T. Schneider, and A. Lumsdaine, "The impact of network noise at large-scale communication performance," in *Parallel & Distributed Processing IPDPS,IEEE Intl. Symp. on*, pp. 1–8, IEEE, 2009.

[3] G. Pfister, M. Gusat, W. Denzel, D. Craddock, N. Ni, W. Rooney, T. Engbersen, R. Luijten, R. Krishnamurthy, and J. Duato, "Solving hot spot contention using InfiniBand architecture congestion control," in *High Perf. Interconnects for Dist. Comp., Research Triangle Park,2005*.

[4] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato, "Congestion control in InfiniBand networks," in *High Perf. Interconnects (HOTI), 13th Symp. on*, pp. 158–159, 2005.

[5] S. Yan, G. Min, and I. Awan, "An enhanced congestion control mechanism in InfiniBand networks for high performance computing systems," in *Adv. Info. Networking and App.(AINA) 20th Intl. Conf. on*, 2006.

[6] E. Gran, M. Eimot, S.-A. Reinemo, T. Skeie, O. Lysne, L. Huse, and G. Shainer, "First experiences with congestion control in InfiniBand hardware," in *Parallel Dist. Proc. (IPDPS), IEEE Intl. Symp. on*, 2010.

[7] E. G. Gran and S.-A. Reinemo, "InfiniBand Congestion Control: Modelling and Validation," in *Proceedings of the 4th Intl. ICST Conf. on Simulation Tools and Techniques*, SIMUTools '11, 2011.

[8] 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. of the First Intl. Workshop on Network-aware Data Management*, NDM '11.

[9] E. Gran, S.-A. Reinemo, O. Lysne, T. Skeie, E. Zahavi, and G. Shainer, "Exploring the scope of the InfiniBand congestion control mechanism," in *Parallel Dist. Proc. Symp. (IPDPS), IEEE 26th Intl.*, 2012.

[10] J. Escudero-Sahuquillo, P. J. Garcia, F. J. Quiles, S.-A. Reinemo, T. Skeie, O. Lysne, and J. Duato, "A new proposal to deal with congestion in InfiniBand-based fat-trees," *Journal of Parallel and Dist. Comp.,2014*.

[11] N. R. Tallent, A. Vishnu, H. Van Dam, J. Daily, D. J. Kerbyson, and A. Hoisie, "Diagnosing the causes and severity of one-sided message contention," in *ACM SIGPLAN Notices*, no. 8, pp. 130–139, ACM, 2015.

[12] F. Mizero, M. Veeraraghavan, Q. Liu, R. Russell, and J. Dennis, "A Dynamic Congestion Management System for InfiniBand Networks," *Supercomputing frontiers and innovations*, vol. 3, no. 2, 2016.

[13] "NCAR-Wyoming Supercomputing Center (NWSC) Yellowstone." https://www2.cisl.ucar.edu/resources/yellowstone.

[14] H. Yi, S. Park, M. Kim, and K. Jeon, "An efficient buffer allocation technique for virtual lanes in InfiniBand networks," in *Proceedings of the 2Nd International Conference on Human.Society@Internet*, HSI'03, (Berlin, Heidelberg), pp. 272–281, Springer-Verlag, 2003.

[15] H. Subramoni, A. Venkatesh, K. Hamidouche, K. Tomko, and D. Panda, "Impact of InfiniBand DC transport protocol on energy consumption of all-to-all collective algorithms," in *IEEE 23rd Annual Symp. on High-Performance Interconnects*, pp. 60–67, IEEE, 2015.

[16] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Supercomputing, ACM/IEEE Conf.,2003*.

[17] H. Subramoni, D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, M. Arnold, and D. K. Panda, "Design of network topology aware scheduling services for large InfiniBand clusters," in *IEEE Intl. Conf. on Cluster Computing (CLUSTER)*, pp. 1–8, IEEE, 2013.

[18] M. Gomez, J. Flich, A. Robles, P. Lopez, and J. Duato, "VOQSW: a methodology to reduce hol blocking in InfiniBand networks," in *Parallel and Dist. Proc. Symp., 2003. Proceedings. Intl.*, p. 10, Apr. 2003.

[19] 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 *Proceedings of the First Intl. Workshop on Network-aware Data Management*, NDM '11, 2011.

[20] W. L. Guay, B. Bogdanski, S.-A. Reinemo, O. Lysne, and T. Skeie, "vFtree - a fat-tree routing algorithm using virtual lanes to alleviate congestion," in *Parallel Dist. Proc. Symp. (IPDPS), IEEE Intl.*, 2011.

[21] P. Y. Segura, J. Escudero-Sahuquillo, C. G. Requena, P. J. Garcia, F. J. Quiles, and J. Duato, "BBQ: a straightforward queuing scheme to reduce HoL-blocking in high-performance hybrid networks," in *European Conf. on Parallel Processing*, pp. 699–712, Springer, 2013.

[22] F. Zahid, E. G. Gran, B. Bogdanski, B. D. Johnsen, and T. Skeie, "Partition-aware routing to improve network isolation in infiniband based multi-tenant clusters," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM Intl. Symp. on*, pp. 189–198, IEEE, 2015.

[23] G. Michelogiannakis, N. Jiang, D. Becker, and W. J. Dally, "Channel reservation protocol for over-subscribed channels and destinations," in *Proceedings of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[24] Q. Liu and R. D. Russell, "RGBCC: A New Congestion Control Mechanism for InfiniBand," in *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro Intl. Conf. on* , 2016.

[25] Q. Liu, R. D. Russell, and E. G. Gran, "Improvements to the InfiniBand Congestion Control Mechanism," in *High-Performance Interconnects (HOTI), 2016 IEEE 24th Annual Symp. on*, pp. 27–36, IEEE, 2016.

[26] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proceedings of the Intl. Conf. on High Perf. Computing, Networking, Storage and Analysis*, SC '13, 2013.